The 30 Year Horizon

Manuel Bronstein
James Davenport
Albrecht Fortenbacher
Jocelyn Gidry
Michael Monagan
Jonathan Steinbach
Stephen Watt

William Burge
Michael Dewar
Patrizia Gianni
Richard Jenks
Scott Morrison
Robert Sutor
Jim Wen

Timothy Daly
Martin Dunstan
Johannes Grabmeier
Larry Lambe
William Sit
Barry Trager
Clifton Williamson

Volume 10: Axiom Algebra: Packages
This book and the Axiom software is licensed as follows:

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of The Numerical ALgorithms Group Ltd. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Inclusion of names in the list of credits is based on historical information and is as accurate as possible. Inclusion of names does not in any way imply an endorsement but represents historical influence on Axiom development.

<table>
<thead>
<tr>
<th>Name</th>
<th>Name</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Michael Albaugh</td>
<td>Cyril Alberga</td>
<td>Roy Adler</td>
</tr>
<tr>
<td>Christian Aistleitner</td>
<td>Richard Anderson</td>
<td>George Andrews</td>
</tr>
<tr>
<td>S.J. Atkins</td>
<td>Henry Baker</td>
<td>Martin Baker</td>
</tr>
<tr>
<td>Stephen Balzac</td>
<td>Yuri Baransky</td>
<td>David R. Barton</td>
</tr>
<tr>
<td>Thomas Baruchel</td>
<td>Gerald Baumgartner</td>
<td>Gilbert Baumslag</td>
</tr>
<tr>
<td>Michael Becker</td>
<td>Nelson H. F. Beebe</td>
<td>Jay Belanger</td>
</tr>
<tr>
<td>David Bindel</td>
<td>Fred Blair</td>
<td>Vladimir Bondarenko</td>
</tr>
<tr>
<td>Mark Botch</td>
<td>Raoul Bourquin</td>
<td>Alexandre Bouyer</td>
</tr>
<tr>
<td>Karen Braman</td>
<td>Peter A. Broadbery</td>
<td>Martin Brock</td>
</tr>
<tr>
<td>Manuel Bronstein</td>
<td>Stephen Buchwald</td>
<td>Florian Bundschuh</td>
</tr>
<tr>
<td>Luanne Burns</td>
<td>William Burge</td>
<td>Ralph Byers</td>
</tr>
<tr>
<td>Quentin Carpent</td>
<td>Robert Caviness</td>
<td>Bruce Char</td>
</tr>
<tr>
<td>Ondrej Certik</td>
<td>Tzu-Yi Chen</td>
<td>Cheekai Chin</td>
</tr>
<tr>
<td>David V. Chudnovsky</td>
<td>Gregory V. Chudnovsky</td>
<td>Mark Clements</td>
</tr>
<tr>
<td>James Cloos</td>
<td>Jia Zhao Cong</td>
<td>Josh Cohen</td>
</tr>
<tr>
<td>Christophe Conil</td>
<td>Don Coppersmith</td>
<td>George Corliss</td>
</tr>
<tr>
<td>Robert Corless</td>
<td>Gary Cornell</td>
<td>Meino Cramer</td>
</tr>
<tr>
<td>Jeremy Du Croz</td>
<td>David Cyganski</td>
<td>Nathaniel Daly</td>
</tr>
<tr>
<td>Timothy Daly Sr.</td>
<td>Timothy Daly Jr.</td>
<td>James H. Davenport</td>
</tr>
<tr>
<td>David Day</td>
<td>James Demmel</td>
<td>Didier Deshommes</td>
</tr>
<tr>
<td>Michael Dewar</td>
<td>Jack Dongarra</td>
<td>Jean Della Dora</td>
</tr>
<tr>
<td>Gabriel Dos Reis</td>
<td>Claire DiCrescendo</td>
<td>Sam Dooley</td>
</tr>
<tr>
<td>Lionel Ducos</td>
<td>Iain Duff</td>
<td>Lee Duhem</td>
</tr>
<tr>
<td>Martin Dunstan</td>
<td>Brian Dupee</td>
<td>Dominique Duval</td>
</tr>
<tr>
<td>Robert Edwards</td>
<td>Heow Eide-Goodman</td>
<td>Lars Erickson</td>
</tr>
<tr>
<td>Richard Fateman</td>
<td>Bertfried Fauser</td>
<td>Stuart Feldman</td>
</tr>
<tr>
<td>John Fletcher</td>
<td>Brian Ford</td>
<td>Albrecht Fortenbacher</td>
</tr>
<tr>
<td>George Frances</td>
<td>Constantine Frangos</td>
<td>Timothy Freeman</td>
</tr>
<tr>
<td>Korrinn Fu</td>
<td>Marc Gaetano</td>
<td>Rudiger Gebauer</td>
</tr>
<tr>
<td>Van de Geijn</td>
<td>Kathy Gerber</td>
<td>Patricia Gianni</td>
</tr>
<tr>
<td>Gustavo Goertkin</td>
<td>Samantha Goldrich</td>
<td>Holger Gollan</td>
</tr>
<tr>
<td>Teresa Gomez-Diaz</td>
<td>Laureano Gonzalez-Vega</td>
<td>Stephen Gortler</td>
</tr>
<tr>
<td>Johannes Grabmeier</td>
<td>Matt Grayson</td>
<td>Klaus Ebbe Grue</td>
</tr>
<tr>
<td>James Griesmer</td>
<td>Vladimir Grinberg</td>
<td>Oswald Gschnitzer</td>
</tr>
<tr>
<td>Ming Gu</td>
<td>Jocelyn Guidry</td>
<td>Gaetan Hache</td>
</tr>
<tr>
<td>Steve Hague</td>
<td>Satoshi Hamaguchi</td>
<td>Sven Hammarling</td>
</tr>
<tr>
<td>Mike Hansen</td>
<td>Richard Hanson</td>
<td>Richard Harke</td>
</tr>
<tr>
<td>Bill Hart</td>
<td>Vilya Harvey</td>
<td>Martin Hassner</td>
</tr>
<tr>
<td>Arthur S. Hathaway</td>
<td>Dan Hatton</td>
<td>Waldek Hebisch</td>
</tr>
<tr>
<td>Karl Hegbloom</td>
<td>Ralf Hemmecke</td>
<td>Henderson</td>
</tr>
<tr>
<td>Antoine Hersen</td>
<td>Roger House</td>
<td>Gernot Hueber</td>
</tr>
<tr>
<td>Pietro Igli</td>
<td>Alejandro Jakubi</td>
<td>Richard Jenks</td>
</tr>
<tr>
<td>William Kahan</td>
<td>Kyriakos Kalorkoti</td>
<td>Kai Kaminski</td>
</tr>
</tbody>
</table>
# Contents

1 Chapter Overview ........................................... 1

2 Chapter A .................................................. 3
   package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis ............. 3
      AffineAlgebraicSetComputeWithGroebnerBasis (AFALGGRO) ............... 4
   package AFALGRES AffineAlgebraicSetComputeWithResultant .................. 8
      AffineAlgebraicSetComputeWithResultant (AFALGRES) ...................... 9
   package AF AlgebraicFunction ........................................... 13
      AlgebraicFunction (AF) ........................................... 14
   package INTHERAL AlgebraicHermiteIntegration .............................. 19
      AlgebraicHermiteIntegration (INTHERAL) ................................ 19
   package INTALG AlgebraicIntegrate ....................................... 23
      AlgebraicIntegrate (INTALG) ....................................... 24
   package INTAF AlgebraicIntegration ...................................... 30
      AlgebraicIntegration (INTAF) ...................................... 32
   package ALGMANIP AlgebraicManipulations .................................. 34
      AlgebraicManipulations (ALGMANIP) .................................. 35
   package ALGMFACT AlgebraicMultFact ..................................... 40
      AlgebraicMultFact (ALGMFACT) ..................................... 41
   package ALGPKG AlgebraPackage ....................................... 42
      AlgebraPackage (ALGPKG) ....................................... 44
   package ALGFACT AlgFactor ........................................... 53
      AlgFactor (ALGFACT) ........................................... 55
   package INTPACK AnnaNumericalIntegrationPackage .......................... 57
      AnnaNumericalIntegrationPackage (INTPACK) ............................ 59
   package OPTPACK AnnaNumericalOptimizationPackage ....................... 69
      AnnaNumericalOptimizationPackage (OPTPACK) ........................ 71
   package ODEPACK AnnaOrdinaryDifferentialEquationPackage ............... 79
      AnnaOrdinaryDifferentialEquationPackage (ODEPACK) .................. 81
   package PDEPACK AnnaPartialDifferentialEquationPackage ................ 89
      AnnaPartialDifferentialEquationPackage (PDEPACK) .................... 91
   package ANY1 AnyFunctions1 ......................................... 96
      AnyFunctions1 (ANY1) ......................................... 97
   package API ApplicationProgramInterface ................................ 99
CONTENTS

ApplicationProgramInterface (API) ............................................. 106
package APPRULE ApplyRules .................................................. 107
  ApplyRules (APPRULE) ..................................................... 109
package APPLYORE ApplyUnivariateSkewPolynomial ...................... 112
  ApplyUnivariateSkewPolynomial (APPLYORE) ........................... 113
package ASSOCEQ AssociatedEquations .................................... 114
  AssociatedEquations (ASSOCEQ) ........................................ 115
package PMPRED AttachPredicates ........................................... 118
  AttachPredicates (PMPRED) ............................................. 119
package AXSERV AxiomServer .................................................. 120
  AxiomServer (AXSERV) .................................................. 121

3 Chapter B ................................................................. 137
package BALFACT BalancedFactorisation ................................... 137
  BalancedFactorisation (BALFACT) ....................................... 138
package BOP1 BasicOperatorFunctions1 ................................... 139
  BasicOperatorFunctions1 (BOP1) ....................................... 141
package BEZIER Bezier ........................................................ 144
  Bezier (BEZIER) .......................................................... 162
package BEZOUT BezoutMatrix ............................................... 163
  BezoutMatrix (BEZOUT) .................................................. 165
package BLUPACK BlowUpPackage ........................................... 169
  BlowUpPackage (BLUPACK) ............................................... 170
package BOUNDZRO BoundIntegerRoots ................................... 175
  BoundIntegerRoots (BOUNDZRO) ....................................... 177
package BRILL BrillhartTests ................................................. 179
  BrillhartTests (BRILL) ................................................ 180

4 Chapter C ................................................................. 185
package CARTEN2 CartesianTensorFunctions2 ............................ 185
  CartesianTensorFunctions2 (CARTEN2) ................................ 186
package CHVAR ChangeOfVariable ............................................ 187
  ChangeOfVariable (CHVAR) ............................................. 188
package CPIMA CharacteristicPolynomialInMonogenicalAlgebra ....... 191
  CharacteristicPolynomialInMonogenicalAlgebra (CPIMA) ............ 193
package CHARPOL CharacteristicPolynomialPackage .................... 194
  CharacteristicPolynomialPackage (CHARPOL) ......................... 195
package IBACHIN ChineseRemainderToolsForIntegralBases ............ 196
  ChineseRemainderToolsForIntegralBases (IBACHIN) ................. 197
package CVMP CoerceVectorMatrixPackage ................................ 201
  CoerceVectorMatrixPackage (CVMP) .................................... 202
package COMBF CombinatorialFunction .................................... 204
  CombinatorialFunction (COMBF) ....................................... 207
package CDEN CommonDenominator ......................................... 219
  CommonDenominator (CDEN) ........................................... 221
package COMMONOP CommonOperators ...................................... 222
CONTENTS

CommonOperators (COMMONOP) ........................................... 223
package COMMUPC CommuteUnivariatePolynomialCategory ............... 228
  CommuteUnivariatePolynomialCategory (COMMUPC) ....................... 229
package COMPFAC ComplexFactorization .................................. 230
  ComplexFactorization (COMPFAC) ....................................... 231
package COMPLEX2 ComplexFunctions2 .................................. 233
  ComplexFunctions2 (COMPLEX2) ......................................... 235
package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation ... 235
  ComplexIntegerSolveLinearPolynomialEquation (CINTSLPE) ............ 237
package COMPLPAT ComplexPattern ...................................... 238
  ComplexPattern (COMPLPAT) ........................................... 239
package CPMATCH ComplexPatternMatch .................................. 240
  ComplexPatternMatch (CPMATCH) ....................................... 241
package CRFP ComplexRootFindingPackage ................................ 243
  ComplexRootFindingPackage (CRFP) ................................... 244
package CMPLXRT ComplexRootPackage .................................. 257
  ComplexRootPackage (CMPLXRT) ....................................... 258
package CTRIGMNP ComplexTrigonometricManipulations .................. 259
  ComplexTrigonometricManipulations (CTRIGMNP) ........................ 260
package ODECONST ConstantLODE ........................................ 263
  ConstantLODE (ODECONST) ............................................. 264
package COORDSYS CoordinateSystems ................................... 266
  CoordinateSystems (COORDSYS) ........................................ 268
package CRAPACK CRApackage ............................................. 272
  CRApackage (CRAPACK) ................................................ 273
package CYCLES CycleIndicators ......................................... 275
  CycleIndicators (CYCLES) ............................................. 294
package CSTTOOLS CyclicStreamTools .................................... 299
  CyclicStreamTools (CSTTOOLS) ........................................ 300
package CYCLOTOM CyclotomicPolynomialPackage ........................ 302
  CyclotomicPolynomialPackage (CYCLOTOM) ............................ 303
package CAD CylindricalAlgebraicDecompositionPackage ............... 305
  CylindricalAlgebraicDecompositionPackage (CAD) ....................... 306
package CADU CylindricalAlgebraicDecompositionUtilities ............ 309
  CylindricalAlgebraicDecompositionUtilities (CADU) .................... 311

5 Chapter D ........................................................................... 313
package DFINTTLS DefiniteIntegrationTools ............................. 313
  DefiniteIntegrationTools (DFINTTLS) .................................. 314
package DEGRED DegreeReductionPackage ................................ 320
  DegreeReductionPackage (DEGRED) ..................................... 321
package DTP DesingTreePackage .......................................... 322
  DesingTreePackage (DTP) ................................................ 324
package DIOSP DiophantineSolutionPackage ............................. 333
  DiophantineSolutionPackage (DIOSP) .................................. 334
package DIRPROD2 DirectProductFunctions2 ................................ 338
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DirectProductFunctions2 (DIRPROD2)</td>
<td>339</td>
</tr>
<tr>
<td>package DLP DiscreteLogarithmPackage</td>
<td>341</td>
</tr>
<tr>
<td>DiscreteLogarithmPackage (DLP)</td>
<td>342</td>
</tr>
<tr>
<td>package DISPLAY DisplayPackage</td>
<td>344</td>
</tr>
<tr>
<td>DisplayPackage (DISPLAY)</td>
<td>345</td>
</tr>
<tr>
<td>package DDFACT DistinctDegreeFactorize</td>
<td>348</td>
</tr>
<tr>
<td>DistinctDegreeFactorize (DDFACT)</td>
<td>349</td>
</tr>
<tr>
<td>package DFSFUN DoubleFloatSpecialFunctions</td>
<td>354</td>
</tr>
<tr>
<td>DoubleFloatSpecialFunctions (DFSFUN)</td>
<td>359</td>
</tr>
<tr>
<td>The Exponential Integral</td>
<td>374</td>
</tr>
<tr>
<td>En:(PI,R)→OPR</td>
<td>379</td>
</tr>
<tr>
<td>The Ei Function</td>
<td>380</td>
</tr>
<tr>
<td>The Fresnel Integral[Pear56, Losc60]</td>
<td>406</td>
</tr>
<tr>
<td>package DBLRESP DoubleResultantPackage</td>
<td>410</td>
</tr>
<tr>
<td>DoubleResultantPackage (DBLRESP)</td>
<td>411</td>
</tr>
<tr>
<td>package DRAWCX DrawComplex</td>
<td>412</td>
</tr>
<tr>
<td>DrawComplex (DRAWCX)</td>
<td>414</td>
</tr>
<tr>
<td>package DRAWHACK DrawNumericHack</td>
<td>418</td>
</tr>
<tr>
<td>DrawNumericHack (DRAWHACK)</td>
<td>419</td>
</tr>
<tr>
<td>package DROPT0 DrawOptionFunctions0</td>
<td>420</td>
</tr>
<tr>
<td>DrawOptionFunctions0 (DROPT0)</td>
<td>421</td>
</tr>
<tr>
<td>package DROPT1 DrawOptionFunctions1</td>
<td>425</td>
</tr>
<tr>
<td>DrawOptionFunctions1 (DROPT1)</td>
<td>426</td>
</tr>
<tr>
<td>package D01AGNT d01AgentsPackage</td>
<td>427</td>
</tr>
<tr>
<td>d01AgentsPackage (D01AGNT)</td>
<td>451</td>
</tr>
<tr>
<td>package D01WGTS d01WeightsPackage</td>
<td>457</td>
</tr>
<tr>
<td>d01WeightsPackage (D01WGTS)</td>
<td>458</td>
</tr>
<tr>
<td>package D02AGNT d02AgentsPackage</td>
<td>464</td>
</tr>
<tr>
<td>d02AgentsPackage (D02AGNT)</td>
<td>490</td>
</tr>
<tr>
<td>package D03AGNT d03AgentsPackage</td>
<td>496</td>
</tr>
<tr>
<td>d03AgentsPackage (D03AGNT)</td>
<td>512</td>
</tr>
<tr>
<td>package EP EigenPackage</td>
<td>515</td>
</tr>
<tr>
<td>EigenPackage (EP)</td>
<td>516</td>
</tr>
<tr>
<td>package EF ElementaryFunction</td>
<td>522</td>
</tr>
<tr>
<td>ElementaryFunction (EF)</td>
<td>535</td>
</tr>
<tr>
<td>package DEFINTEF ElementaryFunctionDefiniteIntegration</td>
<td>553</td>
</tr>
<tr>
<td>ElementaryFunctionDefiniteIntegration (DEFINTEF)</td>
<td>554</td>
</tr>
<tr>
<td>package LODEEF ElementaryFunctionLODESolver</td>
<td>558</td>
</tr>
<tr>
<td>ElementaryFunctionLODESolver (LODEEF)</td>
<td>560</td>
</tr>
<tr>
<td>package ODEEF ElementaryFunctionODESolver</td>
<td>565</td>
</tr>
<tr>
<td>ElementaryFunctionODESolver (ODEEF)</td>
<td>567</td>
</tr>
<tr>
<td>package SIGNEF ElementaryFunctionSign</td>
<td>572</td>
</tr>
<tr>
<td>ElementaryFunctionSign (SIGNEF)</td>
<td>574</td>
</tr>
<tr>
<td>package EFSTRUC ElementaryFunctionStructurePackage</td>
<td>578</td>
</tr>
</tbody>
</table>

6 Chapter E  515
<table>
<thead>
<tr>
<th>Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ElementaryFunctionStructurePackage (EFSTRUC)</td>
<td>579</td>
</tr>
<tr>
<td>package INTEF ElementaryIntegration</td>
<td>588</td>
</tr>
<tr>
<td>ElementaryIntegration (INTEF)</td>
<td>589</td>
</tr>
<tr>
<td>package RDEEF ElementaryRischDE</td>
<td>597</td>
</tr>
<tr>
<td>ElementaryRischDE (RDEEF)</td>
<td>599</td>
</tr>
<tr>
<td>package RDEEFS ElementaryRischDESystem</td>
<td>607</td>
</tr>
<tr>
<td>ElementaryRischDESystem (RDEEFS)</td>
<td>608</td>
</tr>
<tr>
<td>package ELFUTS EllipticFunctionsUnivariateTaylorSeries</td>
<td>610</td>
</tr>
<tr>
<td>EllipticFunctionsUnivariateTaylorSeries (ELFUTS)</td>
<td>611</td>
</tr>
<tr>
<td>package EQ2 EquationFunctions2</td>
<td>613</td>
</tr>
<tr>
<td>EquationFunctions2 (EQ2)</td>
<td>614</td>
</tr>
<tr>
<td>package ERROR ErrorFunctions</td>
<td>615</td>
</tr>
<tr>
<td>ErrorFunctions (ERROR)</td>
<td>617</td>
</tr>
<tr>
<td>package GBEUCLID EuclideanGroebnerBasisPackage</td>
<td>619</td>
</tr>
<tr>
<td>EuclideanGroebnerBasisPackage (GBEUCLID)</td>
<td>642</td>
</tr>
<tr>
<td>package EVALCYC EvaluateCycleIndicators</td>
<td>654</td>
</tr>
<tr>
<td>EvaluateCycleIndicators (EVALCYC)</td>
<td>655</td>
</tr>
<tr>
<td>package ESCONT ExpertSystemContinuityPackage</td>
<td>656</td>
</tr>
<tr>
<td>ExpertSystemContinuityPackage (ESCONT)</td>
<td>657</td>
</tr>
<tr>
<td>package ESCONT1 ExpertSystemContinuityPackage1</td>
<td>663</td>
</tr>
<tr>
<td>ExpertSystemContinuityPackage1 (ESCONT1)</td>
<td>664</td>
</tr>
<tr>
<td>package ESTOOLS ExpertSystemToolsPackage</td>
<td>665</td>
</tr>
<tr>
<td>ExpertSystemToolsPackage (ESTOOLS)</td>
<td>667</td>
</tr>
<tr>
<td>package ESTOOLS1 ExpertSystemToolsPackage1</td>
<td>675</td>
</tr>
<tr>
<td>ExpertSystemToolsPackage1 (ESTOOLS1)</td>
<td>676</td>
</tr>
<tr>
<td>package ESTOOLS2 ExpertSystemToolsPackage2</td>
<td>677</td>
</tr>
<tr>
<td>ExpertSystemToolsPackage2 (ESTOOLS2)</td>
<td>678</td>
</tr>
<tr>
<td>package EXPR2 ExpressionFunctions2</td>
<td>679</td>
</tr>
<tr>
<td>ExpressionFunctions2 (EXPR2)</td>
<td>680</td>
</tr>
<tr>
<td>package EXPRSOL ExpressionSolve</td>
<td>681</td>
</tr>
<tr>
<td>Bugs</td>
<td>681</td>
</tr>
<tr>
<td>ExpressionSolve (EXPRSOL)</td>
<td>683</td>
</tr>
<tr>
<td>package ES1 ExpressionSpaceFunctions1</td>
<td>685</td>
</tr>
<tr>
<td>ExpressionSpaceFunctions1 (ES1)</td>
<td>686</td>
</tr>
<tr>
<td>package ES2 ExpressionSpaceFunctions2</td>
<td>687</td>
</tr>
<tr>
<td>ExpressionSpaceFunctions2 (ES2)</td>
<td>688</td>
</tr>
<tr>
<td>package EXPRODE ExpressionSpaceODESolver</td>
<td>689</td>
</tr>
<tr>
<td>ExpressionSpaceODESolver (EXPRODE)</td>
<td>691</td>
</tr>
<tr>
<td>package OMEXPR ExpressionToOpenMath</td>
<td>695</td>
</tr>
<tr>
<td>ExpressionToOpenMath (OMEXPR)</td>
<td>696</td>
</tr>
<tr>
<td>package EXPR2UPS ExpressionToUnivariatePowerSeries</td>
<td>702</td>
</tr>
<tr>
<td>ExpressionToUnivariatePowerSeries (EXPR2UPS)</td>
<td>704</td>
</tr>
<tr>
<td>package EXPRTUBE ExpressionTubePlot</td>
<td>711</td>
</tr>
<tr>
<td>ExpressionTubePlot (EXPRTUBE)</td>
<td>712</td>
</tr>
<tr>
<td>package EXP3D Export3D</td>
<td>716</td>
</tr>
<tr>
<td>Export3D (EXP3D)</td>
<td>717</td>
</tr>
</tbody>
</table>
package E04AGNT e04AgentsPackage ............................ 719
e04AgentsPackage (E04AGNT) .......................... 721

Chapter F ............................................. 727
package FACTFUNC FactoredFunctions ............................. 727
FactoredFunctions (FACTFUNC) ............................... 728
package FR2 FactoredFunctions2 ..................................... 729
FactoredFunctions2 (FR2) .................................. 730
package FRUTIL FactoredFunctionUtilities .............................. 734
FactoredFunctionUtilities (FRUTIL) .............................. 735
package FACUTIL FactoringUtilities ..................................... 736
FactoringUtilities (FACUTIL) ................................ 737
package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber ...................................... 740
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber (FACTEXT) ...................................... 741
package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber ...................................... 744
FactorisationOverPseudoAlgebraicClosureOfRationalNumber (FACTRN) ...................................... 745
package FGLMICPK FGLMIfCanPackage .................................. 749
FGLMIfCanPackage (FGLMICPK) .................................. 750
package FORDER FindOrderFinite ........................................ 752
FindOrderFinite (FORDER) .................................... 753
package FAMR2 FiniteAbelianMonoidRingFunctions2 ......................... 754
FiniteAbelianMonoidRingFunctions2 (FAMR2) .......................... 755
package FDIV2 FiniteDivisorFunctions2 .................................. 757
FiniteDivisorFunctions2 (FDIV2) .................................. 758
package FFACTOR FiniteFieldFactorization .................................. 759
FiniteFieldFactorization (FFACCTOR) .................................. 760
package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect .................................. 765
FiniteFieldFactorizationWithSizeParseBySideEffect (FFFACTSE) .................................. 766
package FFF FiniteFieldFunctions ........................................ 771
FiniteFieldFunctions (FFF) ..................................... 772
package FFHOM FiniteFieldHomomorphisms ................................ 778
FiniteFieldHomomorphisms (FFHOM) ................................ 779
package FFPOLY FiniteFieldPolynomialPackage ................................ 786
FiniteFieldPolynomialPackage (FFPOLY) ................................ 787
package FFPOLY2 FiniteFieldPolynomialPackage2 ................................ 808
FiniteFieldPolynomialPackage2 (FFPOLY2) ................................ 809
package FSLSPE FiniteFieldSolveLinearPolynomialEquation .................. 812
FiniteFieldSolveLinearPolynomialEquation (FSLSPE) ...................... 813
package FFSQFR FiniteFieldSquareFreeDecomposition ......................... 814
FiniteFieldSquareFreeDecomposition (FFSQFR) ...................... 815
package FLAGG2 FiniteLinearAggregateFunctions2 .......................... 818
FiniteLinearAggregateFunctions2 (FLAGG2) .......................... 819
package FLASORT FiniteLinearAggregateSort .............................. 822
FiniteLinearAggregateSort (FLASORT) ............................ 823
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>package FSAGG2 FiniteSetAggregateFunctions2</td>
<td>826</td>
</tr>
<tr>
<td>FiniteSetAggregateFunctions2 (FSAGG2)</td>
<td>827</td>
</tr>
<tr>
<td>package FLOATCP FloatingComplexPackage</td>
<td>828</td>
</tr>
<tr>
<td>FloatingComplexPackage (FLOATCP)</td>
<td>830</td>
</tr>
<tr>
<td>package FLOATRP FloatingRealPackage</td>
<td>832</td>
</tr>
<tr>
<td>FloatingRealPackage (FLOATRP)</td>
<td>834</td>
</tr>
<tr>
<td>package FCPAK1 FortranCodePackage1</td>
<td>836</td>
</tr>
<tr>
<td>FortranCodePackage1 (FCPAK1)</td>
<td>838</td>
</tr>
<tr>
<td>package FOP FortranOutputStackPackage</td>
<td>841</td>
</tr>
<tr>
<td>FortranOutputStackPackage (FOP)</td>
<td>842</td>
</tr>
<tr>
<td>package FORT FortranPackage</td>
<td>844</td>
</tr>
<tr>
<td>FortranPackage (FORT)</td>
<td>845</td>
</tr>
<tr>
<td>package FRIDEAL2 FractionIdealFunctions2</td>
<td>848</td>
</tr>
<tr>
<td>FractionIdealFunctions2 (FRIDEAL2)</td>
<td>849</td>
</tr>
<tr>
<td>package FFFG FractionFreeFastGaussian</td>
<td>850</td>
</tr>
<tr>
<td>FractionFreeFastGaussian (FFFG)</td>
<td>852</td>
</tr>
<tr>
<td>package FFFGF FractionFreeFastGaussianFractions</td>
<td>862</td>
</tr>
<tr>
<td>FractionFreeFastGaussianFractions (FFFG)</td>
<td>863</td>
</tr>
<tr>
<td>package FRAC2 FractionFunctions2</td>
<td>866</td>
</tr>
<tr>
<td>FractionFunctions2 (FRAC2)</td>
<td>867</td>
</tr>
<tr>
<td>package FRNAAF2 FramedNonAssociativeAlgebraFunctions2</td>
<td>868</td>
</tr>
<tr>
<td>FramedNonAssociativeAlgebraFunctions2 (FRNAAF2)</td>
<td>869</td>
</tr>
<tr>
<td>package FSPECF FunctionalSpecialFunction</td>
<td>870</td>
</tr>
<tr>
<td>FunctionalSpecialFunction (FSPECF)</td>
<td>872</td>
</tr>
<tr>
<td>differentiation of special functions</td>
<td>877</td>
</tr>
<tr>
<td>package FFCAT2 FunctionFieldCategoryFunctions2</td>
<td>880</td>
</tr>
<tr>
<td>FunctionFieldCategoryFunctions2 (FFCAT2)</td>
<td>881</td>
</tr>
<tr>
<td>package FFINTBAS FunctionFieldIntegralBasis</td>
<td>882</td>
</tr>
<tr>
<td>FunctionFieldIntegralBasis (FFINTBAS)</td>
<td>884</td>
</tr>
<tr>
<td>package PMASSFS FunctionSpaceAssertions</td>
<td>887</td>
</tr>
<tr>
<td>FunctionSpaceAssertions (PMASSFS)</td>
<td>888</td>
</tr>
<tr>
<td>package PMPREDFS FunctionSpaceAttachPredicates</td>
<td>890</td>
</tr>
<tr>
<td>FunctionSpaceAttachPredicates (PMPREDFS)</td>
<td>891</td>
</tr>
<tr>
<td>package FSCINT FunctionSpaceComplexIntegration</td>
<td>892</td>
</tr>
<tr>
<td>FunctionSpaceComplexIntegration (FSCINT)</td>
<td>893</td>
</tr>
<tr>
<td>package FS2 FunctionSpaceFunctions2</td>
<td>895</td>
</tr>
<tr>
<td>FunctionSpaceFunctions2 (FS2)</td>
<td>897</td>
</tr>
<tr>
<td>package FSINT FunctionSpaceIntegration</td>
<td>898</td>
</tr>
<tr>
<td>FunctionSpaceIntegration (FSINT)</td>
<td>899</td>
</tr>
<tr>
<td>package FSPRMELT FunctionSpacePrimitiveElement</td>
<td>902</td>
</tr>
<tr>
<td>FunctionSpacePrimitiveElement (FSPRMELT)</td>
<td>903</td>
</tr>
<tr>
<td>package FSRED FunctionSpaceReduce</td>
<td>906</td>
</tr>
<tr>
<td>FunctionSpaceReduce (FSRED)</td>
<td>907</td>
</tr>
<tr>
<td>package SUMFS FunctionSpaceSum</td>
<td>909</td>
</tr>
<tr>
<td>FunctionSpaceSum (SUMFS)</td>
<td>910</td>
</tr>
<tr>
<td>package FS2EXPXP FunctionSpaceToExponentialExpansion</td>
<td>912</td>
</tr>
<tr>
<td>Package Name</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>InnerCommonDenominator (ICDEN)</td>
<td>1220</td>
</tr>
<tr>
<td>package IMATLIN InnerMatrixLinearAlgebraFunctions</td>
<td>1222</td>
</tr>
<tr>
<td>InnerMatrixLinearAlgebraFunctions (IMATLIN)</td>
<td>1223</td>
</tr>
<tr>
<td>package IMATQF InnerMatrixQuotientFieldFunctions</td>
<td>1228</td>
</tr>
<tr>
<td>InnerMatrixQuotientFieldFunctions (IMATQF)</td>
<td>1229</td>
</tr>
<tr>
<td>package INMODGCD InnerModularGcd</td>
<td>1230</td>
</tr>
<tr>
<td>InnerModularGcd (INMODGCD)</td>
<td>1232</td>
</tr>
<tr>
<td>package INNMFACT InnerMultFact</td>
<td>1237</td>
</tr>
<tr>
<td>InnerMultFact (INNMFACT)</td>
<td>1239</td>
</tr>
<tr>
<td>package INBFF InnerNormalBasisFieldFunctions</td>
<td>1248</td>
</tr>
<tr>
<td>InnerNormalBasisFieldFunctions (INBFF)</td>
<td>1249</td>
</tr>
<tr>
<td>package INEP InnerNumericEigenPackage</td>
<td>1257</td>
</tr>
<tr>
<td>InnerNumericEigenPackage (INEP)</td>
<td>1258</td>
</tr>
<tr>
<td>package INFSP InnerNumericFloatSolvePackage</td>
<td>1262</td>
</tr>
<tr>
<td>InnerNumericFloatSolvePackage (INFSP)</td>
<td>1264</td>
</tr>
<tr>
<td>package INPSIGN InnerPolySign</td>
<td>1268</td>
</tr>
<tr>
<td>InnerPolySign (INPSIGN)</td>
<td>1269</td>
</tr>
<tr>
<td>package ISUMP InnerPolySum</td>
<td>1270</td>
</tr>
<tr>
<td>InnerPolySum (ISUMP)</td>
<td>1272</td>
</tr>
<tr>
<td>package ITRIGMNP InnerTrigonometricManipulations</td>
<td>1273</td>
</tr>
<tr>
<td>InnerTrigonometricManipulations (ITRIGMNP)</td>
<td>1275</td>
</tr>
<tr>
<td>package INFORM1 InputFormFunctions1</td>
<td>1279</td>
</tr>
<tr>
<td>InputFormFunctions1 (INFORM1)</td>
<td>1280</td>
</tr>
<tr>
<td>package INTERGB InterfaceGroebnerPackage</td>
<td>1281</td>
</tr>
<tr>
<td>InterfaceGroebnerPackage (INTERGB)</td>
<td>1282</td>
</tr>
<tr>
<td>IntegerBits (INTBIT)</td>
<td>1285</td>
</tr>
<tr>
<td>package COMBINAT IntegerCombinatoricFunctions</td>
<td>1286</td>
</tr>
<tr>
<td>IntegerCombinatoricFunctions (COMBINAT)</td>
<td>1289</td>
</tr>
<tr>
<td>package INTFACT IntegerFactorizationPackage</td>
<td>1292</td>
</tr>
<tr>
<td>IntegerFactorizationPackage (INTFACT)</td>
<td>1293</td>
</tr>
<tr>
<td>squareFree</td>
<td>1294</td>
</tr>
<tr>
<td>PollardSmallFactor</td>
<td>1295</td>
</tr>
<tr>
<td>BasicSieve</td>
<td>1297</td>
</tr>
<tr>
<td>BasicMethod</td>
<td>1298</td>
</tr>
<tr>
<td>factor</td>
<td>1298</td>
</tr>
<tr>
<td>package ZLINDEP IntegerLinearDependence</td>
<td>1300</td>
</tr>
<tr>
<td>IntegerLinearDependence (ZLINDEP)</td>
<td>1304</td>
</tr>
<tr>
<td>package INTHEORY IntegerNumberTheoryFunctions</td>
<td>1305</td>
</tr>
<tr>
<td>IntegerNumberTheoryFunctions (INTHEORY)</td>
<td>1320</td>
</tr>
<tr>
<td>package PRIMES IntegerPrimesPackage</td>
<td>1325</td>
</tr>
<tr>
<td>IntegerPrimesPackage (PRIMES)</td>
<td>1326</td>
</tr>
<tr>
<td>smallPrimes</td>
<td>1327</td>
</tr>
<tr>
<td>primes</td>
<td>1332</td>
</tr>
<tr>
<td>rabinProvesCompositeSmall</td>
<td>1333</td>
</tr>
<tr>
<td>rabinProvesComposite</td>
<td>1333</td>
</tr>
<tr>
<td>prime?</td>
<td>1334</td>
</tr>
<tr>
<td>Content</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>nextPrime</td>
<td>1335</td>
</tr>
<tr>
<td>prevPrime</td>
<td>1335</td>
</tr>
<tr>
<td>package INTRET IntegerRetractions</td>
<td>1336</td>
</tr>
<tr>
<td>IntegerRetractions (INTRET)</td>
<td>1337</td>
</tr>
<tr>
<td>package IROOT IntegerRoots</td>
<td>1338</td>
</tr>
<tr>
<td>IntegerRoots (IROOT)</td>
<td>1339</td>
</tr>
<tr>
<td>perfectSquare?</td>
<td>1340</td>
</tr>
<tr>
<td>perfectNthPower?</td>
<td>1341</td>
</tr>
<tr>
<td>perfectNthRoot</td>
<td>1341</td>
</tr>
<tr>
<td>approxNthRoot</td>
<td>1341</td>
</tr>
<tr>
<td>perfectNthRoot</td>
<td>1342</td>
</tr>
<tr>
<td>perfectSqrt</td>
<td>1342</td>
</tr>
<tr>
<td>approxSqrt</td>
<td>1343</td>
</tr>
<tr>
<td>package INTSLPE IntegerSolveLinearPolynomialEquation</td>
<td>1343</td>
</tr>
<tr>
<td>IntegerSolveLinearPolynomialEquation (INTSLPE)</td>
<td>1344</td>
</tr>
<tr>
<td>package IBATOOL IntegralBasisTools</td>
<td>1346</td>
</tr>
<tr>
<td>IntegralBasisTools (IBATOOL)</td>
<td>1347</td>
</tr>
<tr>
<td>package IBPTOOLS IntegralBasisPolynomialTools</td>
<td>1350</td>
</tr>
<tr>
<td>IntegralBasisPolynomialTools (IBPTOOLS)</td>
<td>1352</td>
</tr>
<tr>
<td>package IR2 IntegrationResultFunctions2</td>
<td>1354</td>
</tr>
<tr>
<td>IntegrationResultFunctions2 (IR2)</td>
<td>1355</td>
</tr>
<tr>
<td>package IRRF2F IntegrationResultRFToFunction</td>
<td>1357</td>
</tr>
<tr>
<td>IntegrationResultRFToFunction (IRRF2F)</td>
<td>1358</td>
</tr>
<tr>
<td>package IR2F IntegrationResultToFunction</td>
<td>1360</td>
</tr>
<tr>
<td>IntegrationResultToFunction (IR2F)</td>
<td>1361</td>
</tr>
<tr>
<td>package INTTOOLS IntegrationTools</td>
<td>1366</td>
</tr>
<tr>
<td>IntegrationTools (INTTOOLS)</td>
<td>1367</td>
</tr>
<tr>
<td>package IPRNTPK InternalPrintPackage</td>
<td>1370</td>
</tr>
<tr>
<td>InternalPrintPackage (IPRNTPK)</td>
<td>1371</td>
</tr>
<tr>
<td>package IRURPK InternalRationalUnivariateRepresentationPackage</td>
<td>1372</td>
</tr>
<tr>
<td>InternalRationalUnivariateRepresentationPackage (IRURPK)</td>
<td>1374</td>
</tr>
<tr>
<td>package INTFRSP InterpolateFormsPackage</td>
<td>1378</td>
</tr>
<tr>
<td>InterpolateFormsPackage (INTFRSP)</td>
<td>1379</td>
</tr>
<tr>
<td>package INTDIVP IntersectionDivisorPackage</td>
<td>1386</td>
</tr>
<tr>
<td>IntersectionDivisorPackage (INTDIVP)</td>
<td>1387</td>
</tr>
<tr>
<td>package IRREDFFX IrredPolyOverFiniteField</td>
<td>1389</td>
</tr>
<tr>
<td>IrredPolyOverFiniteField (IRREDFFX)</td>
<td>1390</td>
</tr>
<tr>
<td>package IRSN IrrRepSymNatPackage</td>
<td>1392</td>
</tr>
<tr>
<td>IrrRepSymNatPackage (IRSN)</td>
<td>1394</td>
</tr>
<tr>
<td>package INVCLAPLA InverseLaplaceTransform</td>
<td>1400</td>
</tr>
<tr>
<td>InverseLaplaceTransform (INVCLAPLA)</td>
<td>1401</td>
</tr>
</tbody>
</table>
### 12 Chapter K

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernelfunctions2 (KERNEL2)</td>
<td>1407</td>
</tr>
<tr>
<td>kovacic (KOVACIC)</td>
<td>1409</td>
</tr>
</tbody>
</table>

### 13 Chapter L

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>laplaceTransform (LAPLACE)</td>
<td>1413</td>
</tr>
<tr>
<td>lazarSetSolvingPackage (LAZM3PK)</td>
<td>1419</td>
</tr>
<tr>
<td>leadingCoefDetermination (LEADCDET)</td>
<td>1443</td>
</tr>
<tr>
<td>lextriangularPackage (LEXTRIPK)</td>
<td>1446</td>
</tr>
<tr>
<td>linearDependence (LINDEP)</td>
<td>1522</td>
</tr>
<tr>
<td>linearOrdinaryDifferentialOperatorFactorizer</td>
<td>1525</td>
</tr>
<tr>
<td>linearOrdinaryDifferentialOperatorsOps</td>
<td>1529</td>
</tr>
<tr>
<td>linearPolynomialEquationByFractions</td>
<td>1533</td>
</tr>
<tr>
<td>linearSystemFromPowerSeriesPackage</td>
<td>1535</td>
</tr>
<tr>
<td>linearSystemMatrixPackage</td>
<td>1538</td>
</tr>
<tr>
<td>linearSystemMatrixPackage1</td>
<td>1542</td>
</tr>
<tr>
<td>linearSystemPolynomialPackage</td>
<td>1545</td>
</tr>
<tr>
<td>linGroebnerPackage</td>
<td>1547</td>
</tr>
<tr>
<td>linearOpPack</td>
<td>1555</td>
</tr>
<tr>
<td>liouvillianFunction (LF)</td>
<td>1559</td>
</tr>
<tr>
<td>listFunctions2 (LIST2)</td>
<td>1565</td>
</tr>
<tr>
<td>listFunctions3 (LIST3)</td>
<td>1567</td>
</tr>
<tr>
<td>listToMap</td>
<td>1570</td>
</tr>
<tr>
<td>localParametrizationOfSimplePointPackage</td>
<td>1573</td>
</tr>
</tbody>
</table>
## Contents

### 14 Chapter M

<table>
<thead>
<tr>
<th>Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>package MKBCFUNC MakeBinaryCompiledFunction</td>
<td>1581</td>
</tr>
<tr>
<td>MakeBinaryCompiledFunction (MKBCFUNC)</td>
<td>1582</td>
</tr>
<tr>
<td>package MKFLCFN MakeFloatCompiledFunction</td>
<td>1583</td>
</tr>
<tr>
<td>MakeFloatCompiledFunction (MKFLCFN)</td>
<td>1585</td>
</tr>
<tr>
<td>package MKFUNC MakeFunction</td>
<td>1588</td>
</tr>
<tr>
<td>MakeFunction (MKFUNC)</td>
<td>1593</td>
</tr>
<tr>
<td>package MKRECORD MakeRecord</td>
<td>1594</td>
</tr>
<tr>
<td>MakeRecord (MKRECORD)</td>
<td>1595</td>
</tr>
<tr>
<td>package MKUCFUNC MakeUnaryCompiledFunction</td>
<td>1596</td>
</tr>
<tr>
<td>MakeUnaryCompiledFunction (MKUCFUNC)</td>
<td>1597</td>
</tr>
<tr>
<td>package MAPHACK1 MappingPackageInternalHacks1</td>
<td>1598</td>
</tr>
<tr>
<td>MappingPackageInternalHacks1 (MAPHACK1)</td>
<td>1600</td>
</tr>
<tr>
<td>package MAPHACK2 MappingPackageInternalHacks2</td>
<td>1601</td>
</tr>
<tr>
<td>MappingPackageInternalHacks2 (MAPHACK2)</td>
<td>1602</td>
</tr>
<tr>
<td>package MAPHACK3 MappingPackageInternalHacks3</td>
<td>1603</td>
</tr>
<tr>
<td>MappingPackageInternalHacks3 (MAPHACK3)</td>
<td>1604</td>
</tr>
<tr>
<td>package MAPPKG1 MappingPackage1</td>
<td>1605</td>
</tr>
<tr>
<td>MappingPackage1 (MAPPKG1)</td>
<td>1614</td>
</tr>
<tr>
<td>package MAPPKG2 MappingPackage2</td>
<td>1616</td>
</tr>
<tr>
<td>MappingPackage2 (MAPPKG2)</td>
<td>1625</td>
</tr>
<tr>
<td>package MAPPKG3 MappingPackage3</td>
<td>1626</td>
</tr>
<tr>
<td>MappingPackage3 (MAPPKG3)</td>
<td>1636</td>
</tr>
<tr>
<td>package MAPPKG4 MappingPackage4</td>
<td>1638</td>
</tr>
<tr>
<td>MappingPackage4 (MAPPKG4)</td>
<td>1643</td>
</tr>
<tr>
<td>package MATCAT2 MatrixCategoryFunctions2</td>
<td>1645</td>
</tr>
<tr>
<td>MatrixCategoryFunctions2 (MATCAT2)</td>
<td>1646</td>
</tr>
<tr>
<td>package MCDEN MatrixCommonDenominator</td>
<td>1648</td>
</tr>
<tr>
<td>MatrixCommonDenominator (MCDEN)</td>
<td>1649</td>
</tr>
<tr>
<td>package MATLIN MatrixLinearAlgebraFunctions</td>
<td>1650</td>
</tr>
<tr>
<td>MatrixLinearAlgebraFunctions (MATLIN)</td>
<td>1652</td>
</tr>
<tr>
<td>package MAMA MatrixManipulation</td>
<td>1659</td>
</tr>
<tr>
<td>MatrixManipulation (MAMA)</td>
<td>1703</td>
</tr>
<tr>
<td>package MTHING MergeThing</td>
<td>1711</td>
</tr>
<tr>
<td>MergeThing (MTHING)</td>
<td>1712</td>
</tr>
<tr>
<td>package MESH MeshCreationRoutinesForThreeDimensions</td>
<td>1713</td>
</tr>
<tr>
<td>MeshCreationRoutinesForThreeDimensions (MESH)</td>
<td>1715</td>
</tr>
<tr>
<td>package MDDFACT ModularDistinctDegreeFactorizer</td>
<td>1718</td>
</tr>
<tr>
<td>ModularDistinctDegreeFactorizer (MDDFACT)</td>
<td>1719</td>
</tr>
<tr>
<td>package MHROWRED ModularHermitianRowReduction</td>
<td>1724</td>
</tr>
<tr>
<td>ModularHermitianRowReduction (MHROWRED)</td>
<td>1725</td>
</tr>
<tr>
<td>package MRF2 MonoidRingFunctions2</td>
<td>1731</td>
</tr>
<tr>
<td>MonoidRingFunctions2 (MRF2)</td>
<td>1732</td>
</tr>
<tr>
<td>package MONOTOOL MonomialExtensionTools</td>
<td>1733</td>
</tr>
<tr>
<td>MonomialExtensionTools (MONOTOOL)</td>
<td>1734</td>
</tr>
<tr>
<td>package MSYSCMD MoreSystemCommands</td>
<td>1736</td>
</tr>
</tbody>
</table>
CONTENTS

package NAGC02 NagPolynomialRootsPackage .............................................. 2796
NagPolynomialRootsPackage (NAGC02) ...................................................... 2812
package NAGC05 NagRootFindingPackage .................................................. 2814
NagRootFindingPackage (NAGC05) .......................................................... 2836
package NAGC06 NagSeriesSummationPackage ........................................... 2839
NagSeriesSummationPackage (NAGC06) ..................................................... 2901
package NAGS NagSpecialFunctionsPackage ............................................. 2907
NagSpecialFunctionsPackage (NAGS) ....................................................... 3058
package NSUP2 NewSparseUnivariatePolynomialFunctions2 ............................. 3074
NewSparseUnivariatePolynomialFunctions2 (NSUP2) .................................. 3075
package NEWTON NewtonInterpolation .................................................... 3076
NewtonInterpolation (NEWTON) ............................................................ 3077
package NPOLYGON NewtonPolygon ......................................................... 3079
NewtonPolygon (NPOLYGON) ................................................................. 3080
package NCODIV NonCommutativeOperatorDivision ..................................... 3084
NonCommutativeOperatorDivision (NCODIV) ............................................. 3086
package NONE1 NoneFunctions1 ............................................................... 3088
NoneFunctions1 (NONE1) ............................................................................ 3089
package NODE1 NonLinearFirstOrderODESolver ......................................... 3090
NonLinearFirstOrderODESolver (NODE1) .................................................. 3091
package NLINSOL NonLinearSolvePackage ............................................... 3095
NonLinearSolvePackage (NLINSOL) .......................................................... 3096
package NORMPK NormalizationPackage ................................................... 3098
NormalizationPackage (NORMPK) ............................................................. 3100
package NORMMA NormInMonogenicAlgebra .............................................. 3104
NormInMonogenicAlgebra (NORMMA) ....................................................... 3105
package NORMRETR NormRetractPackage ................................................ 3106
NormRetractPackage (NORMRETR) ........................................................... 3108
package NPCOEF NPCoef ......................................................................... 3109
NPCoef (NPCOEF) ...................................................................................... 3111
package NFINTBAS NumberFieldIntegralBasis ............................................ 3114
NumberFieldIntegralBasis (NFINTBAS) ..................................................... 3116
package NUMFMT NumberFormats ............................................................. 3121
NumberFormats (NUMFMT) ......................................................................... 3122
package NTPOLFN NumberTheoreticPolynomialFunctions .............................. 3126
NumberTheoreticPolynomialFunctions (NTPOLFN) ..................................... 3127
package NUMERIC Numeric ..................................................................... 3129
Numeric (NUMERIC) ................................................................................... 3131
package NUMODE NumericalOrdinaryDifferentialEquations .......................... 3140
NumericalOrdinaryDifferentialEquations (NUMODE) ................................ 3144
package NUMQUAD NumericalQuadrature ............................................... 3151
NumericalQuadrature (NUMQUAD) ............................................................ 3154
package NCEP NumericComplexEigenPackage ......................................... 3165
NumericComplexEigenPackage (NCEP) ...................................................... 3166
package NCNTFRAC NumericContinuedFraction ........................................ 3168
NumericContinuedFraction (NCNTFRAC) .................................................. 3170
package NREP NumericRealEigenPackage ........................................... 3171
   NumericRealEigenPackage (NREP) ............................................. 3172
package NUMTUBE NumericTubePlot .............................................. 3174
   NumericTubePlot (NUMTUBE) .................................................. 3175

16 Chapter O ................................................................................. 3179
   package OCTCT2 OctonionCategoryFunctions2 ................................... 3179
      OctonionCategoryFunctions2 (OCTCT2) ...................................... 3180
   package ODEINT ODEIntegration .................................................. 3181
      ODEIntegration (ODEINT) ...................................................... 3182
   package ODETOOLS ODETools ...................................................... 3185
      ODETools (ODETOOLS) .......................................................... 3186
   package ARRAY12 OneDimensionalArrayFunctions2 ......................... 3188
      OneDimensionalArrayFunctions2 (ARRAY12) ............................... 3189
   package ONECOMP2 OnePointCompletionFunctions2 ....................... 3190
      OnePointCompletionFunctions2 (ONECOMP2) ............................. 3191
   package OMPKG OpenMathPackage .............................................. 3193
      OpenMathPackage (OMPKG) .................................................... 3194
   package OMSERVER OpenMathServerPackage .................................. 3196
      OpenMathServerPackage (OMSERVER) ....................................... 3197
   package OPQUERY OperationsQuery .............................................. 3199
      OperationsQuery (OPQUERY) .................................................. 3200
   package ORDCOMP2 OrderedCompletionFunctions2 ....................... 3201
      OrderedCompletionFunctions2 (ORDCOMP2) ............................... 3202
   package ORDFUNS OrderingFunctions .......................................... 3203
      OrderingFunctions (ORDFUNS) ............................................... 3204
   package ORTHPOL OrthogonalPolynomialFunctions .......................... 3206
      OrthogonalPolynomialFunctions (ORTHPOL) ............................. 3207
   package OUT OutputPackage ...................................................... 3210
      OutputPackage (OUT) .......................................................... 3211

17 Chapter P ................................................................................. 3213
   package PAFF PackageForAlgebraicFunctionField ............................ 3213
      PackageForAlgebraicFunctionField (PAFF) ................................ 3215
   package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField ....... 3221
      PackageForAlgebraicFunctionFieldOverFiniteField (PAFFFF) ......... 3223
   package PFORP PackageForPoly .................................................... 3231
      PackageForPoly (PFORP) ........................................................ 3232
   package PADEPAC PadeApproximantPackage .................................. 3239
      PadeApproximantPackage (PADEPAC) ....................................... 3240
   package PADE PadeApproximants .................................................. 3241
      PadeApproximants (PADE) ....................................................... 3242
   package PWFFINTB PAdicWildFunctionFieldIntegralBasis .................. 3246
      PAdicWildFunctionFieldIntegralBasis (PWFFINTB) ....................... 3247
   package YSTREAM ParadoxicalCombinatorsForStreams ..................... 3252
      ParadoxicalCombinatorsForStreams (YSTREAM) .......................... 3253
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PermutationGroupExamples</td>
<td>Package containing examples of permutation groups</td>
<td>3344</td>
</tr>
<tr>
<td>PiCoercions</td>
<td>Package containing coercion functions</td>
<td>3354</td>
</tr>
<tr>
<td>PlotFunctions1</td>
<td>Package containing plot functions</td>
<td>3356</td>
</tr>
<tr>
<td>PlotTools</td>
<td>Package containing plot tools</td>
<td>3358</td>
</tr>
<tr>
<td>ProjectiveAlgebraicSetPackage</td>
<td>Package containing projective algebraic set tools</td>
<td>3361</td>
</tr>
<tr>
<td>PointFunctions2</td>
<td>Package containing point functions</td>
<td>3366</td>
</tr>
<tr>
<td>PointPackage</td>
<td>Package containing point package</td>
<td>3368</td>
</tr>
<tr>
<td>PointsOfFiniteOrder</td>
<td>Package containing points of finite order</td>
<td>3372</td>
</tr>
<tr>
<td>PointsOfFiniteOrderRational</td>
<td>Package containing rational points of finite order</td>
<td>3379</td>
</tr>
<tr>
<td>PointsOfFiniteOrderTools</td>
<td>Package containing point tools for finite order</td>
<td>3382</td>
</tr>
<tr>
<td>PolynomialPackageForCurve</td>
<td>Package containing polynomial functions for curve</td>
<td>3385</td>
</tr>
<tr>
<td>PolynomialAN2Expression</td>
<td>Polynomial category lifting</td>
<td>3395</td>
</tr>
<tr>
<td>PolynomialAN2Expression</td>
<td>Polynomial category quotient functions</td>
<td>3399</td>
</tr>
<tr>
<td>PolynomialComposition</td>
<td>Package containing polynomial composition</td>
<td>3404</td>
</tr>
<tr>
<td>PolynomialDecomposition</td>
<td>Package containing polynomial decomposition</td>
<td>3406</td>
</tr>
<tr>
<td>PolynomialDecomposition</td>
<td>Package containing polynomial factorization by recursion</td>
<td>3409</td>
</tr>
<tr>
<td>PolynomialDecomposition</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3416</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursion</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3418</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursionUnivariate</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3423</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursionUnivariate</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3424</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursionUnivariate</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3425</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursionUnivariate</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3426</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursionUnivariate</td>
<td>Package containing polynomial factorization by recursion univariate</td>
<td>3427</td>
</tr>
</tbody>
</table>
### CONTENTS

- **package PINTERPA** PolynomialInterpolationAlgorithms ........................................ 3438
  PolynomialInterpolationAlgorithms (PINTERPA) .................................................. 3440
- **package PNTHEORY** PolynomialNumberTheoryFunctions ...................................... 3441
  PolynomialNumberTheoryFunctions (PNTHEORY) .................................................. 3442
- **package POLYROOT** PolynomialRoots ................................................................. 3447
  PolynomialRoots (POLYROOT) ............................................................................. 3448
- **package PSETPK** PolynomialSetUtilitiesPackage .............................................. 3451
  PolynomialSetUtilitiesPackage (PSETPK) ........................................................... 3453
- **package SOLVEFOR** PolynomialSolveByFormulas .............................................. 3470
  PolynomialSolveByFormulas (SOLVEFOR) ........................................................... 3471
- **package PSQFR** PolynomialSquareFree ............................................................. 3477
  PolynomialSquareFree (PSQFR) ........................................................................ 3478
- **package POLY2UP** PolynomialToUnivariatePolynomial .................................... 3481
  PolynomialToUnivariatePolynomial (POLY2UP) .................................................. 3482
- **package LIMITPS** PowerSeriesLimitPackage .................................................... 3483
  PowerSeriesLimitPackage (LIMITPS) ................................................................. 3484
- **package PREAASSOC** PrecomputedAssociatedEquations ................................... 3495
  PrecomputedAssociatedEquations (PREAASSOC) ............................................... 3497
- **package PRIMARR2** PrimitiveArrayFunctions2 .................................................. 3499
  PrimitiveArrayFunctions2 (PRIMARR2) ............................................................. 3500
- **package PRIMELT** PrimitiveElement .................................................................. 3502
  PrimitiveElement (PRIMELT) ............................................................................ 3503
- **package ODEPRIM** PrimitiveRatDE ................................................................. 3506
  PrimitiveRatDE (ODEPRIM) .............................................................................. 3507
- **package ODEPRRIC** PrimitiveRatRicDE .......................................................... 3511
  PrimitiveRatRicDE (ODEPRRIC) ..................................................................... 3512
- **package PRINT** PrintPackage ............................................................................ 3518
  PrintPackage (PRINT) .................................................................................... 3519
- **package PSEUDLIN** PseudoLinearNormalForm ................................................. 3520
  PseudoLinearNormalForm (PSEUDLIN) .............................................................. 3521
- **package PRS** PseudoRemainderSequence .......................................................... 3524
  PseudoRemainderSequence (PRS) ................................................................... 3526
- **package INTPAF** PureAlgebraicIntegration ...................................................... 3546
  PureAlgebraicIntegration (INTPAF) ................................................................... 3547
- **package ODEPAL** PureAlgebraicLODE ............................................................. 3555
  PureAlgebraicLODE (ODEPAL) .......................................................................... 3556
- **package PUSHVAR** PushVariables .................................................................... 3557
  PushVariables (PUSHVAR) ............................................................................ 3559

#### 18 Chapter Q

- **package QALGSET2** QuasiAlgebraicSet2 ......................................................... 3561
  QuasiAlgebraicSet2 (QALGSET2) ..................................................................... 3563
- **package QCMPACK** QuasiComponentPackage .................................................. 3565
  QuasiComponentPackage (QCMPACK) .............................................................. 3567
- **package QFCAT2** QuotientFieldCategoryFunctions2 ....................................... 3575
  QuotientFieldCategoryFunctions2 (QFCAT2) .................................................... 3576
<table>
<thead>
<tr>
<th>Package Name</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>QuaternionCategoryFunctions2 QUATCT2</td>
<td>Section R</td>
<td>3577</td>
</tr>
<tr>
<td>QuaternionCategoryFunctions2 (QUATCT2)</td>
<td></td>
<td>3579</td>
</tr>
</tbody>
</table>

**19 Chapter R**

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadicalEigenPackage REP</td>
<td>Section R</td>
<td>3581</td>
</tr>
<tr>
<td>RadicalEigenPackage (REP)</td>
<td></td>
<td>3582</td>
</tr>
<tr>
<td>RadicalSolvePackage SOLVERAD</td>
<td>Section R</td>
<td>3586</td>
</tr>
<tr>
<td>RadicalSolvePackage (SOLVERAD)</td>
<td></td>
<td>3600</td>
</tr>
<tr>
<td>RadixUtilities RADUTIL</td>
<td>Section R</td>
<td>3607</td>
</tr>
<tr>
<td>RadixUtilities (RADUTIL)</td>
<td></td>
<td>3608</td>
</tr>
<tr>
<td>RandomDistributions RDIST</td>
<td>Section R</td>
<td>3609</td>
</tr>
<tr>
<td>RandomDistributions (RDIST)</td>
<td></td>
<td>3610</td>
</tr>
<tr>
<td>RandomFloatDistributions RFDIST</td>
<td>Section R</td>
<td>3611</td>
</tr>
<tr>
<td>RandomFloatDistributions (RFDIST)</td>
<td></td>
<td>3613</td>
</tr>
<tr>
<td>RandomIntegerDistributions RIDIST</td>
<td>Section R</td>
<td>3615</td>
</tr>
<tr>
<td>RandomIntegerDistributions (RIDIST)</td>
<td></td>
<td>3616</td>
</tr>
<tr>
<td>RandomNumberSource RANDSRC</td>
<td>Section R</td>
<td>3618</td>
</tr>
<tr>
<td>RandomNumberSource (RANDSRC)</td>
<td></td>
<td>3619</td>
</tr>
<tr>
<td>RationalFactorize RATFACT</td>
<td>Section R</td>
<td>3621</td>
</tr>
<tr>
<td>RationalFactorize (RATFACT)</td>
<td></td>
<td>3622</td>
</tr>
<tr>
<td>RationalFunction RF</td>
<td>Section R</td>
<td>3623</td>
</tr>
<tr>
<td>RationalFunction (RF)</td>
<td></td>
<td>3625</td>
</tr>
<tr>
<td>RationalFunctionDefiniteIntegration DEFINTRF</td>
<td>Section R</td>
<td>3627</td>
</tr>
<tr>
<td>RationalFunctionDefiniteIntegration (DEFINTRF)</td>
<td></td>
<td>3628</td>
</tr>
<tr>
<td>RationalFunctionFactor RFFACT</td>
<td>Section R</td>
<td>3630</td>
</tr>
<tr>
<td>RationalFunctionFactor (RFFACT)</td>
<td></td>
<td>3631</td>
</tr>
<tr>
<td>RationalFunctionFactorizer RFFACTOR</td>
<td>Section R</td>
<td>3632</td>
</tr>
<tr>
<td>RationalFunctionFactorizer (RFFACTOR)</td>
<td></td>
<td>3634</td>
</tr>
<tr>
<td>RationalFunctionIntegration INTRF</td>
<td>Section R</td>
<td>3635</td>
</tr>
<tr>
<td>RationalFunctionIntegration (INTRF)</td>
<td></td>
<td>3636</td>
</tr>
<tr>
<td>RationalFunctionLimitPackage LIMITRF</td>
<td>Section R</td>
<td>3638</td>
</tr>
<tr>
<td>RationalFunctionLimitPackage (LIMITRF)</td>
<td></td>
<td>3639</td>
</tr>
<tr>
<td>RationalFunctionSign SIGNRF</td>
<td>Section R</td>
<td>3642</td>
</tr>
<tr>
<td>RationalFunctionSign (SIGNRF)</td>
<td></td>
<td>3644</td>
</tr>
<tr>
<td>RationalFunctionSum SUMRF</td>
<td>Section R</td>
<td>3646</td>
</tr>
<tr>
<td>RationalFunctionSum (SUMRF)</td>
<td></td>
<td>3652</td>
</tr>
<tr>
<td>RationalIntegration INTRAT</td>
<td>Section R</td>
<td>3654</td>
</tr>
<tr>
<td>RationalIntegration (INTRAT)</td>
<td></td>
<td>3655</td>
</tr>
<tr>
<td>RationalInterpolation RINTERP</td>
<td>Section R</td>
<td>3657</td>
</tr>
<tr>
<td>RationalInterpolation (RINTERP)</td>
<td></td>
<td>3657</td>
</tr>
<tr>
<td>RationalLODE ODERAT</td>
<td>Section R</td>
<td>3661</td>
</tr>
<tr>
<td>RationalLODE (ODERAT)</td>
<td></td>
<td>3662</td>
</tr>
<tr>
<td>RationalRetractions RATRET</td>
<td>Section R</td>
<td>3667</td>
</tr>
<tr>
<td>RationalRetractions (RATRET)</td>
<td></td>
<td>3669</td>
</tr>
<tr>
<td>Package Name</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>ODERTRIC RationalRicDE</td>
<td></td>
<td>3670</td>
</tr>
<tr>
<td>RationalRicDE (ODERTRIC)</td>
<td></td>
<td>3671</td>
</tr>
<tr>
<td>RURPK RationalUnivariateRepresentationPackage</td>
<td></td>
<td>3677</td>
</tr>
<tr>
<td>RationalUnivariateRepresentationPackage (RURPK)</td>
<td></td>
<td>3679</td>
</tr>
<tr>
<td>POLUTIL RealPolynomialUtilitiesPackage</td>
<td></td>
<td>3682</td>
</tr>
<tr>
<td>RealPolynomialUtilitiesPackage (POLUTIL)</td>
<td></td>
<td>3684</td>
</tr>
<tr>
<td>REALSOLV RealSolvePackage</td>
<td></td>
<td>3686</td>
</tr>
<tr>
<td>RealSolvePackage (REALSOLV)</td>
<td></td>
<td>3690</td>
</tr>
<tr>
<td>REAL0 RealZeroPackage</td>
<td></td>
<td>3692</td>
</tr>
<tr>
<td>RealZeroPackage (REAL0)</td>
<td></td>
<td>3693</td>
</tr>
<tr>
<td>REAL0Q RealZeroPackageQ</td>
<td></td>
<td>3699</td>
</tr>
<tr>
<td>RealZeroPackageQ (REAL0Q)</td>
<td></td>
<td>3701</td>
</tr>
<tr>
<td>RMCAT2 RectangularMatrixCategoryFunctions2</td>
<td></td>
<td>3703</td>
</tr>
<tr>
<td>RectangularMatrixCategoryFunctions2 (RMCAT2)</td>
<td></td>
<td>3704</td>
</tr>
<tr>
<td>RECOP RecurrenceOperator</td>
<td></td>
<td>3705</td>
</tr>
<tr>
<td>RecurrenceOperator (RECOP)</td>
<td></td>
<td>3707</td>
</tr>
<tr>
<td>Defining new operators</td>
<td></td>
<td>3708</td>
</tr>
<tr>
<td>Recurrences</td>
<td></td>
<td>3710</td>
</tr>
<tr>
<td>Functional Equations</td>
<td></td>
<td>3714</td>
</tr>
<tr>
<td>RDIV ReducedDivisor</td>
<td></td>
<td>3718</td>
</tr>
<tr>
<td>ReducedDivisor (RDIV)</td>
<td></td>
<td>3719</td>
</tr>
<tr>
<td>ODERED ReduceLODE</td>
<td></td>
<td>3720</td>
</tr>
<tr>
<td>ReduceLODE (ODERED)</td>
<td></td>
<td>3722</td>
</tr>
<tr>
<td>REDORDER ReductionOfOrder</td>
<td></td>
<td>3723</td>
</tr>
<tr>
<td>ReductionOfOrder (REDORDER)</td>
<td></td>
<td>3725</td>
</tr>
<tr>
<td>RSDCMPK RegularSetDecompositionPackage</td>
<td></td>
<td>3727</td>
</tr>
<tr>
<td>RegularSetDecompositionPackage (RSDCMPK)</td>
<td></td>
<td>3728</td>
</tr>
<tr>
<td>RSETGCD RegularTriangularSetGcdPackage</td>
<td></td>
<td>3734</td>
</tr>
<tr>
<td>RegularTriangularSetGcdPackage (RSETGCD)</td>
<td></td>
<td>3736</td>
</tr>
<tr>
<td>REPDB RepeatedDoubling</td>
<td></td>
<td>3743</td>
</tr>
<tr>
<td>RepeatedDoubling (REPDB)</td>
<td></td>
<td>3744</td>
</tr>
<tr>
<td>REPSQ RepeatedSquaring</td>
<td></td>
<td>3746</td>
</tr>
<tr>
<td>RepeatedSquaring (REPSQ)</td>
<td></td>
<td>3747</td>
</tr>
<tr>
<td>REP1 RepresentationPackage1</td>
<td></td>
<td>3748</td>
</tr>
<tr>
<td>RepresentationPackage1 (REP1)</td>
<td></td>
<td>3750</td>
</tr>
<tr>
<td>REP2 RepresentationPackage2</td>
<td></td>
<td>3757</td>
</tr>
<tr>
<td>RepresentationPackage2 (REP2)</td>
<td></td>
<td>3758</td>
</tr>
<tr>
<td>RESLATC ResolveLatticeCompletion</td>
<td></td>
<td>3774</td>
</tr>
<tr>
<td>ResolveLatticeCompletion (RESLATC)</td>
<td></td>
<td>3775</td>
</tr>
<tr>
<td>RETSOL RetractSolvePackage</td>
<td></td>
<td>3776</td>
</tr>
<tr>
<td>RetractSolvePackage (RETSOL)</td>
<td></td>
<td>3777</td>
</tr>
<tr>
<td>RFP RootsFindingPackage</td>
<td></td>
<td>3779</td>
</tr>
<tr>
<td>RootsFindingPackage (RFP)</td>
<td></td>
<td>3781</td>
</tr>
</tbody>
</table>
## 20 Chapter S

<table>
<thead>
<tr>
<th>Package</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAEFFC</td>
<td>SAERationalFunctionAlgFactor</td>
</tr>
<tr>
<td>FORMULA1</td>
<td>ScriptFormulaFormat1</td>
</tr>
<tr>
<td>SEGBIND2</td>
<td>SegmentBindingFunctions2</td>
</tr>
<tr>
<td>SEG2</td>
<td>SegmentFunctions2</td>
</tr>
<tr>
<td>SAEFACT</td>
<td>SimpleAlgebraicExtensionAlgFactor</td>
</tr>
<tr>
<td>SIMPAN</td>
<td>SimplifyAlgebraicNumberConvertPackage</td>
</tr>
<tr>
<td>SMITH</td>
<td>SmithNormalForm</td>
</tr>
<tr>
<td>SCACHE</td>
<td>SortedCache</td>
</tr>
<tr>
<td>SORTPAK</td>
<td>SortPackage</td>
</tr>
<tr>
<td>SUP2</td>
<td>SparseUnivariatePolynomialFunctions2</td>
</tr>
<tr>
<td>SPECOUT</td>
<td>SpecialOutputPackage</td>
</tr>
<tr>
<td>SFQCMPK</td>
<td>SquareFreeQuasiComponentPackage</td>
</tr>
<tr>
<td>SRDCMPK</td>
<td>SquareFreeRegularSetDecompositionPackage</td>
</tr>
<tr>
<td>SFRGCD</td>
<td>SquareFreeRegularTriangularSetGcdPackage</td>
</tr>
<tr>
<td>MATSTOR</td>
<td>StorageEfficientMatrixOperations</td>
</tr>
<tr>
<td>STREAM1</td>
<td>StreamFunctions1</td>
</tr>
<tr>
<td>STREAM2</td>
<td>StreamFunctions2</td>
</tr>
<tr>
<td>STREAM3</td>
<td>StreamFunctions3</td>
</tr>
<tr>
<td>STINPROD</td>
<td>StreamInfiniteProduct</td>
</tr>
<tr>
<td>STTAYLOR</td>
<td>StreamTaylorSeriesOperations</td>
</tr>
<tr>
<td>STNSR</td>
<td>StreamTensor</td>
</tr>
<tr>
<td>STTF</td>
<td>StreamTranscendentalFunctions</td>
</tr>
<tr>
<td>STTFNC</td>
<td>StreamTranscendentalFunctionsNonCommutative</td>
</tr>
<tr>
<td>CONTENTS</td>
<td>xxvii</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>StreamTranscendentalFunctionsNonCommutative (STTFNC)</td>
<td>3886</td>
</tr>
<tr>
<td>package SCPKG StructuralConstantsPackage</td>
<td>3891</td>
</tr>
<tr>
<td>StructuralConstantsPackage (SCPKG)</td>
<td>3893</td>
</tr>
<tr>
<td>package SHP SturmHabichtPackage</td>
<td>3896</td>
</tr>
<tr>
<td>SturmHabichtPackage (SHP)</td>
<td>3897</td>
</tr>
<tr>
<td>package SUBRESP SubResultantPackage</td>
<td>3905</td>
</tr>
<tr>
<td>SubResultantPackage (SUBRESP)</td>
<td>3906</td>
</tr>
<tr>
<td>package SUPFRACF SupFractionFactorizer</td>
<td>3909</td>
</tr>
<tr>
<td>SupFractionFactorizer (SUPFRACF)</td>
<td>3910</td>
</tr>
<tr>
<td>package ODESYS SystemODESolver</td>
<td>3912</td>
</tr>
<tr>
<td>SystemODESolver (ODESYS)</td>
<td>3913</td>
</tr>
<tr>
<td>package SYSSOLP SystemSolvePackage</td>
<td>3918</td>
</tr>
<tr>
<td>SystemSolvePackage (SYSSOLP)</td>
<td>3920</td>
</tr>
<tr>
<td>package SGCF SymmetricGroupCombinatoricFunctions</td>
<td>3925</td>
</tr>
<tr>
<td>SymmetricGroupCombinatoricFunctions (SGCF)</td>
<td>3927</td>
</tr>
<tr>
<td>package SYMFUNC SymmetricFunctions</td>
<td>3937</td>
</tr>
<tr>
<td>SymmetricFunctions (SYMFUNC)</td>
<td>3938</td>
</tr>
<tr>
<td>21 Chapter T</td>
<td>3941</td>
</tr>
<tr>
<td>package TABLBUMP TableauxBumpers</td>
<td>3941</td>
</tr>
<tr>
<td>TableauxBumpers (TABLBUMP)</td>
<td>3942</td>
</tr>
<tr>
<td>package TBCMPPK TabulatedComputationPackage</td>
<td>3945</td>
</tr>
<tr>
<td>TabulatedComputationPackage (TBCMPPK)</td>
<td>3947</td>
</tr>
<tr>
<td>package TANEXP TangentExpansions</td>
<td>3950</td>
</tr>
<tr>
<td>TangentExpansions (TANEXP)</td>
<td>3951</td>
</tr>
<tr>
<td>package UTSSOL TaylorSolve</td>
<td>3952</td>
</tr>
<tr>
<td>TaylorSolve (UTSSOL)</td>
<td>3954</td>
</tr>
<tr>
<td>package TEMUTL TemplateUtilities</td>
<td>3957</td>
</tr>
<tr>
<td>TemplateUtilities (TEMUTL)</td>
<td>3958</td>
</tr>
<tr>
<td>package TEX1 TexFormat1</td>
<td>3959</td>
</tr>
<tr>
<td>TexFormat1 (TEX1)</td>
<td>3960</td>
</tr>
<tr>
<td>package TOOLSIGN ToolsForSign</td>
<td>3961</td>
</tr>
<tr>
<td>ToolsForSign (TOOLSIGN)</td>
<td>3962</td>
</tr>
<tr>
<td>package DRAW TopLevelDrawFunctions</td>
<td>3964</td>
</tr>
<tr>
<td>TopLevelDrawFunctions (DRAW)</td>
<td>3965</td>
</tr>
<tr>
<td>package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves</td>
<td>3972</td>
</tr>
<tr>
<td>TopLevelDrawFunctionsForAlgebraicCurves (DRAWCURV)</td>
<td>3974</td>
</tr>
<tr>
<td>package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions</td>
<td>3977</td>
</tr>
<tr>
<td>TopLevelDrawFunctionsForCompiledFunctions (DRAWCFUN)</td>
<td>3978</td>
</tr>
<tr>
<td>package DRAWPT TopLevelDrawFunctionsForPoints</td>
<td>3991</td>
</tr>
<tr>
<td>TopLevelDrawFunctionsForPoints (DRAWPT)</td>
<td>3993</td>
</tr>
<tr>
<td>package TOPSP TopLevelThreeSpace</td>
<td>3995</td>
</tr>
<tr>
<td>TopLevelThreeSpace (TOPSP)</td>
<td>3996</td>
</tr>
<tr>
<td>package INTHERTR TranscendentalHermiteIntegration</td>
<td>3997</td>
</tr>
<tr>
<td>TranscendentalHermiteIntegration (INTHERTR)</td>
<td>3998</td>
</tr>
<tr>
<td>package INTTR TranscendentalIntegration</td>
<td>4000</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>TranscendentalIntegration (INTTR)</td>
<td>4001</td>
</tr>
<tr>
<td>package TRMANIP TranscendentalManipulations</td>
<td>4011</td>
</tr>
<tr>
<td>TranscendentalManipulations (TRMANIP)</td>
<td>4012</td>
</tr>
<tr>
<td>The htrigs function</td>
<td>4020</td>
</tr>
<tr>
<td>package RDETR TranscendentalRischDE</td>
<td>4026</td>
</tr>
<tr>
<td>TranscendentalRischDE (RDETR)</td>
<td>4027</td>
</tr>
<tr>
<td>package RDETRS TranscendentalRischDESystem</td>
<td>4031</td>
</tr>
<tr>
<td>TranscendentalRischDESystem (RDETRS)</td>
<td>4032</td>
</tr>
<tr>
<td>package SOLVETRA TransSolvePackage</td>
<td>4037</td>
</tr>
<tr>
<td>TransSolvePackage (SOLVETRA)</td>
<td>4043</td>
</tr>
<tr>
<td>package SOLVESER TransSolvePackageService</td>
<td>4054</td>
</tr>
<tr>
<td>TransSolvePackageService (SOLVESER)</td>
<td>4055</td>
</tr>
<tr>
<td>package TRIMAT TriangularMatrixOperations</td>
<td>4059</td>
</tr>
<tr>
<td>TriangularMatrixOperations (TRIMAT)</td>
<td>4059</td>
</tr>
<tr>
<td>package TRIGMNIP TrigonometricManipulations</td>
<td>4061</td>
</tr>
<tr>
<td>TrigonometricManipulations (TRIGMNIP)</td>
<td>4062</td>
</tr>
<tr>
<td>package TUBETOOL TubePlotTools</td>
<td>4065</td>
</tr>
<tr>
<td>TubePlotTools (TUBETOOL)</td>
<td>4067</td>
</tr>
<tr>
<td>package CLIP TwoDimensionalPlotClipping</td>
<td>4070</td>
</tr>
<tr>
<td>TwoDimensionalPlotClipping (CLIP)</td>
<td>4071</td>
</tr>
<tr>
<td>package TWOFACT TwoFactorize</td>
<td>4077</td>
</tr>
<tr>
<td>TwoFactorize (TWOFACT)</td>
<td>4078</td>
</tr>
</tbody>
</table>

22 Chapter U 4085

<table>
<thead>
<tr>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>package UNIFACT UnivariateFactorize</td>
</tr>
<tr>
<td>UnivariateFactorize (UNIFACT)</td>
</tr>
<tr>
<td>package UFPS1 UnivariateFormalPowerSeriesFunctions</td>
</tr>
<tr>
<td>UnivariateFormalPowerSeriesFunctions (UFPS1)</td>
</tr>
<tr>
<td>package ULS2 UnivariateLaurentSeriesFunctions2</td>
</tr>
<tr>
<td>UnivariateLaurentSeriesFunctions2 (ULS2)</td>
</tr>
<tr>
<td>package UPOLYC2 UnivariatePolynomialCategoryFunctions2</td>
</tr>
<tr>
<td>UnivariatePolynomialCategoryFunctions2 (UPOLYC2)</td>
</tr>
<tr>
<td>package UPCDEN UnivariatePolynomialCommonDenominator</td>
</tr>
<tr>
<td>UnivariatePolynomialCommonDenominator (UPCDEN)</td>
</tr>
<tr>
<td>package UPDECOMP UnivariatePolynomialDecompositionPackage</td>
</tr>
<tr>
<td>UnivariatePolynomialDecompositionPackage (UPDECOMP)</td>
</tr>
<tr>
<td>package UPDIVP UnivariatePolynomialDivisionPackage</td>
</tr>
<tr>
<td>UnivariatePolynomialDivisionPackage (UPDIVP)</td>
</tr>
<tr>
<td>package UP2 UnivariatePolynomialFunctions2</td>
</tr>
<tr>
<td>UnivariatePolynomialFunctions2 (UP2)</td>
</tr>
<tr>
<td>package UPMP UnivariatePolynomialMultiplicationPackage</td>
</tr>
<tr>
<td>UnivariatePolynomialMultiplicationPackage (UPMP)</td>
</tr>
<tr>
<td>package UPSQFREE UnivariatePolynomialSquareFree</td>
</tr>
<tr>
<td>UnivariatePolynomialSquareFree (UPSQFREE)</td>
</tr>
<tr>
<td>package UPXS2 UnivariatePuiseuxSeriesFunctions2</td>
</tr>
<tr>
<td>UnivariatePuiseuxSeriesFunctions2 (UPXS2)</td>
</tr>
<tr>
<td>CONTENTS</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>package OREPCTO UnivariateSkewPolynomialCategoryOps</td>
</tr>
<tr>
<td>UnivariateSkewPolynomialCategoryOps (OREPCTO)</td>
</tr>
<tr>
<td>package UTS2 UnivariateTaylorSeriesFunctions2</td>
</tr>
<tr>
<td>UnivariateTaylorSeriesFunctions2 (UTS2)</td>
</tr>
<tr>
<td>package UTSODE UnivariateTaylorSeriesODESolver</td>
</tr>
<tr>
<td>UnivariateTaylorSeriesODESolver (UTSODE)</td>
</tr>
<tr>
<td>package UNISEG2 UniversalSegmentFunctions2</td>
</tr>
<tr>
<td>UniversalSegmentFunctions2 (UNISEG2)</td>
</tr>
<tr>
<td>package UDPO UserDefinedPartialOrdering</td>
</tr>
<tr>
<td>UserDefinedPartialOrdering (UDPO)</td>
</tr>
<tr>
<td>package UDVO UserDefinedVariableOrdering</td>
</tr>
<tr>
<td>UserDefinedVariableOrdering (UDVO)</td>
</tr>
<tr>
<td>package UTSODETL UTSodetools</td>
</tr>
<tr>
<td>UTSodetools (UTSODETL)</td>
</tr>
<tr>
<td>package POLYVEC U32VectorPolynomialOperations</td>
</tr>
<tr>
<td>U32VectorPolynomialOperations (POLYVEC)</td>
</tr>
<tr>
<td>23 Chapter V</td>
</tr>
<tr>
<td>package VECTOR2 VectorFunctions2</td>
</tr>
<tr>
<td>VectorFunctions2 (VECTOR2)</td>
</tr>
<tr>
<td>package VIEWDEF ViewDefaultsPackage</td>
</tr>
<tr>
<td>ViewDefaultsPackage (VIEWDEF)</td>
</tr>
<tr>
<td>package VIEW ViewportPackage</td>
</tr>
<tr>
<td>ViewportPackage (VIEW)</td>
</tr>
<tr>
<td>24 Chapter W</td>
</tr>
<tr>
<td>package WEIER WeierstrassPreparation</td>
</tr>
<tr>
<td>WeierstrassPreparation (WEIER)</td>
</tr>
<tr>
<td>package WFFINTBS WildFunctionFieldIntegralBasis</td>
</tr>
<tr>
<td>WildFunctionFieldIntegralBasis (WFFINTBS)</td>
</tr>
<tr>
<td>25 Chapter X</td>
</tr>
<tr>
<td>package XEXPPKG XExponentialPackage</td>
</tr>
<tr>
<td>XExponentialPackage (XEXPPKG)</td>
</tr>
<tr>
<td>26 Chapter Y</td>
</tr>
<tr>
<td>27 Chapter Z</td>
</tr>
<tr>
<td>package ZDSOLVE ZeroDimensionalSolvePackage</td>
</tr>
<tr>
<td>ZeroDimensionalSolvePackage (ZDSOLVE)</td>
</tr>
<tr>
<td>28 Chunk collections</td>
</tr>
<tr>
<td>29 Bibliography</td>
</tr>
<tr>
<td>30 Index</td>
</tr>
</tbody>
</table>
New Foreword

On October 1, 2001 Axiom was withdrawn from the market and ended life as a commercial product. On September 3, 2002 Axiom was released under the Modified BSD license, including this document. On August 27, 2003 Axiom was released as free and open source software available for download from the Free Software Foundation’s website, Savannah.

Work on Axiom has had the generous support of the Center for Algorithms and Interactive Scientific Computation (CAISS) at City College of New York. Special thanks go to Dr. Gilbert Baumslag for his support of the long term goal.

The online version of this documentation is roughly 1000 pages. In order to make printed versions we’ve broken it up into three volumes. The first volume is tutorial in nature. The second volume is for programmers. The third volume is reference material. We’ve also added a fourth volume for developers. All of these changes represent an experiment in print-on-demand delivery of documentation. Time will tell whether the experiment succeeded.

Axiom has been in existence for over thirty years. It is estimated to contain about three hundred man-years of research and has, as of September 3, 2003, 143 people listed in the credits. All of these people have contributed directly or indirectly to making Axiom available. Axiom is being passed to the next generation. I’m looking forward to future milestones. With that in mind I’ve introduced the theme of the “30 year horizon”. We must invent the tools that support the Computational Mathematician working 30 years from now. How will research be done when every bit of mathematical knowledge is online and instantly available? What happens when we scale Axiom by a factor of 100, giving us 1.1 million domains? How will we integrate theory with code? How will we integrate theorems and proofs of the mathematics with space-time complexity proofs and running code? What visualization tools are needed? How do we support the conceptual structures and semantics of mathematics in effective ways? How do we support results from the sciences? How do we teach the next generation to be effective Computational Mathematicians?

The “30 year horizon” is much nearer than it appears.

Tim Daly
CAISS, City College of New York
November 10, 2003 ((iHy))
Chapter 1

Chapter Overview

This book contains the packages in Axiom, in alphabetical order.

Each package has an associated 'dotpic' chunk which only lists the packages, categories, and packages that are in the layer immediately below in the build order. For the full list see the algebra Makefile where this information is maintained.

Each package is preceded by a picture. The picture indicates several things. The colors indicate whether the name refers to a category, package, or package. An ellipse means that the name refers to something in the bootstrap set. Thus,

Category  Domain  Package  CABMON
Chapter 2

Chapter A

package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis

-- AffineAlgebraicSetComputeWithGroebnerBasis.input --

)set break resume
)sys rm -f AffineAlgebraicSetComputeWithGroebnerBasis.output
)spool AffineAlgebraicSetComputeWithGroebnerBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AffineAlgebraicSetComputeWithGroebnerBasis

--R
--R AffineAlgebraicSetComputeWithGroebnerBasis(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: DirectProductCategory(#(symb),NonNegativeInteger),ProjPt: ProjectiveSpaceCategory(K)) is a package constructor
--R Abbreviation for AffineAlgebraicSetComputeWithGroebnerBasis is AFALGGRO
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AFALGGRO
--R
--R-------------------------------- Operations --------------------------------
--R affineAlgSet : List(PolyRing) -> Union(List(ProjPt),"failed",Infinite,Integer)
--R affineRationalPoints : (PolyRing,PositiveInteger) -> List(ProjPt)
--R affineSingularPoints : PolyRing -> Union(List(ProjPt),"failed",Infinite,Integer)
--R
--E 1

)spool
)lisp (bye)
AffineAlgebraicSetComputeWithGroebnerBasis (AFALGGRO)

Exports:

affineAlgSet  affineRationalPoints  affineSingularPoints

--- package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis ---

)abbrev package AFALGGRO AffineAlgebraicSetComputeWithGroebnerBasis
++ Author: Gaetan Hache
++ Date Created: 17 Nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
AffineAlgebraicSetComputeWithGroebnerBasis(K,symb,PolyRing,E,ProjPt):Exports_
symb: List Symbol
OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)

OF ==> OutputForm
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
RFP ==> RootsFindingPackage
SUP ==> SparseUnivariatePolynomial
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)

Exports ==> with

affineAlgSet: List PolyRing -> _
          Union(List(ProjPt),"failed","Infinite",Integer)

affineSingularPoints : PolyRing -> _
          Union(List(ProjPt),"failed","Infinite",Integer)

affineRationalPoints: (PolyRing,PI) -> List ProjPt
++ \texttt{rationalPoints(f,d)} returns all points on the curve
++ \texttt{f} in the extension of the ground field of degree \texttt{d}.
++ For \texttt{d > 1} this only works if \texttt{K} is a
++ \texttt{LocallyAlgebraicallyClosedField}

Implementation ==> add

ss2:List Symbol:= [X1,X2]

DD ==> DistributedMultivariatePolynomial(ss2,K)
LexE ==> DirectProduct(#ss2,NonNegativeInteger)
OV2 ==> OrderedVariableList(ss2)
InGB ==> InterfaceGroebnerPackage(K,ss2,LexE,OV2,DD)

affineAlgSetLocal : List DD -> _
          Union(List(ProjPt),"failed","Infinite",Integer)

import PPFC1
import PolyRing
import ProjPt

listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]

polyToYX1 : PolyRing -> DD
-- NOTE : polyToYX1 set the last variable to 1 and swap the 1st and 2nd var
-- so that a call to grobner will eliminate the second var before the
-- first one
-- 23/10/98 : Ce n'est plus vrai. La fonction a ete "repare".
-- A priori ce la ne creait pas de bug, car on tenait compte de
-- cette particularite dans la fonction affineAlgSetLocal.
-- Cette derniere fct a aussi ete "ajuste"
-- 27/10/98
-- Ce n'est pas vraie !!! Il faut trouve X d'abord et ensuite Y !!
-- sinon tout sur la notion de places distinguee fout le camp !!!

polyToX10 : PolyRing -> SUP(K)

--fonctions de resolution de sys. alg. de dim 0

if K has FiniteFieldCategory then
  affineRationalPoints(crv:PolyRing,extdegree:PI):List(ProjPt) ==
  --The code of this is almost the same as for algebraicSet
  --We could just construct the ideal and call algebraicSet
  --Should we do that? This might be a bit faster.

  listPtsIdl:List(ProjPt):= empty()

  x:= monomial(1,directProduct(vector([1,0])$Vector(NNI)))$DD
  y:= monomial(1,directProduct(vector([0,1])$Vector(NNI)))$DD

  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)$K
    q:= size($K
    px:= x**(q**extdegree) - x
    py:= y**(q**extdegree) - y

  crvXY1 := polyToYX1 crv
  rpts:= affineAlgSetLocal([crvXY1,px,py])

  -- si les 3 tests qui suivent ne sont pas la,
  -- alors ca ne compile pas !!! ???
  rpts case "failed" =>
    error "failed: From affineRationalPoints in AFALGGRO,"
  rpts case "Infinite" =>
    error "Infinite: From affineRationalPoints in AFALGGRO,"
  rpts case Integer =>
    error "Integer: From affineRationalPoints in AFALGGRO,"
  rpts case List(ProjPt) => rpts
  error "Unknown: From affineRationalPoints in AFALGGRO,"

  affineSingularPoints(crb)==
  F:= polyToYX1 crb
  Fx:= differentiate(F,index(1)$OV2)
  Fy:= differentiate(F,index(2)$OV2)
  affineAlgSetLocal([F,Fx,Fy])
\textbf{PACKAGE AFALGGRO AFFINEALGEBRAICSETCOMPUTEWITHGROEBNERBASIS}

```lisp
affineAlgSet(ideal : List PolyRing )==
  idealXY1 := [polyToYX1 pol for pol in ideal]
  affineAlgSetLocal idealXY1

--fonctions de resolution de sys. alg. de dim 0
affineAlgSetLocal(idealToXY1:List DD ) ==
  listPtsIdl:List(ProjPt)
  idealGroXY1:=groebner(idealToXY1)$InGB
  listZeroY:List(K):=empty()
  listZeroX:List(K):=empty()
  listOfExtDeg:List(Integer):=empty()
  polyZeroX:DD:=last(idealGroXY1)
  member?(index(1)$OV2, variables(polyZeroX)$DD) =>
    print(("The number of point in the algebraic set is not finite")::OF)
    print(("or the curve is not absolutely irreducible.")::OF)
    error "Have a nice day"
  --now we find all of the projective points where z ^= 0
  recOfZerosX:=distinguishedRootsOf(univariate(polyZeroX),1$K)$RFP(K)
  -- HERE CHANGE
  degExtX:=recOfZerosX.extDegree
  listZeroX:List K := recOfZerosX.zeros
  listOfExtDeg:=cons(degExtX,listOfExtDeg)
  for a in listZeroX repeat
    tjeker := [(eval(f,index(2)$OV2,a)$DD) for f in idealGroXY1]
    idealGroaXb1 := [univariate(f)$DD for f in tjeker]
    recOfZerosOfIdeal:=distinguishedCommonRootsOf(idealGroaXb1,a)$RFP(K)
    listZeroY:= recOfZerosOfIdeal.zeros
    listOfExtDeg:=cons(recOfZerosOfIdeal.extDegree, listOfExtDeg)
    listPtsIdl:=
      concat( [projectivePoint([a,b,1]) for b in listZeroY],listPtsIdl)
  degExt:=lcm listOfExtDeg
  zero?(degExt) =>
    print(("-------- Infinite number of points --------")::OF)
    "Infinite"
  ^one?(degExt) =>
    print(("You need an extension of degree")::OF)
    print(degExt::OF)
    degExt
    listPtsIdl

polyToYX1(pol)==
  zero?(pol) => 0
  dd:= degree pol
  lc:= leadingCoefficient pol
  pp:= parts dd
  ppr:= rest reverse pp
  ppv:Vector(NNI):= vector ppr
  eppr:=directProduct(ppv)$LexE
  monomial(lc,eppr)$DD + polyToYX1 reductum pol
```

polyToX10(pol) ==
  zero?(pol) => 0
  dd := degree pol
  lc := leadingCoefficient pol
  pp := parts dd
  lp := last pp
  ~zero?(lp) => polyToX10 reductum pol
  e1 := pp.1
  monomial(lc, e1)$SUP(K) + polyToX10 reductum pol

— AFALGGRO.dotabb —

"AFALGGRO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=AFALGGRO"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"AFALGGRO" -> "DIRPCAT"
"AFALGGRO" -> "PFECAT"

package AFALGRES AffineAlgebraicSetComputeWithResultant

— AffineAlgebraicSetComputeWithResultant.input —

)set break resume
/sys rm -f AffineAlgebraicSetComputeWithResultant.output
/spool AffineAlgebraicSetComputeWithResultant.output
/set message test on
/set message auto off
/clear all

--S 1 of 1
/show AffineAlgebraicSetComputeWithResultant
--R
--R AffineAlgebraicSetComputeWithResultant(K: Field, symb: List(Symbol), PolyRing: PolynomialCategory)
--R Abbreviation for AffineAlgebraicSetComputeWithResultant is AFALGRES
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AFALGRES
--R
--R-------------------------------- Operations --------------------------------
AffineAlgebraicSetComputeWithResultant examples

The following is part of the PAFF package

See Also:
  o )show AffineAlgebraicSetComputeWithResultant

Exports:
  affineAlgSet  affineAlgSetLocal  affineRationalPoints
  affineSingularPoints  allPairsAmong  polyRing2UPUP
package AFALGRES AffineAlgebraicSetComputeWithResultant

)abbrev package AFALGRES AffineAlgebraicSetComputeWithResultant
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
AffineAlgebraicSetComputeWithResultant(K,symb,PolyRing,E,ProjPt):Ex==Impl where
  K : Field
  symb : List(Symbol)
  OV ==> OrderedVariableList(symb)
  E : DirectProductCategory(#symb,NonNegativeInteger)
  PolyRing : PolynomialCategory(K,E,OV)
  ProjPt : ProjectiveSpaceCategory(K)
  PCS : LocalPowerSeriesCategory(K)

  SUP ==> SparseUnivariatePolynomial
  UPUP ==> SUP(SUP(K))
  NNI ==> NonNegativeInteger
  RFP ==> RootsFindingPackage

Ex ==> with

  affineSingularPoints: PolyRing -> _
    Union(List(ProjPt),"failed","Infinite",Integer)

  affineSingularPoints: UPUP -> _
    Union(List(ProjPt),"failed","Infinite",Integer)

  affineAlgSetLocal: List UPUP -> _
    Union(List(ProjPt),"failed","Infinite",Integer)

  affineAlgSet: List PolyRing -> _
    Union(List ProjPt,"failed","Infinite",Integer)

polyRing2UPUP: PolyRing -> UPUP

allPairsAmong: List UPUP -> List List UPUP

affineRationalPoints: (PolyRing, PositiveInteger) -> _
  Union(List(ProjPt),"failed","Infinite",Integer)

Impl ==> add

import ProjPt

evAtcoef: (UPUP,K) -> SUP(K)
evAtcoef(pol,a)==
  zero?(pol) => 0
  dd:= degree pol
  lc:= leadingCoefficient pol
  monomial( lc(a), dd )$SUP(K) + evAtcoef( reductum(pol), a )

polyRing2UPUP(pol)==
  zero?(pol) => 0
  dd:= degree pol
  lc:= leadingCoefficient pol
  pp:= parts dd
  monomial(monomial(lc,pp.1)$SUP(K),pp.2)$UPUP+polyRing2UPUP(reductum(pol))

if K has FiniteFieldCategory then
  affineRationalPoints(crv:PolyRing,extdegree:PositiveInteger) ==
  listPtsIdl:List(ProjPt):= empty()
  x:= monomial(1,directProduct(vector([1,0,0])$Vector(NNI)))$PolyRing
  y:= monomial(1,directProduct(vector([0,1,0])$Vector(NNI)))$PolyRing
  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)$K
    q:= size()$K
    px:= x**(q**extdegree) - x
    py:= y**(q**extdegree) - y
  rpts:= affineAlgSet([crv,px,py])
  -- si les 3 tests qui suivent ne sont pas la,
  -- alors ca ne compile pas ???
  rpts case "failed" => _
  error "case failed: From affineRationalPoints in AFALGRES"
  rpts case "Infinite" => _
  error "case infinite: From affineRationalPoints in AFALGRES"
  rpts case Integer => _
  error "case Integer: From affineRationalPoints in AFALGRES"
  rpts case List(ProjPt) => rpts
  error "case unknown: From affineRationalPoints in AFALGRES"

allPairsAmong(lp)==
  #lp = 2 => [lp]
  rlp:=rest lp
  subL:= allPairsAmong rlp
  pol:=first lp
  frontL:= [[pol,p] for p in rlp]
  concat( frontL , subL )

affineSingularPoints(pol:PolyRing)==
  affineSingularPoints( polyRing2UPUP pol )

affineSingularPoints(pol:UPUP)==
  ground? pol => "failed"
lc := coefficients pol
lcb := [ ground?( c )$SUP(K) for c in lc ]
reduce("and", lcb) => "failed"
dy:=differentiate(pol)
dx:=map(differentiate$SUP(K),pol)
affineAlgSetLocal( [ pol, dy, dx ] )

resultantL: List UPUP -> SUP(K)
resultantL(lp)==
g:=first lp
h:= last lp
resultant(g,h)

affineAlgSet(lpol:List PolyRing)==
affineAlgSetLocal( [ polyRing2UPUP pol for pol in lpol ] )

affineAlgSetLocal(lpol:List UPUP)==
listPtsIdl:List(ProjPt)
allP:= allPairsAmong lpol
beforGcd:List SUP(K) := [resultantL(lp) for lp in allP]
polyZeroX:SUP(K):=gcd beforGcd
zero? polyZeroX => "failed"
listZeroY:List(K):=empty()
listZeroX:List(K):=empty()
recOfZerosX:=distinguishedRootsOf(polyZeroX,1$K)$RFP(K)
degExtX:=recOfZerosX.extDegree
listZeroX:List K := recOfZerosX.zeros
listOfExtDeg:List(Integer):=empty()
listOfExtDeg:=cons(degExtX,listOfExtDeg)
lpolEval:List SUP(K)
for a in listZeroX repeat
lpolEval := [ evAtcoef(p,a) for p in lpol ]
recOfZerosOfIdeal:=distinguishedCommonRootsOf( lpolEval ,a)$RFP(K)
listZeroY:= recOfZerosOfIdeal.zeros
listOfExtDeg:=cons(recOfZerosOfIdeal.extDegree,listOfExtDeg)
lstPtsIdl:=
concat( [projectivePoint([a,b,1]) for b in listZeroY] ,lstPtsIdl)
degExt:=lcm listOfExtDeg
zero?(degExt) =>
print("AFALGRES:Infinite number of points")::OutputForm
"Infinite"
"one?(degExt) =>
print("AFALGRES:You need an extension of degree")::OutputForm
print(degExt::OutputForm)
degExt
lstPtsIdl
package AF AlgebraicFunction

-- AlgebraicFunction.input --

)set break resume
)sys rm -f AlgebraicFunction.output
)spool AlgebraicFunction.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show AlgebraicFunction
--R
--R AlgebraicFunction(R:Join(OrderedSet,IntegralDomain),F:FunctionSpace(R)) is a package constructor
--R Abbreviation for AlgebraicFunction is AF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AF
--R
--R-------------------------------------- Operations --------------------------------------
--R belong? : BasicOperator -> Boolean   droot : List(F) -> OutputForm
--R ?**? : (F,Fraction(Integer)) -> F if R has RETRACT(INT)
--R definingPolynomial : F -> F if R has RETRACT(INT)
--R inrootof : (SparseUnivariatePolynomial(F),F) -> F
--R iroot : (R,Integer) -> F if R has RETRACT(INT)
--R minPoly : Kernel(F) -> SparseUnivariatePolynomial(F) if R has RETRACT(INT)
--R operator : BasicOperator -> BasicOperator
--R rootOf : (SparseUnivariatePolynomial(F),Symbol) -> F
--R
--;E 1

)spool
)lisp (bye)

— AlgebraicFunction.help —
AlgebraicFunction examples

This package provides algebraic functions over an integral domain.

See Also:
- )show AlgebraicFunction

---

**AlgebraicFunction (AF)**

```plaintext
Exports:
belong? droot definingPolynomial inrootof iroot
minPoly operator rootOf ?**?
```

---

```plaintext
)abbrev package AF AlgebraicFunction
++ Author: Manuel Bronstein
++ Date Created: 21 March 1988
++ Date Last Updated: 11 November 1993
++ Description:
++ This package provides algebraic functions over an integral domain.
```

```plaintext
AlgebraicFunction(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain)
  F: FunctionSpace R

  SE ==> Symbol
  Z ==> Integer
  Q ==> Fraction Z
  OP ==> BasicOperator
```
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
UPR ==> SparseUnivariatePolynomial R
ALGOP ==> "%alg"
SPECIALDISP ==> "%specialDisp"
SPECIALDIFF ==> "%specialDiff"

Exports ==> with
  rootOf : (UP, SE) -> F
    ++ rootOf(p, y) returns y such that \spad{p(y) = 0}.
    ++ The object returned displays as \spad{'y}.
  operator: OP -> OP
    ++ operator(op) returns a copy of \spad{op} with the domain-dependent
    ++ properties appropriate for \spad{F}.
    ++ Error: if op is not an algebraic operator, that is,
    ++ an nth root or implicit algebraic operator.
  belong?: OP -> Boolean
    ++ belong?(op) is true if \spad{op} is an algebraic operator, that is,
    ++ an nth root or implicit algebraic operator.
  inrootof: (UP, F) -> F
    ++ inrootof(p, x) should be a non-exported function.
    -- un-export when the compiler accepts conditional local functions!
  droot : List F -> OutputForm
    ++ droot(l) should be a non-exported function.
    -- un-export when the compiler accepts conditional local functions!
if R has RetractableTo Integer then
  "**" : (F, Q) -> F
    ++ x ** q is \spad{x} raised to the rational power \spad{q}.
  minPoly: K -> UP
    ++ minPoly(k) returns the defining polynomial of \spad{k}.
  definingPolynomial: F -> F
    ++ definingPolynomial(f) returns the defining polynomial of \spad{f}
    ++ as an element of \spad{F}.
    ++ Error: if f is not a kernel.
  iroot : (R, Z) -> F
    ++ iroot(p, n) should be a non-exported function.
    -- un-export when the compiler accepts conditional local functions!

Implementation ==> add
  ialg : List F -> F
dvalg: (List F, SE) -> F
dalg : List F -> OutputForm
  opalg := operator("rootOf"::Symbol)$CommonOperators
  oproot := operator("nthRoot"::Symbol)$CommonOperators
  belong? op == has?(op, ALGOP)
dalg 1    == second(1)::OutputForm
rootOf(p, x) ==
  k := kernel(x)$K
  (r := retractIfCan(p)@Union(F, "failed")) case "failed" =>
    inrootof(p, k::F)
  n := numer(f := univariate(r::F, k))
  degree denom f > 0 => error "rootOf: variable appears in denom"
  inrootof(n, k::F)

dvalg(l, x) ==
  p := numer univariate(first l, retract(second l)@K)
  alpha := kernel(opalg, l)
  - (map((s:F):F +-> differentiate(s, x), p) alpha)_
    / ((differentiate p) alpha)

ialg l ==
  f := univariate(p := first l, retract(x := second l)@K)
  degree denom f > 0 => error "rootOf: variable appears in denom"
  inrootof(numer f, x)

operator op ==
  is?(op, "rootOf"::Symbol) => opalg
  is?(op, "nthRoot"::Symbol) => oproot
  error "Unknown operator"

if R has AlgebraicallyClosedField then
  UP2R: UP -> Union(UPR, "failed")

inrootof(q, x) ==
  monomial? q => 0
  (d := degree q) <= 0 => error "rootOf: constant polynomial"
  -- one? d=> - leadingCoefficient(reductum q) / leadingCoefficient q
  (d = 1) => - leadingCoefficient(reductum q) / leadingCoefficient q
  ((rx := retractIfCan(x)@Union(SE, "failed")) case SE) and
    ((r := UP2R q) case UPR) => rootOf(r::UPR, rx::SE)::F
  kernel(opalg, [q x, x])

UP2R p ==
  ans:UPR := 0
  while p ^= 0 repeat
    (r := retractIfCan(leadingCoefficient p)@Union(R, "failed"))
    case "failed" => return "failed"
    ans := ans + monomial(r::R, degree p)
    p := reductum p
    ans

else
  inrootof(q, x) ==
    monomial? q => 0
    (d := degree q) <= 0 => error "rootOf: constant polynomial"
PACKAGE AF ALGEBRAICFUNCTION

-- one? d => - leadingCoefficient(reductum q) / leadingCoefficient q
(d = 1) => - leadingCoefficient(reductum q) / leadingCoefficient q
kernel(opalg, [q x, x])

evaluate(opalg, ialg)$BasicOperatorFunctions1(F)
setProperty(opalg, SPECIALDIFF,
   dvalg@((List F, SE) -> F) pretend None)
setProperty(opalg, SPECIALDISP,
   dalg@List F -> OutputForm pretend None)

if R has RetractableTo Integer then
   import PolynomialRoots(IndexedExponents K, K, R, P, F)

dumvar := "%%var"::Symbol::F

lzero : List F -> F
dvroot : List F -> F
inroot : List F -> F
hackroot: (F, Z) -> F
inroot0 : (F, Z, Boolean, Boolean) -> F

lzero l == 0

droot l ==
   x := first(l)::OutputForm
   (n := retract(second l)@Z) = 2 => root x
   root(x, n::OutputForm)

dvroot l ==
   n := retract(second l)@Z
   (first(l) ** ((1 - n) / n)) / (n::F)

x ** q ==
   qr := divide(numer q, denom q)
   x ** qr.quotient * inroot([x, (denom q)::F]) ** qr.remainder

hackroot(x, n) ==
   (n = 1) or (x = 1) => x
   (((dx := denom x) ^= 1) and
       ((rx := retractIfCan(dx)@Union(Integer,"failed"))) case Integer) and
       positive?(rx))
   => hackroot((numer x)::F, n)/hackroot(rx::Integer::F, n)
   (x = -1) and n = 4 =>
       ((-1::F) ** (1::Q / 2::Q) + 1) / ((2::F) ** (1::Q / 2::Q))
   kernel(ooproot, [x, n::F])

inroot l ==
   zero?(n := retract(second l)@Z) => error "root: exponent = 0"
   one?(x := first 1) or one? n => x
   ((x := first 1) = 1) or (n = 1) => x
(r := retractIfCan(x)@Union(R, "failed")) case R => iroot(r::R, n)
(u := isExpt(x, oproot)) case Record(var:K, exponent:Z) =>
      pr := u::Record(var:K, exponent:Z)
      (first argument(pr.var)) **
      (pr.exponent /$Fraction(Z)
       (n * retract(second argument(pr.var))@Z))
inroot0(x, n, false, false)

-- removes powers of positive integers from numer and denom
-- num? or den? is true if numer or denom already processed
inroot0(x, n, num?, den?) ==
      rn:Union(Z, "failed") := (num? => "failed"; retractIfCan numer x)
      rd:Union(Z, "failed") := (den? => "failed"; retractIfCan denom x)
      (rn case Z) and (rd case Z) =>
      rec := qroot(rn::Z / rd::Z, n::NonNegativeInteger)
      rec.coef * hackroot(rec.radicand, rec.exponent)
      rn case Z =>
      rec := qroot(rn::Z::Fraction(Z), n::NonNegativeInteger)
      rec.coef * inroot0((rec.radicand**((n exquo rec.exponent))::Z)
                    / (denom(x)::F), n, true, den?)
      rd case Z =>
      rec0 := qroot(rd::Z::Fraction(Z), n::NonNegativeInteger)
      inroot0((numer(x)::F) /
               (rec.radicand ** (n exquo rec.exponent))::Z),
               n, num?, true) / rec.coef
      hackroot(x, n)

if R has AlgebraicallyClosedField then iroot(r, n) == nthRoot(r, n)::F
else
   iroot0: (R, Z) -> F
   if R has RadicalCategory then
      if R has imaginary:() -> R then iroot(r, n) == nthRoot(r, n)::F
      else
         iroot(r, n) ==
         odd? n or r >= 0 => nthRoot(r, n)::F
         iroot0(r, n)
      else iroot(r, n) == iroot0(r, n)
   else iroot0(r, n) ==
      rec := rroot(r, n::NonNegativeInteger)
      rec.coef * hackroot(rec.radicand, rec.exponent)
   definingPolynomial x ==
      (r := retractIfCan(x)@Union(K, "failed")) case K =>
      is?(k := r::K, opalg) => first argument k
      is?(k, oproot) =>
         dumvar ** retract(second argument k)@Z - first argument k
dumvar - x
dumvar - x

minPoly k ==
  is?(k, opalg) =>
    numer univariate(first argument k, retract(second argument k)@K)
  is?(k, oproot) =>
    monomial(1, retract(second argument k)@Z :: NonNegativeInteger)
    - first(argument k)::UP
    monomial(1, 1) - k::F::UP

evaluate(oproot, inroot)$BasicOperatorFunctions1(F)
derivative(oproot, [dvroot, lzero])

else -- R is not retractable to Integer
droot l ==
  x := first(l)::OutputForm
  (n := second l) = 2::F => root x
  root(x, n::OutputForm)

minPoly k ==
  is?(k, opalg) =>
    numer univariate(first argument k, retract(second argument k)@K)

setProperty(oproot, SPECIALDISP, droot@(List F -> OutputForm) pretend None)

package INTHERAL AlgebraicHermiteIntegration
AlgebraicHermiteIntegration (INTHERAL)
CHAPTER 2. CHAPTER A

— AlgebraicHermiteIntegration.input —

)set break resume
)sys rm -f AlgebraicHermiteIntegration.output
)spool AlgebraicHermiteIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicHermiteIntegration
 --R
 --R AlgebraicHermiteIntegration(F: Field,UP: UnivariatePolynomialCategory(F),UPUP: UnivariatePolynomialCategory(Fraction(UP))) is a package constructor
 --R Abbreviation for AlgebraicHermiteIntegration is INTHERAL
 --R This constructor is not exposed in this frame.
 --R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTHERAL
 --R
 --R------------------------------- Operations --------------------------------
 --R HermiteIntegrate : (R,(UP -> UP)) -> Record(answer: R,logpart: R)
 --E 1

)spool
)lisp (bye)

— AlgebraicHermiteIntegration.help —

====================================================================
AlgebraicHermiteIntegration examples
====================================================================

Algebraic Hermite reduction.

See Also:
 o )show AlgebraicHermiteIntegration
Exports:
HermiteIntegrate

— package INTHERAL AlgebraicHermiteIntegration —

)abbrev package INTHERAL AlgebraicHermiteIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 25 July 1990
++ Description:
++ Algebraic Hermite reduction.

AlgebraicHermiteIntegration(F,UP,UPUP,R):Exports == Implementation where
F : Field
UP : UnivariatePolynomialCategory F
UPUP: UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(F, UP, UPUP)

N ==> NonNegativeInteger
RF ==> Fraction UP

Exports ==> with
HermiteIntegrate: (R, UP -> UP) -> Record(answer:R, logpart:R)
++ HermiteIntegrate(f, \') returns \spad{[g,h]} such that
++ \spad{f = g' + h} and h has a only simple finite normal poles.

Implementation ==> add
localsolve: (Matrix UP, Vector UP, UP) -> Vector UP

-- the denominator of f should have no prime factor P s.t. P | P'
-- (which happens only for P = t in the exponential case)
HermiteIntegrate(f, derivation) ==
ratform:R := 0
n := rank()
m := transpose((mat:= integralDerivationMatrix derivation).num)
inum := (cform := integralCoordinates f).num
if ((iden := cform.den) exquo (e := mat.den)) case "failed" then
iden := (coef := (e exquo gcd(e, iden))::UP) * iden
inum := coef * inum
for trm in factors squareFree iden | (j:= trm.exponent) > 1 repeat
  u' := (u:=(iden exquo (v:=trm.factor)**(j::N))::UP) * derivation v
  sys := ((u * v) exquo e)::UP * m
  nn := minRowIndex sys - minIndex inum
  while j > 1 repeat
    j := j - 1
    p := - j * u'
    sol := localsolve(sys + scalarMatrix(n, p), inum, v)
    ratform := ratform + integralRepresents(sol, v ** (j::N))
    inum := 
    [(((qelt(inum, i) - p * qelt(sol, i) -
        dot(row(sys, i - nn), sol))
      exquo v)::UP - u * derivation qelt(sol, i)
        for i in minIndex inum .. maxIndex inum
    iden := u * v
    [ratform, integralRepresents(inum, iden)]

localsolve(mat, vec, modulus) ==
ans:=Vector(UP) := new(nrows mat, 0)
diagonal? mat =>
  for i in minIndex ans .. maxIndex ans
    for j in minRowIndex mat .. maxRowIndex mat
      for k in minColIndex mat .. maxColIndex mat repeat
        (bc := extendedEuclidean(qelt(mat, j, k), modulus,
          qelt(vec, i))) case "failed" => return new(0, 0)
        qsetelt_!(ans, i, bc.coef1)
    ans
sol := particularSolution(
  map(x+->x::RF, mat)$MatrixCategoryFunctions2(UP,
    Vector UP, Vector UP, Matrix UP, RF,
    Vector RF, Vector RF, Matrix RF),
  map(x+->x::RF, vec)$VectorFunctions2(UP,
    RF))$LinearSystemMatrixPackage(RF,
    Vector RF, Vector RF, Matrix RF)
sol case "failed" => new(0, 0)
for i in minIndex ans .. maxIndex ans repeat
  (bc := extendedEuclidean(denom qelt(sol, i), modulus, 1))
  case "failed" => return new(0, 0)
  qsetelt_!(ans, i, (numer qelt(sol, i) * bc.coef1) rem modulus)
ans

— INTHERAL.dotabb —

"INTHERAL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTHERAL"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
package INTALG AlgebraicIntegrate

--- AlgebraicIntegrate.input ---

)set break resume
)sys rm -f AlgebraicIntegrate.output
)spool AlgebraicIntegrate.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show AlgebraicIntegrate

--R AlgebraicIntegrate(R0: Join(OrderedSet,IntegralDomain,RetractableTo(Integer)),F: Join(AlgebraicallyClosedField),R: Join(ElementaryExtension,ReflectedOn(R0)),R6: Join(IntegralDomain,RetractableTo(Integer)),R4: Join(IntegralDomain,RetractableTo(Integer)),R5: Join(IntegralDomain,RetractableTo(Integer)),R7: Join(IntegralDomain,RetractableTo(Integer)),R8: Join(IntegralDomain,RetractableTo(Integer)),R3: FunctionFieldCategory(F,UP,UPUP),R2: Join(R,F,F)) is a package constructor

--R Abbreviation for AlgebraicIntegrate is INTALG
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTALG

--- AlgebraicIntegrate.help ---

====================================================================
AlgebraicIntegrate examples
====================================================================

This package provides functions for integrating a function
on an algebraic curve.

See Also:
AlgebraicIntegrate (INTALG)

Exports:
algintegrate palginfieldint palgintegrate

--- package INTALG AlgebraicIntegrate ---

AlgebraicIntegrate(R0, F, UP, UPUP, R):Exports == Implementation where
R0  : Join(OrderedSet, IntegralDomain, RetractableTo Integer)
F   : Join(AlgebraicallyClosedField, FunctionSpace R0)
UP  : UnivariatePolynomialCategory F
UPUP : UnivariatePolynomialCategory Fraction UP
R   : FunctionFieldCategory(F, UP, UPUP)

SE  ==> Symbol
Z   ==> Integer
Q   ==> Fraction Z
SUP ==> SparseUnivariatePolynomial F
QF  ==> Fraction UP
GP  ==> LaurentPolynomial(F, UP)
K   ==> Kernel F
PACKAGE INTALG ALGEBRICINTEGRATE

IR ==> IntegrationResult R
UPQ ==> SparseUnivariatePolynomial Q
UPR ==> SparseUnivariatePolynomial R
FRQ ==> Factored UPQ
FD ==> FiniteDivisor(F, UP, UPUP, R)
FAC ==> Record(factor:UPQ, exponent:Z)
LOG ==> Record(scalar:Q, coeff:UPR, logand:UPR)
DIV ==> Record(num:R, den:UP, derivden:UP, gd:UP)
FAIL0 ==> error "integrate: implementation incompleate (constant residues)"
FAIL1 ==> error "integrate: implementation incompleate (non-algebraic residues)"
FAIL2 ==> error "integrate: implementation incompleate (residue poly has multiple non-linear factors)"
FAIL3 ==> error "integrate: implementation incompleate (has polynomial part)"
NOTI ==> error "Not integrable (provided residues have no relations)"

Exports ==> with
  algintegrate : (R, UP -> UP) -> IR
    ++ algintegrate(f, d) integrates f with respect to the derivation d.
  palgintegrate : (R, UP -> UP) -> IR
    ++ palgintegrate(f, d) integrates f with respect to the derivation d.
    ++ Argument f must be a pure algebraic function.
  palgfieldint: (R, UP -> UP) -> Union(R, "failed")
    ++ palgfieldint(f, d) returns an algebraic function g
    ++ such that \( \text{spad}\{dg = f\} \) if such a g exists, "failed" otherwise.
    ++ Argument f must be a pure algebraic function.

Implementation ==> add
import FD
import DoubleResultantPackage(F, UP, UPUP, R)
import PointsOfFiniteOrder(R0, F, UP, UPUP, R)
import AlgebraicHermiteIntegration(F, UP, UPUP, R)
import InnerCommonDenominator(Z, Q, List Z, List Q)
import FunctionSpaceUnivariatePolynomialFactor(R0, F, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R0, SparseMultivariatePolynomial(R0, K), F)

F2R : F -> R
F2UPR : F -> UPR
UP2SUP : UP -> SUP
SUP2UP : SUP -> UP
UPQ2F : UPQ -> UP
univ : (F, K) -> QF
pLogDeriv : (LOG, R -> R) -> R
nonLinear : List FAC -> Union(FAC, "failed")
mkLog : (UP, Q, R, F) -> List LOG
R2UP : (R, K) -> UPR
alglogint : (R, UP -> UP) -> Union(List LOG, "failed")
palglogint : (R, UP -> UP) -> Union(List LOG, "failed")
trace00 : (DIV, UP, List LOG) -> Union(List LOG, "failed")
trace0 : (DIV, UP, Q, FD) -> Union(List LOG, "failed")
trace1 : (DIV, UP, List Q, List FD, Q) -> Union(List LOG, "failed")
nonQ : (DIV, UP) -> Union(List LOG, "failed")
rlift : (F, K, K) -> R
varRoot? : (UP, F -> F) -> Boolean
algintexp : (R, UP -> UP) -> IR
algintprim : (R, UP -> UP) -> IR

dummy:R := 0
dumx := kernel(new()$SE)$K
dumy := kernel(new()$SE)$K

F2UPR f == F2R(f)::UPR
F2R f == f::UP::QF::R

algintexp(f, derivation) ==
  d := (c := integralCoordinates f).den
  v := c.num
  vp:Vector(GP) := new(n := #v, 0)
  vf:Vector(QF) := new(n, 0)
  for i in minIndex v .. maxIndex v repeat
    r := separate(qelt(v, i) / d)$GP
    qsetelt_!(vf, i, r.fracPart)
    qsetelt_!(vp, i, r.polyPart)
  ff := represents(vf, w := integralBasis())
  h := HermiteIntegrate(ff, derivation)
  p := represents(map((x1:GP):QF+->convert(x1)@QF, vp)$VectorFunctions2(GP, QF), w)
  zero?(h.logpart) and zero? p => h.answer::IR
  (u := alglogint(h.logpart, derivation)) case "failed" =>
    mkAnswer(h.answer, empty(), [[p + h.logpart, dummy]])
  zero? p => mkAnswer(h.answer, u::List(LOG), empty())
FAIL3

algintprim(f, derivation) ==
  h := HermiteIntegrate(f, derivation)
  zero?(h.logpart) => h.answer::IR
  (u := alglogint(h.logpart, derivation)) case "failed" =>
    mkAnswer(h.answer, empty(), [[h.logpart, dummy]])
  mkAnswer(h.answer, u::List(LOG), empty())
FAIL3

-- checks whether f = +/\[ci (ui)'/(ui)\]
-- f dx must have no pole at infinity
palglogint(f, derivation) ==
  rec := algSplitSimple(f, derivation)
  ground?(r := doubleResultant(f, derivation)) => "failed"
-- r(z) has roots which are the residues of f at all its poles
  (u := qfactor r) case "failed" => nonQ(rec, r)
  (fc := nonLinear(lf := factors(u::FRQ))) case "failed" => FAIL2
-- at this point r(z) = fc(z) (z - b1)^e1 .. (z - bk)^ek
-- where the ri's are rational numbers, and fc(z) is arbitrary
-- (fc can be linear too)
-- la = [b1, ..., bk] (all rational residues)
  la := [- coefficient(q.factor, 0) for q in remove!(fc::FAC, 1f)]
-- ld = [D1, ..., Dk] where D_i is the sum of places where f has residue b_i
  ld := [divisor(rec.num, rec.den, rec.derivden, rec.gd, b::F) for b in la]
pp := UPQ2F(fc.factor)
-- bb = sum of all the roots of fc (i.e. the other residues)
  zero?(bb := coefficient(fc.factor, (degree(fc.factor) - 1)::NonNegativeInteger)) =>
    cd := [[a1, ..., ak], d] such that bi = ai/d
    g := gcd(a1, ..., ak), so bi = (g/d) c_i with c_i = bi / g
    so [g/d] is a basis for [a1, ..., ak] over the integers
    dv0 is the divisor +/[ci D_i] corresponding to all the residues
 -- of f except the ones which are root of fc(z)
    dv0 := +/[(a quo g) * dv for a in cd.num for dv in ld]
  trace0(rec, pp, g / cd.den, dv0)
trace1(rec, pp, la, ld, bb)

UPQ2F p ==
  map((x:Q):F+->x::F,p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,F,UP)

UP2SUP p ==
  map((x:F):F+->x,p)$UnivariatePolynomialCategoryFunctions2(F, UP, F, SUP)

SUP2UP p ==
  map((x:F):F+->x,p)$UnivariatePolynomialCategoryFunctions2(F, SUP, F, UP)

varRoot?(p, derivation) ==
  for c in coefficients primitivePart p repeat
    derivation(c) ^= 0 => return true
  false

pLogDeriv(log, derivation) ==
  map(derivation, log.coeff) ^= 0 =>
    error "can only handle logs with constant coefficients"

  one?(n := degree(log.coeff)) =>
    (n := degree(log.coeff) = 1) =>
    c := - (leadingCoefficient reductum log.coeff)
    / (leadingCoefficient log.coeff)
    ans := (log.logand) c
    (log.scalar)::R * c * derivation(ans) / ans
    numlog := map(derivation, log.logand)
    (diflog := extendedEuclidean(log.logand, log.coeff, numlog)) case
      "failed" => error "this shouldn't happen"
    algans := diflog.coef1
    ans:=R := 0
    for i in 0..n-1 repeat
\begin{verbatim}
algans := (algans * monomial(1, 1)) rem log.coeff
ans := ans + coefficient(algans, i)
(log.scalar)::R * ans

R2UP(f, k) ==
x := dumx :: F
g :=
(map((f1:QF):F+->f1(x), lift f)_
     $UnivariatePolynomialCategoryFunctions2(QF,UPUP,F,UP))
(y := dumy::F)
map((x1:F):R+->rlift(x1, dumx, dumy), univariate(g, k, minPoly k))_
     $UnivariatePolynomialCategoryFunctions2(F,SUP,R,UPR)

univ(f, k) ==
g := univariate(f, k)
(SUP2UP numer g) / (SUP2UP denom g)

rlift(f, kx, ky) ==
reduce map(x1+->univ(x1, kx), retract(univariate(f, ky))@SUP)_
     $UnivariatePolynomialCategoryFunctions2(F,SUP,QF,UPUP)

nonQ(rec, p) ==
empty? rest(lf := factors ffactor primitivePart p) =>
    trace00(rec, first(lf).factor, empty()$List(LOG))
FAIL1

-- case when the irreducible factor p has roots which sum to 0
-- p is assumed doubly transitive for now
trace0(rec, q, r, dv0) ==
lg:List(LOG) :=
    zero? dv0 => empty()
    (rc0 := torsionIfCan dv0) case "failed" => NOTI
    mkLog(1, r / (rc0.order::Q), rc0.function, 1)
trace00(rec, q, lg)

trace00(rec, pp, lg) ==
p0 := divisor(rec.num, rec.den, rec.derivden, rec.gd,
    alpha0 := zero0f UP2SUP pp)
q := (pp exquo (monomial(1, 1)$UP - alpha0::UP))::UP
alpha := root0f UP2SUP q
dvr := divisor(rec.num, rec.den, rec.derivden, rec.gd, alpha) - p0
    (rc := torsionIfCan dvr) case "failed" =>
        degree(pp) <= 2 => "failed"
    NOTI
    concat(lg, mkLog(q, inv(rc.order::Q), rc.function, alpha))

-- case when the irreducible factor p has roots which sum <> 0
-- the residues of f are of the form [a1,...,ak] rational numbers
-- plus all the roots of q(z), which is squarefree
-- la is the list of residues la := [a1,...,ak]
\end{verbatim}
-- ld is the list of divisors \([D_1, \ldots, D_k]\) where \(D_i\) is the sum of all the
-- places where \(f\) has residue \(a_i\)
-- \(q(z)\) is assumed doubly transitive for now.
-- let \([\alpha_1, \ldots, \alpha_m]\) be the roots of \(q(z)\)
-- in this function, \(b = - \alpha_1 - \ldots - \alpha_m\) is \(\not= 0\)
-- which implies only one generic log term
trace1(rec, q, la, ld, b) ==
-- \(cd = [[b_1, \ldots, b_k], d]\) such that \(\frac{a_i}{b_i} = \frac{b_i}{d}\)
\(cd :=\) splitDenominator \([a / b \text{ for } a \text{ in } la]\)
-- then, a basis for all the residues of \(f\) over the integers is
-- \([\beta_1 = - \frac{\alpha_1}{d}, \ldots, \beta_m = - \frac{\alpha_m}{d}\], since:
-- \(\alpha_i = -d \beta_i\)
-- \(a_i = (a_i / b) \ast b = (b_i / d) \ast b = b_1 \ast \beta_1 + \ldots + b_m \ast \beta_m\)
-- linear independence is a consequence of the doubly transitive assumption
-- \(v_0\) is the divisor \(+[b_i D_i]\) corresponding to the residues \([a_1, \ldots, a_k]\)
\(v_0 := +/[a \ast dv \text{ for } a \text{ in } cd\text{.num for } dv \text{ in } ld]\)
-- \(\alpha\) is a generic root of \(q(z)\)
\(\alpha :=\) rootOf UP2SUP q
-- \(v\) is the divisor corresponding to all the residues
\(v := v_0 - \frac{\text{cd\text{.den}} \ast \text{divisor(rec\text{.num}, rec\text{.den}, rec\text{.derivden}, rec\text{.gd}, alpha)}}}{(rc :=\text{torsionIfCan v}) \text{ case "failed" } \Rightarrow \text{ -- non-torsion case}}\)
\(\text{degree}(q) \leq 2 \Rightarrow \text{"failed" } \text{ -- guaranteed doubly-transitive}}\)
\(\text{NOTI } \text{ -- maybe doubly-transitive}}\)
\(\text{mkLog}(q, \text{inv}((- \text{rc\text{.order}} \ast \text{cd\text{.den}})::Q), \text{rc\text{.function}, alpha}})\)
\(\text{mkLog}(q, \text{scaler, lgd, alpha}) ==\)
\(\text{degree}(q) \leq 1 \Rightarrow\)
\(\text{[[scaler, monomial(1, 1)$UPR - F2UPR alpha, lgd::UPR]\]
[scaler,
\(\text{map}(F2R, q)$UnivariatePolynomialCategoryFunctions2(F,UP,R,UPR),
\(\text{R2UP}(lgd, \text{retract}(alpha)@K)\)])\]
-- return the non-linear factor, if unique
-- or any linear factor if they are all linear
nonLinear l ==
\(\text{found:Boolean} := \text{false}\)
\(\text{ans} := \text{first l}\)
for \(q \text{ in } l\) repeat
if \(\text{degree}(q\text{.factor}) > 1\) then
\(\text{found} \Rightarrow \text{return "failed"}\)
\(\text{found} := \text{true}\)
\(\text{ans} := q\)
\(\text{ans}\)
-- \(f\) \(dx\) must be locally integral at infinity
\(\text{palginfieldint}(f, \text{derivation}) ==\)
\(h :=\text{HermiteIntegrate}(f, \text{derivation})\)
\(\text{zero?(h.logpart}) \Rightarrow h\text{.answer}\)
"failed"
-- f dx must be locally integral at infinity
palgintegrate(f, derivation) ==
  h := HermiteIntegrate(f, derivation)
  zero?(h.logpart) => h.answer::IR
  (not integralAtInfinity?(h.logpart)) or
  ((u := palglogint(h.logpart, derivation)) case "failed") =>
    mkAnswer(h.answer, empty(), [[h.logpart, dummy]])
  zero?(difFirstKind := h.logpart - +/[pLogDeriv(lg,
    x1+->differentiate(x1, derivation)) for lg in u::List(LOG)]) =>
    mkAnswer(h.answer, u::List(LOG), empty())
  mkAnswer(h.answer, u::List(LOG), [[difFirstKind, dummy]])

-- for mixed functions. f dx not assumed locally integral at infinity
algintegrate(f, derivation) ==
  zero? degree(x' := derivation(x := monomial(1, 1)$UP)) =>
    algintprim(f, derivation)
  ((xx := x' exquo x) case UP) and
  (retractIfCan(xx::UP)@Union(F, "failed") case F) =>
    algintexp(f, derivation)
  error "should not happen"
alglogint(f, derivation) ==
  varRoot?(doubleResultant(f, derivation),
    x1+->retract(derivation(x1::UP))@F) => "failed"
FAIL0

—— INTALG.dotabb ——

"INTALG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTALG"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"INTALG" -> "ACF"
"INTALG" -> "FS"
"INTALG" -> "FFCAT"

package INTAF AlgebraicIntegration

—— AlgebraicIntegration.input ——

)set break resume
--S 1 of 1
)show AlgebraicIntegration
--R
--R AlgebraicIntegration(R: Join(OrderedSet,IntegralDomain),F: Join(AlgebraicallyClosedField,FunctionSpace(R))) is a package constructor
--R Abbreviation for AlgebraicIntegration is INTAF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTAF
--R
--R----------------------------------------- Operations -----------------------------------------
--R algint : (F,Kernel(F),Kernel(F),(SparseUnivariatePolynomial(F) -> SparseUnivariatePolynomial(F))) -> IntegrationResult(F)
--R
--E 1

---

--- AlgebraicIntegration.help ---

================================================================================
AlgebraicIntegration examples
================================================================================

This package provides functions for the integration of algebraic integrands over transcendental functions;

See Also:
  o )show AlgebraicIntegration

---
AlgebraicIntegration (INTAF)

Exports:

\begin{verbatim}
    algint
\end{verbatim}

\begin{verbatim}
— package INTAF AlgebraicIntegration —
\end{verbatim}

\begin{verbatim}
)abbrev package INTAF AlgebraicIntegration
++ Author: Manuel Bronstein
++ Date Created: 12 October 1988
++ Date Last Updated: 4 June 1988
++ Description:
++ This package provides functions for the integration of
++ algebraic integrands over transcendental functions;

AlgebraicIntegration(R, F):Exports == Implementation where
  R : Join(OrderedSet, IntegralDomain)
  F : Join(AlgebraicallyClosedField, FunctionSpace R)

SY ==> Symbol
N ==> NonNegativeInteger
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
UPUP==> SparseUnivariatePolynomial RF
IR ==> IntegrationResult F
IR2 ==> IntegrationResultFunctions2(curve, F)
ALG ==> AlgebraicIntegrate(R, F, UP, UPUP, curve)
FAIL==> error "failed - cannot handle that integrand"

Exports ==> with
  algint: (F, K, K, UP -> UP) -> IR
    ++ algint(f, x, y, d) returns the integral of \text{d}(f(x,y))
    ++ where y is an algebraic function of x;
    ++ d is the derivation to use on \text{k}[x].
\end{verbatim}
Implementation ==> add
import ChangeOfVariable(F, UP, UPUP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

rootintegrate: (F, K, K, UP -> UP) -> IR
algintegrate : (F, K, K, UP -> UP) -> IR
UPUP2F : (UPUP, RF, K, K) -> F
F2UPUP : (F, K, K, UP) -> UPUP
UP2UPUP : (UP, K) -> UPUP

F2UPUP(f, kx, k, p) == UP2UPUP(univariate(f, k, p), kx)

rootintegrate(f, t, k, derivation) ==
  r1 := mkIntegral(modulus := UP2UPUP(p := minPoly k, t))
  f1 := F2UPUP(f, t, k, p) monomial(inv(r1.coef), 1)
  r := radPoly(r1.poly)::Record(radicand:RF, deg:N)
  q := retract(r.radicand)
  curve := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
  map(x1+->UPUP2F(lift x1, r1.coef, t, k),
       algintegrate(reduce f1, derivation)$ALG)$IR2

algintegrate(f, t, k, derivation) ==
  r1 := mkIntegral(modulus := UP2UPUP(p := minPoly k, t))
  f1 := F2UPUP(f, t, k, p) monomial(inv(r1.coef), 1)
  modulus := UP2UPUP(p := minPoly k, t)
  curve := AlgebraicFunctionField(F, UP, UPUP, r1.poly)
  map(x1+->UPUP2F(lift x1, r1.coef, t, k),
       algintegrate(reduce f1, derivation)$ALG)$IR2

UP2UPUP(p, k) ==
  map(x1+->univariate(x1,k),p)$SparseUnivariatePolynomialFunctions2(F,RF)

UPUP2F(p, cf, t, k) ==
  map((x1:RF):F+->multivariate(x1, t),
     p)$SparseUnivariatePolynomialFunctions2(RF, F)
   (multivariate(cf, t) * k::F)

algint(f, t, y, derivation) ==
  is?(y, "nthRoot":SY) => rootintegrate(f, t, y, derivation)
  is?(y, "rootOf":SY) => algintegrate(f, t, y, derivation)
  FAIL
package ALGMANIP AlgebraicManipulations

-- AlgebraicManipulations.input --

)set break resume
/sys rm -f AlgebraicManipulations.output
/spool AlgebraicManipulations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicManipulations
  --R AlgebraicManipulations(R: IntegralDomain,F)where
  --R F: Join(Field,ExpressionSpace)with
  --R   numer : % -> SparseMultivariatePolynomial(R,Kernel(%))
  --R   denom : % -> SparseMultivariatePolynomial(R,Kernel(%))
  --R   coerce : SparseMultivariatePolynomial(R,Kernel(%)) -> % is a package constructor
  --R Abbreviation for AlgebraicManipulations is ALGMANIP
  --R This constructor is exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for ALGMANIP
  --R
  --R------------------------------- Operations --------------------------------
  --R   ratDenom : F -> F   ratDenom : (F,F) -> F
  --R   ratDenom : (F,List(F)) -> F   ratDenom : (F,List(Kernel(F))) -> F
  --R   rootSplit : F -> F
  --R   ratPoly : F -> SparseUnivariatePolynomial(F)
  --R   rootKerSimp : (BasicOperator,F,NonNegativeInteger) -> F if F has FS(R) and R has GCDDOM and
  --R   rootPower : F -> F if F has FS(R) and R has GCDDOM and R has ORDSET and R has RETRACT(INT)
  --R   rootProduct : F -> F if F has FS(R) and R has GCDDOM and R has ORDSET and R has RETRACT(INT)
  --R   rootSimp : F -> F if F has FS(R) and R has GCDDOM and R has ORDSET and R has RETRACT(INT)
  --R
  --E 1

)spool
)lisp (bye)
AlgebraicManipulations (ALGMANIP)

Exports:
ratDenom  ratPoly  rootKerSimp  rootPower
rootProduct  rootSimp  rootSplit

— package ALGMANIP AlgebraicManipulations —

)abbrev package ALGMANIP AlgebraicManipulations
++ Author: Manuel Bronstein
++ Date Created: 28 Mar 1988
++ Date Last Updated: 5 August 1993
++ Description:
++ AlgebraicManipulations provides functions to simplify and expand
++ expressions involving algebraic operators.

AlgebraicManipulations(R, F): Exports == Implementation where
  R : IntegralDomain
  F : Join(Field, ExpressionSpace) with
numer : $ \rightarrow $ SparseMultivariatePolynomial(R, Kernel $)
  ++ numer(x) \ undocumented

denom : $ \rightarrow $ SparseMultivariatePolynomial(R, Kernel $)
  ++ denom(x) \ undocumented

coerce : SparseMultivariatePolynomial(R, Kernel $) \rightarrow $
  ++ coerce(x) \ undocumented

N ==> NonNegativeInteger
Z ==> Integer
OP ==> BasicOperator
SY ==> Symbol
K ==> Kernel F
P ==>
SparseMultivariatePolynomial(R, K)
RF ==>
Fraction P
REC ==>
Record(ker:List K, exponent: List Z)
ALGOP ==>
"%alg"
NTHR ==>
"nthRoot"

Exports ==>
with

rootSplit: F \rightarrow F
  ++ rootSplit(f) transforms every radical of the form
  ++ \spad{(a/b)**(1/n)} appearing in f into \spad{a**(1/n) / b**(1/n)}.
  ++ This transformation is not in general valid for all
  ++ complex numbers \spad{a} and b.

ratDenom : F \rightarrow F
  ++ ratDenom(f) rationalizes the denominators appearing in f
  ++ by moving all the algebraic quantities into the numerators.

ratDenom : (F, F) \rightarrow F
  ++ ratDenom(f, a) removes \spad{a} from the denominators in f
  ++ if \spad{a} is an algebraic kernel.

ratDenom : (F, List F) \rightarrow F
  ++ ratDenom(f, [a1,...,an]) removes the ai's which are
  ++ algebraic kernels from the denominators in f.

ratDenom : (F, List K) \rightarrow F
  ++ ratDenom(f, [a1,...,an]) removes the ai's which are
  ++ algebraic from the denominators in f.

ratPoly : F \rightarrow SparseUnivariatePolynomial F
  ++ ratPoly(f) returns a polynomial p such that p has no
  ++ algebraic coefficients, and \spad{p(f) = 0}.

if R has Join(OrderedSet, GcdDomain, RetractableTo Integer)
and F has FunctionSpace(R) then

rootPower : F \rightarrow F
  ++ rootPower(f) transforms every radical power of the form
  ++ \spad{(a**(1/n))**m} into a simpler form if \spad{m} and
  ++ \spad{n} have a common factor.

rootProduct : F \rightarrow F
  ++ rootProduct(f) combines every product of the form
  ++ \spad{(a**(1/n))**m * (a**(1/s))**t} into a single power
  ++ of a root of \spad{a}, and transforms every radical power
  ++ of the form \spad{(a**(1/n))**m} into a simpler form.
rootSimp : F -> F
++ rootSimp(f) transforms every radical of the form
++ \( (a \cdot b^{(q \cdot n + r)})^{(1/n)} \) appearing in f into
++ \( b^{q} \cdot (a \cdot b^{r})^{(1/n)} \).
++ This transformation is not in general valid for all
++ complex numbers b.
rootKerSimp: (OP, F, N) -> F
++ rootKerSimp(op,f,n) should be local but conditional.

Implementation ==> add
import PolynomialCategoryQuotientFunctions(IndexedExponents K,K,R,P,F)

innerRF : (F, List K) -> F
rootExpand : K -> F
algkernels : List K -> List K
rootkernels: List K -> List K
dummy := kernel(new()$SY)$K

ratDenom x == innerRF(x, algkernels tower x)
ratDenom(x:F, l:List K):F == innerRF(x, algkernels l)
ratDenom(x:F, y:F) == ratDenom(x, \[y\]$List(K))
ratDenom(x:F, l:List F) == ratDenom(x, \[retract(y)@K for y in l\]$List(K))
algkernels l == select_!((z1:K):Boolean +-> has?(operator z1, ALGOP), l)
rootkernels l == select_!((z1:K):Boolean +-> is?(operator z1, NTHR::SY), l)

ratPoly x ==
  numer univariate(denom(ratDenom inv(dummy::P::F - x))::F, dummy)

rootSplit x ==
  lk := rootkernels tower x
  eval(x, lk, \[map((z3:F):F+->innerRF(z3,ll), kk) for kk in lk\])
rootExpand k ==
  x := first argument k
  n := second argument k
  op := operator k
  op(numer(x)::F, n) / op(denom(x)::F, n)

-- all the kernels in ll must be algebraic
innerRF(x, ll) ==
  empty?(l := sort_!((z1:K,z2:K):Boolean +-> z1 > z2,kernels x)$List(K)) or
  empty? setIntersection(ll, tower x) => x
  lk := empty()$List(K)
  while not member?(k := first l, ll) repeat
    lk := concat(k, lk)
  empty?(l := rest l) =>
    return eval(x, lk, \[map((z3:F):F+->innerRF(z3,ll), kk) for kk in lk\])
  q := univariate(eval(x, lk,
    \[map((z4:F):F+->innerRF(z4,ll),kk) for kk in lk\],k,minPoly k)
map((z5:F):F+->innerRF(z5, ll), q) \ (map((z6:F):F+->innerRF(z6, ll), k))

if R has Join(OrderedSet, GcdDomain, RetractableTo Integer)
and F has FunctionSpace(R) then
import PolynomialRoots(IndexedExponents K, K, R, P, F)

sroot : K -> F
inroot : (GP, F, N) -> F
radeval: (P, K) -> F
breakup: List K -> List REC

if R has RadicalCategory then
rootKerSimp(op, x, n) ==
  (r := retractIfCan(x)@Union(R, "failed")) case R =>
    nthRoot(r::R, n)::F
    inroot(op, x, n)
else
  rootKerSimp(op, x, n) == inroot(op, x, n)

-- l is a list of nth-roots, returns a list of records of the form
-- [a**((1/n1),a**((1/n2),.,.,[n1,n2,...]]
-- such that the whole list covers l exactly
breakup l ==
  empty? l => empty()
  k := first l
  a := first(arg := argument(k := first l))
  n := retract(second arg)@Z
  expo := empty()$List(Z)
  others := same := empty()$List(K)
  for kk in rest l repeat
    if (a = first(arg := argument kk)) then
      same := concat(kk, same)
      expo := concat(retract(second arg)@Z, expo)
    else others := concat(kk, others)
  ll := breakup others
  concat([concat(k, same), concat(n, expo)], ll)

rootProduct x ==
  for rec in breakup rootkernels tower x repeat
    k0 := first(l := rec.ker)
    nx := numer x; dx := denom x
    if empty? rest l then x := radeval(nx, k0) / radeval(dx, k0)
    else
      n := lcm(rec.exponent)
      k := kernel(operator k0, [first argument k0, n:F], height k0)$K
      lv := [monomial(1, k, (n quo m)::N) for m in rec.exponent]$List(P)
      x := radeval(eval(nx, l, lv), k) / radeval(eval(dx, l, lv), k)
    endif
  endfor
  x

rootPower x ==
for k in rootkernels tower x repeat
  x := radeval(numer x, k) / radeval(denom x, k)
x
-- replaces (a**(1/n))**m in p by a power of a simpler radical of a if
-- n and m have a common factor
radeval(p, k) ==
a := first(arg := argument k)
n := (retract(second arg)@Integer)::NonNegativeInteger
ans:F := 0
q := univariate(p, k)
while (d := degree q) > 0 repeat
term :=
  one?(g := gcd(d, n)) => monomial(1, k, d)
  (g := gcd(d, n)) = 1 => monomial(1, k, d)
  monomial(1, kernel(operator k, [a,(n quo g)::F], height k), d quo g)
ans := ans + leadingCoefficient(q)::F * term::F
q := reductum q
leadingCoefficient(q)::F + ans

inroot(op, x, n) ==
-- one? x => x
-- (x ^= -1) and (one?(num := numer x) or (num = -1)) =>
-- (x ^= -1) and (((num := numer x) = 1) or (num = -1)) =>
inv inroot(op, (num * denom x)::F, n)
(u := isExpt(x, op)) case "failed" => kernel(op, [x, n::F])
pr := u::Record(var:K, exponent:Integer)
q := pr.exponent /$Fraction(Z)
  (n * retract(second argument(pr.var))@Z)
qr := divide(numer q, denom q)
x := first argument(pr.var)
x ** qr.quotient * rootKerSimp(op,x,denom(q)::N) ** qr.remainder

sroot k ==
  pr := froot(first(arg := argument k),(retract(second arg)@Z)::N)
  pr.coef * rootKerSimp(operator k, pr.radicand, pr.exponent)

rootSimp x ==
  lk := rootkernels tower x
eval(x, lk, [sroot k for k in lk])
package ALGMFACT AlgebraicMultFact

--- AlgebraicMultFact.input ---

)set break resume
)sys rm -f AlgebraicMultFact.output
)spool AlgebraicMultFact.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AlgebraicMultFact

--R AlgebraicMultFact(OV: OrderedSet,E: OrderedAbelianMonoidSup,P: PolynomialCategory(AlgebraicNumber,E,OV)) is a package constructor
--R Abbreviation for AlgebraicMultFact is ALGMFACT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ALGMFACT
--R
--R----------------------------------- Operations -----------------------------------
--R factor : (P,List(AlgebraicNumber)) -> Factored(P)
--R factor : (SparseUnivariatePolynomial(P),List(AlgebraicNumber)) -> Factored(SparseUnivariatePolynomial(P))
--E 1

)spool
)lisp (bye)

--- AlgebraicMultFact.help ---

====================================================================
AlgebraicMultFact examples
====================================================================

This package factors multivariate polynomials over the domain of AlgebraicNumber by allowing the user to specify a list of algebraic numbers generating the particular extension to factor over.

See Also:
o )show AlgebraicMultFact

---

AlgebraicMultFact (ALGMFACT)

Exports:
factor

--- package ALGMFACT AlgebraicMultFact ---

)abbrev package ALGMFACT AlgebraicMultFact
++ Author: P. Gianni
++ Date Created: 1990
++ Description:
++ This package factors multivariate polynomials over the
++ domain of \spadtype{AlgebraicNumber} by allowing the user
++ to specify a list of algebraic numbers generating the particular
++ extension to factor over.

AlgebraicMultFact(OV,E,P) : C == T
where
  AN  ==> AlgebraicNumber
  OV  : OrderedSet
  E   : OrderedAbelianMonoidSup
  P   : PolynomialCategory(AN,E,OV)
  BP  ==> SparseUnivariatePolynomial AN
  Z   ==> Integer
  MParFact ==> Record(irr:P,pow:Z)
  USP  ==> SparseUnivariatePolynomial P
  SUParFact ==> Record(irr:USP,pow:Z)
  SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
  MFinalFact ==> Record(contp:R,factors:List MParFact)
-- contp  =  content,
-- factors =  List  of  irreducible  factors  with  exponent
L  ==>  List

C  ==  with
    factor  :  (P,L  AN)  ->  Factored  P
      ++  factor(p,lan)  factors  the  polynomial  p  over  the  extension
      ++  generated  by  the  algebraic  numbers  given  by  the  list  lan.
    factor  :  (USP,L  AN)  ->  Factored  USP
      ++  factor(p,lan)  factors  the  polynomial  p  over  the  extension
      ++  generated  by  the  algebraic  numbers  given  by  the  list  lan.
      ++  p  is  presented  as  a  univariate  polynomial  with  multivariate
      ++  coefficients.
T  ==  add

AF  :=  AlgFactor(BP)

INNER  ==>  InnerMultFact(OV,E,AN,P)

factor(p:P,lalg:L  AN)  :  Factored  P  ==
    factor(p,(z1:BP):Factored(BP)  +->  factor(z1,lalg)$AF)$INNER

factor(up:USP,lalg:L  AN)  :  Factored  USP  ==
    factor(up,(z1:BP):Factored(BP)  +->  factor(z1,lalg)$AF)$INNER

package  ALGPKG  AlgebraPackage

    —  AlgebraPackage.input  —

)set  break  resume
)sys  rm  -f  AlgebraPackage.output
)spool  AlgebraPackage.output
)set  message  test  on
AlgebraPackage.help

AlgebraPackage assembles a variety of useful functions for general algebras.

See Also:
- )show AlgebraPackage

---
AlgebraPackage (ALGPKG)

Exports:

- basis
- basisOfCenter
- basisOfCentroid
- basisOfCommutingElements
- basisOfLeftAnnihilator
- basisOfLeftNucleus
- basisOfLeftNucloid
- basisOfMiddleNucleus
- basisOfNucleus
- basisOfRightAnnihilator
- basisOfRightNucleus
- basisOfRightNucloid
- biRank
- doubleRank
- leftRank
- radicalOfLeftTraceForm
- rightRank
- weakBiRank

— package ALGPKG AlgebraPackage —

)abbrev package ALGPKG AlgebraPackage
++ Authors: J. Grabmeier, R. Wisbauer
++ Date Created: 04 March 1991
++ Date Last Updated: 04 April 1992
++ Reference:
  ++ R.S. Pierce: Associative Algebras
  ++ Graduate Texts in Mathematics 88
  ++
  ++ R.D. Schafer: An Introduction to Nonassociative Algebras
  ++ Academic Press, New York, 1966
  ++
  ++ A. Woerz-Busekros: Algebra in Genetics
  ++ Lectures Notes in Biomathematics 36,
  ++ Springer-Verlag, Heidelberg, 1980
++ Description:
++ AlgebraPackage assembles a variety of useful functions for
++ general algebras.

AlgebraPackage(R: IntegralDomain, A: FramedNonAssociativeAlgebra(R)): _
  public == private where

  V ==> Vector
  M ==> Matrix
  I ==> Integer
  NNI ==> NonNegativeInteger
  REC ==> Record(particular: Union(V R,"failed"),basis: List V R)
**PACKAGE ALGPKG ALGEBRAPACKAGE**

\[ \text{LSMP} \Rightarrow \text{LinearSystemMatrixPackage}(R,V,R,V,M,R) \]

\[ \text{public} \Rightarrow \text{with} \]

- **leftRank**: \( A \rightarrow \text{NonNegativeInteger} \)
  
  + leftRank(x) determines the number of linearly independent elements
  
  + in \( \spad{x*b1}, \ldots, \spad{x*bn} \),
  
  + where \( \spad{b=[b1,\ldots,bn]} \) is a basis.

- **rightRank**: \( A \rightarrow \text{NonNegativeInteger} \)
  
  + rightRank(x) determines the number of linearly independent elements
  
  + in \( \spad{b1*x}, \ldots, \spad{bn*x} \),
  
  + where \( \spad{b=[b1,\ldots,bn]} \) is a basis.

- **doubleRank**: \( A \rightarrow \text{NonNegativeInteger} \)
  
  + doubleRank(x) determines the number of linearly independent elements
  
  + in \( \spad{b1*x}, \ldots, \spad{bn*x} \),
  
  + where \( \spad{b=[b1,\ldots,bn]} \) is a basis.

- **weakBiRank**: \( A \rightarrow \text{NonNegativeInteger} \)
  
  + weakBiRank(x) determines the number of linearly independent elements
  
  + in the \( \spad{b1*x*bj}, \spad{b=j=1,\ldots,n} \),
  
  + where \( \spad{b=[b1,\ldots,bn]} \) is a basis.

- **biRank**: \( A \rightarrow \text{NonNegativeInteger} \)
  
  + biRank(x) determines the number of linearly independent elements
  
  + in \( \spad{x}, \spad{x*b1}, \spad{b1*x}, \spad{bi*x*bj} \),
  
  + where \( \spad{b=[b1,\ldots,bn]} \) is a basis.
  
  + Note that if \( \spad{A} \) has a unit,
  
  + then doubleRank, weakBiRank, and biRank coincide.

- **basisOfCommutingElements**: () \( \rightarrow \) List \( A \)
  
  + basisOfCommutingElements() returns a basis of the space of all \( x \) satisfying \( \spad{0 = \text{commutator}(x,a)} \) for all \( \spad{a} \) in \( \spad{A} \).

- **basisOfLeftAnnihilator**: \( A \rightarrow \) List \( A \)
  
  + basisOfLeftAnnihilator(a) returns a basis of the space of all \( x \) of \( \spad{A} \) satisfying \( \spad{0 = x*a} \).

- **basisOfRightAnnihilator**: \( A \rightarrow \) List \( A \)
  
  + basisOfRightAnnihilator(a) returns a basis of the space of all \( x \) of \( \spad{A} \) satisfying \( \spad{0 = a*x} \).

- **basisOfLeftNucleus**: () \( \rightarrow \) List \( A \)
  
  + basisOfLeftNucleus() returns a basis of the space of all \( x \) of \( \spad{A} \) satisfying \( \spad{0 = \text{associator}(x,a,b)} \) for all \( \spad{a,b} \) in \( \spad{A} \).

- **basisOfRightNucleus**: () \( \rightarrow \) List \( A \)
  
  + basisOfRightNucleus() returns a basis of the space of all \( x \) of \( \spad{A} \) satisfying \( \spad{0 = \text{associator}(a,b,x)} \) for all \( \spad{a,b} \) in \( \spad{A} \).

- **basisOfMiddleNucleus**: () \( \rightarrow \) List \( A \)
  
  + basisOfMiddleNucleus() returns a basis of the space of all \( x \) of \( \spad{A} \) satisfying \( \spad{0 = \text{associator}(a,x,b)} \)
++ for all \texttt{a}, b in \texttt{A}.

\begin{verbatim}
++ basisOfNucleus: () -> List A
++ basisOfNucleus() returns a basis of the space of
++ all x of \texttt{A} satisfying
++ \texttt{associator(x,a,b) = associator(a,x,b) = associator(a,b,x) = 0}
++ for all \texttt{a}, b in \texttt{A}.
\end{verbatim}

\begin{verbatim}
++ basisOfCenter: () -> List A
++ basisOfCenter() returns a basis of the space of
++ all x of \texttt{A} satisfying \texttt{commutator(x,a) = 0} and
++ \texttt{associator(x,a,b) = associator(a,x,b) = associator(a,b,x) = 0}
++ for all \texttt{a}, b in \texttt{A}.
\end{verbatim}

\begin{verbatim}
++ basisOfLeftNucloid: () -> List Matrix R
++ basisOfLeftNucloid() returns a basis of the space of
++ endomorphisms of \texttt{A} as right module.
++ Note that left nucloid coincides with left nucleus
++ if \texttt{A} has a unit.
\end{verbatim}

\begin{verbatim}
++ basisOfRightNucloid: () -> List Matrix R
++ basisOfRightNucloid() returns a basis of the space of
++ endomorphisms of \texttt{A} as left module.
++ Note that right nucloid coincides with right nucleus
++ if \texttt{A} has a unit.
\end{verbatim}

\begin{verbatim}
++ basisOfCentroid: () -> List Matrix R
++ basisOfCentroid() returns a basis of the centroid, i.e. the
++ endomorphism ring of \texttt{A} considered as \texttt{(A,A)}-bimodule.
\end{verbatim}

\begin{verbatim}
++ radicalOfLeftTraceForm: () -> List A
++ radicalOfLeftTraceForm() returns basis for null space of
++ \texttt{leftTraceMatrix()}, if the algebra is
++ associative, alternative or a Jordan algebra, then this
++ space equals the radical (maximal nil ideal) of the algebra.
\end{verbatim}

if \texttt{R} has \texttt{EuclideanDomain} then

\begin{verbatim}
  basis : V A -> V A
  ++ basis(va) selects a basis from the elements of va.
\end{verbatim}

\begin{verbatim}
private ==> add

-- constants

n : PositiveInteger := rank()$A
n2 : PositiveInteger := n*n
n3 : PositiveInteger := n*n2
gamma : Vector Matrix R := structuralConstants()$A

-- local functions

convVM : Vector R -> Matrix R
  -- converts n2-vector to (n,n)-matrix row by row
convMV : Matrix R -> Vector R
  -- converts n-square matrix to n2-vector row by row
\end{verbatim}
convVM v ==
  cond : Matrix(R) := new(n, n, O$R)$M(R)
  z : Integer := 0
  for i in 1..n repeat
    for j in 1..n repeat
      z := z+1
      setelt(cond, i, j, v.z)
  cond

-- convMV m ==
-- vec : Vector(R) := new(n*n, O$R)
-- z : Integer := 0
-- for i in 1..n repeat
--  for j in 1..n repeat
--    z := z+1
--    setelt(vec, z, elt(m, i, j))
-- vec

radicalOfLeftTraceForm() ==
  ma : M R := leftTraceMatrix()$A
  map(represents, nullSpace ma)$ListFunctions2(Vector R, A)

basisOfLeftAnnihilator a ==
  ca : M R := transpose(coordinates(a) :: M R)
  cond : M R := reduce(vertConcat$(M R),
    [ca*transpose(gamma.i) for i in 1..#gamma])
  map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfRightAnnihilator a ==
  ca : M R := transpose(coordinates(a) :: M R)
  cond : M R := reduce(vertConcat$(M R),
    [ca*(gamma.i) for i in 1..#gamma])
  map(represents, nullSpace cond)$ListFunctions2(Vector R, A)

basisOfLeftNucloid() ==
  cond : Matrix(R) := new(n3, n2, O$R)$M(R)
  cond0 : Matrix(R) := new(n3, n2, O$R)$M(R)
  z : Integer := 0
  for i in 1..n repeat
    for j in 1..n repeat
      r1 : Integer := 0
      for k in 1..n repeat
        z := z + 1
        -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
      r2 : Integer := i
      for r in 1..n repeat
        r1 := r1 + 1
-- here \( r_1 \) equals \( (k-1) n + r \) (loop-invariant)
\[
\text{setelt}(\text{cond}, z, r_1, \text{elt}(\gamma, i, j))
\]
-- here \( r_2 \) equals \( (r-1) n + i \) (loop-invariant)
\[
\text{setelt}(\text{condo}, z, r_2, -\text{elt}(\gamma, k, j))
\]
\[
r_2 := r_2 + n
\]
\[
[\text{convVM}(\text{sol}) \text{ for sol in nullSpace(\text{cond}+\text{condo})}]
\]

\text{basisOfCommutingElements()} ==
\begin{itemize}
\item \( \gamma_1 := \text{first } \gamma \)
\item \( \gamma_1 := \gamma_1 - \text{transpose } \gamma_1 \)
\item \( \text{cond} := \text{Matrix}(R) := \gamma_1 :: \text{Matrix}(R) \)
\item \( \text{for } i \text{ in } 2..n \text{ repeat} \)
\item \( \gamma_k := \gamma_i \)
\item \( \gamma_k := \gamma_k - \text{transpose } \gamma_k \)
\item \( \text{cond} := \text{vertConcat}(\text{cond}, \gamma_k :: \text{Matrix}(R))$\text{Matrix}(R) \)
\item \( \text{map}(\text{represents}, \text{nullSpace } \text{cond})$\text{ListFunctions2}(\text{Vector } R, A) \)
\end{itemize}

\[
\text{cond} : \text{M } R := \text{reduce(vertConcat}$(\text{M } R), \ [\text{gam} := \gamma_i) - \text{transpose } \text{gam for } i \text{ in } 1..\#\gamma) \]
\[
\text{map}(\text{represents}, \text{nullSpace } \text{cond})$\text{ListFunctions2}(\text{Vector } R, A) \)

\text{basisOfLeftNucleus()} ==
\begin{itemize}
\item \( \text{condi} := \text{Matrix}(R) := \text{new(n3,n,0$R)}$\text{Matrix}(R) \)
\item \( z := \text{Integer} := 0 \)
\item \( \text{for } k \text{ in } 1..n \text{ repeat} \)
\item \( \text{for } j \text{ in } 1..n \text{ repeat} \)
\item \( \text{for } s \text{ in } 1..n \text{ repeat} \)
\item \( z := z+1 \)
\item \( \text{for } i \text{ in } 1..n \text{ repeat} \)
\item \( \text{entry} := R := 0 \)
\item \( \text{for } l \text{ in } 1..n \text{ repeat} \)
\item \( \text{entry} := \text{entry} + \text{elt}(\gamma, l, j, k) * \text{elt}(\gamma, s, i, l) - \text{elt}(\gamma, l, i, j) * \text{elt}(\gamma, s, l, k) \)
\item \( \text{setelt}(\text{condi}, z, i, \text{entry})$\text{Matrix}(R) \)
\item \( \text{map}(\text{represents}, \text{nullSpace } \text{condi})$\text{ListFunctions2}(\text{Vector } R, A) \)
\end{itemize}

\text{basisOfRightNucleus()} ==
\begin{itemize}
\item \( \text{condo} := \text{Matrix}(R) := \text{new(n3,n,0$R)}$\text{Matrix}(R) \)
\item \( z := \text{Integer} := 0 \)
\item \( \text{for } k \text{ in } 1..n \text{ repeat} \)
\item \( \text{for } j \text{ in } 1..n \text{ repeat} \)
\item \( \text{for } s \text{ in } 1..n \text{ repeat} \)
\item \( z := z+1 \)
\item \( \text{for } i \text{ in } 1..n \text{ repeat} \)
\item \( \text{entry} := R := 0 \)
\item \( \text{for } l \text{ in } 1..n \text{ repeat} \)
\item \( \text{entry} := \text{entry} + \text{elt}(\gamma, l, k, i) * \text{elt}(\gamma, s, j, l) - \text{elt}(\gamma, l, j, k) * \text{elt}(\gamma, s, l, i) \)
\item \( \text{setelt}(\text{condo}, z, i, \text{entry})$\text{Matrix}(R) \)
\item \( \text{map}(\text{represents}, \text{nullSpace } \text{condo})$\text{ListFunctions2}(\text{Vector } R, A) \)
basisOfMiddleNucleus() ==
conda : Matrix(R) := new(n3,n,0$R)$Matrix(R)
z : Integer := 0
for k in 1..n repeat
for j in 1..n repeat
for s in 1..n repeat
z := z+1
for i in 1..n repeat
entry : R := 0
for l in 1..n repeat
entry := entry + elt(gamma.l,j,i)*elt(gamma.s,l,k) - elt(gamma.l,i,k)*elt(gamma.s,j,l)
setelt(conda,z,i,entry)$Matrix(R)
map(represents, nullSpace conda)$ListFunctions2(Vector R,A)

basisOfNucleus() ==
condi: Matrix(R) := new(3*n3,n,0$R)$Matrix(R)
z : Integer := 0
u : Integer := n3
w : Integer := 2*n3
for k in 1..n repeat
for j in 1..n repeat
for s in 1..n repeat
z := z+1
u := u+1
w := w+1
for i in 1..n repeat
entry : R := 0
enter : R := 0
ent : R := 0
for l in 1..n repeat
entry := entry + elt(gamma.l,j,k)*elt(gamma.s,i,l) - elt(gamma.l,i,j)*elt(gamma.s,l,k)
enter := enter + elt(gamma.l,k,i)*elt(gamma.s,j,l) - elt(gamma.l,j,k)*elt(gamma.s,l,i)
ent := ent + elt(gamma.l,j,k)*elt(gamma.s,i,l) - elt(gamma.l,j,i)*elt(gamma.s,l,k)
setelt(condi,z,i,entry)$Matrix(R)
setelt(condi,u,i,enter)$Matrix(R)
setelt(condi,w,i,ent)$Matrix(R)
map(represents, nullSpace condi)$ListFunctions2(Vector R,A)

basisOfCenter() ==
gammal := first gamma
gamma1 := gamma1 - transpose gamma1
cond : Matrix(R) := gammal :: Matrix(R)
for i in 2..n repeat
gammak := gamma.i
CHAPTER 2. CHAPTER A

\[ \text{gammak} := \text{gammak} - \text{transpose gammak} \]
\[ \text{cond} := \text{vertConcat}(\text{cond}, \text{gammak} :: \text{Matrix(R)}) :: \text{Matrix(R)} \]
\[ \text{B} := \text{cond} :: \text{Matrix(R)} \]
\[ \text{condi} :: \text{Matrix(R)} := \text{new}(2*\text{n3}, \text{n}, 0 :: \text{R}) :: \text{Matrix(R)} \]
\[ \text{z} :: \text{Integer} := 0 \]
\[ \text{u} :: \text{Integer} := \text{n3} \]
\[ \text{for k in 1..n repeat} \]
\[ \text{for j in 1..n repeat} \]
\[ \text{for s in 1..n repeat} \]
\[ \text{z} := \text{z+1} \]
\[ \text{u} := \text{u+1} \]
\[ \text{for i in 1..n repeat} \]
\[ \text{entry} :: \text{R} := 0 \]
\[ \text{enter} :: \text{R} := 0 \]
\[ \text{for l in 1..n repeat} \]
\[ \text{entry} := \text{entry} + \text{elt(gamma.l,j,k)} * \text{elt(gamma.s,i,l)} - \text{elt(gamma.l,i,j)} * \text{elt(gamma.s,l,k)} \]
\[ \text{enter} := \text{enter} + \text{elt(gamma.l,k,i)} * \text{elt(gamma.s,j,l)} - \text{elt(gamma.l,j,k)} * \text{elt(gamma.s,l,i)} \]
\[ \text{setelt}(\text{condi}, \text{z}, \text{i}, \text{entry}) :: \text{Matrix(R)} \]
\[ \text{setelt}(\text{condi}, \text{u}, \text{i}, \text{enter}) :: \text{Matrix(R)} \]
\[ \text{D} := \text{vertConcat}(\text{condi}, \text{B}) :: \text{Matrix(R)} \]
\[ \text{map}(\text{represents}, \text{nullSpace D}) :: \text{ListFunctions2(Vector R, A)} \]

\[ \text{basisOfRightNucloid()} \]
\[ \text{cond} :: \text{Matrix(R)} := \text{new}(\text{n3}, \text{n2}, 0 :: \text{R}) :: \text{Matrix(R)} \]
\[ \text{condo} :: \text{Matrix(R)} := \text{new}(\text{n3}, \text{n2}, 0 :: \text{R}) :: \text{Matrix(R)} \]
\[ \text{z} :: \text{Integer} := 0 \]
\[ \text{for i in 1..n repeat} \]
\[ \text{for j in 1..n repeat} \]
\[ \text{r1} :: \text{Integer} := 0 \]
\[ \text{for k in 1..n repeat} \]
\[ \text{z} := \text{z+1} \]
\[ \text{-- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)} \]
\[ \text{r2} :: \text{Integer} := \text{i} \]
\[ \text{for r in 1..n repeat} \]
\[ \text{r1} := \text{r1 + 1} \]
\[ \text{-- here r1 equals (k-1)*n+r (loop-invariant)} \]
\[ \text{setelt}(\text{cond}, \text{z}, \text{r1}, \text{elt(gamma.r,j,i)}) \]
\[ \text{-- here r2 equals (r-1)*n+i (loop-invariant)} \]
\[ \text{setelt}(\text{condo}, \text{z}, \text{r2}, -\text{elt(gamma.k,j,r)}) \]
\[ \text{r2} := \text{r2 + n} \]
\[ \text{[convVM(sol) for sol in nullSpace(\text{cond}+\text{condo})]} \]

\[ \text{basisOfCentroid()} \]
\[ \text{cond} :: \text{Matrix(R)} := \text{new}(2*\text{n3}, \text{n2}, 0 :: \text{R}) :: \text{Matrix(R)} \]
\[ \text{condo} :: \text{Matrix(R)} := \text{new}(2*\text{n3}, \text{n2}, 0 :: \text{R}) :: \text{Matrix(R)} \]
\[ \text{z} :: \text{Integer} := 0 \]
\[ \text{u} :: \text{Integer} := \text{n3} \]
\[ \text{for i in 1..n repeat} \]
for j in 1..n repeat
  r1 : Integer := 0
for k in 1..n repeat
  z := z + 1
  u := u + 1
  -- z equals (i-1)*n*n+(j-1)*n+k (loop-invariant)
  -- u equals n**3 + (i-1)*n*n+(j-1)*n+k (loop-invariant)
  r2 : Integer := i
for r in 1..n repeat
  r1 := r1 + 1
  -- here r1 equals (k-1)*n+r (loop-invariant)
  setelt(cond,z,r1,elt(gamma.r,i,j))
  setelt(cond,u,r1,elt(gamma.r,j,i))
  -- here r2 equals (r-1)*n+i (loop-invariant)
  setelt(condo,z,r2,-elt(gamma.k,r,j))
  setelt(condo,u,r2,-elt(gamma.k,j,r))
  r2 := r2 + n
[convVM(sol) for sol in nullSpace(cond+condo)]

doubleRank x ==
  cond : Matrix(R) := new(2*n,n,0$R)
for k in 1..n repeat
  z : Integer := 0
  u : Integer := n
for j in 1..n repeat
  z := z+1
  u := u+1
  entry : R := 0
  enter : R := 0
for i in 1..n repeat
  entry := entry + elt(x,i)*elt(gamma.k,j,i)
  enter := enter + elt(x,i)*elt(gamma.k,i,j)
  setelt(cond,z,k,entry)$Matrix(R)
  setelt(cond,u,k,enter)$Matrix(R)
rank(cond)$(M R)

weakBiRank(x) ==
  cond : Matrix(R) := new(n^2,n,0$R)$Matrix(R)
  z : Integer := 0
for i in 1..n repeat
  for j in 1..n repeat
    z := z+1
    for k in 1..n repeat
      entry : R := 0
      for l in 1..n repeat
        for s in 1..n repeat
          entry:=entry+elt(x,l)*elt(gamma.s,i,l)*elt(gamma.k,s,j)
        setelt(cond,z,k,entry)$Matrix(R)
      rank(cond)$(M R)
biRank(x) ==
    cond : Matrix(R) := new(n2+2*n+1,n,0$R)$Matrix(R)
    z : Integer := 0
    for j in 1..n repeat
        for i in 1..n repeat
            z := z+1
            for k in 1..n repeat
                entry : R := 0
                for s in 1..n repeat
                    entry:=entry+elt(x,l)*elt(gamma.s,i,l)*elt(gamma.k,s,j)
                setelt(cond,z,k,entry)$Matrix(R)
    u : Integer := n*n
    w : Integer := n*(n+1)
    c := n2 + 2*n + 1
    for j in 1..n repeat
        u := u+1
        w := w+1
        for k in 1..n repeat
            entry : R := 0
            enter : R := 0
            for i in 1..n repeat
                entry := entry + elt(x,i)*elt(gamma.k,i,j)
                enter := enter + elt(x,i)*elt(gamma.k,i,j)
            setelt(cond,u,k,entry)$Matrix(R)
            setelt(cond,w,k,enter)$Matrix(R)
            setelt(cond,c,j, elt(x,j))
    rank(cond)$(M R)

leftRank x ==
    cond : Matrix(R) := new(n,n,0$R)
    for k in 1..n repeat
        for j in 1..n repeat
            entry : R := 0
            for i in 1..n repeat
                entry := entry + elt(x,i)*elt(gamma.k,i,j)
            setelt(cond,j,k,entry)$Matrix(R)
    rank(cond)$(M R)

rightRank x ==
    cond : Matrix(R) := new(n,n,0$R)
    for k in 1..n repeat
        for j in 1..n repeat
            entry : R := 0
            for i in 1..n repeat
                entry := entry + elt(x,i)*elt(gamma.k,j,i)
            setelt(cond,j,k,entry)$Matrix(R)
    rank(cond)$(M R)
if R has EuclideanDomain then
  basis va ==
  v : V A := remove(zero?, va)$V A
  v : V A := removeDuplicates v
  empty? v => [0$A]
  m : Matrix R := coerce(coordinates(v.1))$Matrix R
  for i in 2..maxIndex v repeat
    m := horizConcat(m,coerce(coordinates(v.i))$Matrix R)
  m := rowEchelon m
  lj : List Integer := []
  h : Integer := 1
  mRI : Integer := maxRowIndex m
  mCI : Integer := maxColIndex m
  finished? : Boolean := false
  j : Integer := 1
  while not finished? repeat
    not zero? m(h,j) => -- corner found
      lj := cons(j,lj)
      h := mRI
      while zero? m(h,j) repeat h := h-1
      finished? := (h = mRI)
      if not finished? then h := h+1
    if j < mCI then
      j := j + 1
    else
      finished? := true
      [v.j for j in reverse lj]

package ALGFACT AlgFactor

— AlgFactor.input —

)set break resume
sys rm -f AlgFactor.output
spool AlgFactor.output
set message test on
set message auto off
clear all
--S 1 of 1
show AlgFactor
--R
--R AlgFactor(UP: UnivariatePolynomialCategory(AlgebraicNumber)) is a package constructor
--R Abbreviation for AlgFactor is ALGFACT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ALGFACT
--R
--R----------------------------------- Operations -----------------------------------
--R doublyTransitive? : UP -> Boolean factor : UP -> Factored(UP)
--R split : UP -> Factored(UP)
--R factor : (UP,List(AlgebraicNumber)) -> Factored(UP)
--R
--E 1

spool
lisp (bye)

---

AlgFactor.help

Factorization of univariate polynomials with coefficients in AlgebraicNumber.

See Also:
o )show AlgFactor
AlgFactor (ALGFACT)

Exports:
- doublyTransitive?
- factor
- split

--- package ALGFACT AlgFactor ---

)`abbrev package ALGFACT AlgFactor
++ Author: Manuel Bronstein
++ Description:
++ Factorization of univariate polynomials with coefficients in
++ \spadtype{AlgebraicNumber}.

AlgFactor(UP):Exports == Implementation where
  UP: UnivariatePolynomialCategory AlgebraicNumber

  N  ==> NonNegativeInteger
  Z  ==> Integer
  Q  ==> Fraction Integer
  AN ==> AlgebraicNumber
  K  ==> Kernel AN
  UPQ ==> SparseUnivariatePolynomial Q
  SUP ==> SparseUnivariatePolynomial AN
  FR ==> Factored UP

Exports == with
  factor: (UP, List AN) -> FR
  ++ factor(p, [a1,...,an]) returns a prime factorisation of p
  ++ over the field generated by its coefficients and a1,...,an.
  factor: UP     -> FR
  ++ factor(p) returns a prime factorisation of p
  ++ over the field generated by its coefficients.
  split : UP     -> FR
  ++ split(p) returns a prime factorisation of p
  ++ over its splitting field.
  doublyTransitive?: UP -> Boolean
  ++ doublyTransitive?(p) is true if p is irreducible over
++ over the field $K$ generated by its coefficients, and
++ if \texttt{p(X) / (X - a)} is irreducible over
++ \texttt{K(a)} where \texttt{p(a) = 0}.

Implementation ==> add
import PolynomialCategoryQuotientFunctions(IndexedExponents K,
K, Z, SparseMultivariatePolynomial(Z, K), AN)

UPCF2 ==> UnivariatePolynomialCategoryFunctions2

fact : (UP, List K) -> FR
ifactor : (SUP, List K) -> Factored SUP
extend : (UP, Z) -> FR
allk : List AN -> List K
downpoly: UP -> UPQ
liftpoly: UPQ -> UP
irred? : UP -> Boolean

allk l == removeDuplicates concat [kernels x for x in l]
liftpoly p == map(x +-> x::AN, p)$UPCF2(Q, UPQ, AN, UP)
downpoly p == map(x +-> retract(x)@Q, p)$UPCF2(AN, UP ,Q, UPQ)
ifactor(p,l) == (fact(p pretend UP, l)) pretend Factored(SUP)
factor p == fact(p, allk coefficients p)

factor(p, 1) ==
fact(p, allk removeDuplicates concat(1, coefficients p))

split p ==
fp := factor p
unit(fp) * 
_*/[extend(fc.factor, fc.exponent) for fc in factors fp]

extend(p, n) ==
  -- one? degree p => primeFactor(p, n)
  (degree p = 1) => primeFactor(p, n)
  q := monomial(1, 1)$UP - zeroOf(p pretend SUP)::UP
  primeFactor(q, n) * split((p exquo q)::UP) ** (n::N)

doublyTransitive? p ==
  irred? p and irred?((p exquo 
   (monomial(1, 1)$UP - zeroOf(p pretend SUP)::UP))::UP)

irred? p ==
fp := factor p
  -- one? numberFactors fp and one? nthExponent(fp, 1)
  (numberOfFactors fp = 1) and (nthExponent(fp, 1) = 1)

fact(p, 1) ==
  -- one? degree p => primeFactor(p, 1)
  (degree p = 1) => primeFactor(p, 1)
empty? 1 =>
  dr := factor(downpoly p)$RationalFactorize(UPQ)
  (liftpoly unit dr) * 
  _/*[primeFactor(liftpoly dc.factor,dc.exponent)
    for dc in factors dr]
q := minPoly(alpha := "max"/l)$AN
newl := remove((x:K)::Boolean +-> alpha = x, l)
sez := SimpleAlgebraicExtension(AN, SUP, q)
ups := SparseUnivariatePolynomial sez
fr := factor(map(x +-> reduce univariate(x, alpha, q),p)_
    $UPCF2(AN, UP, sez, ups),_
    x +-> ifactor(x, newl))$InnerAlgFactor(AN, SUP, sez, ups)
newalpha := alpha::AN
map((x:sez)::AN +-> (lift(x)$sez) newalpha, unit fr)_
$UPCF2(sez, ups, AN, UP) * 
  _/*[primeFactor(map((y:sez)::AN +-> (lift(y)$sez) newalpha,fc.factor)_
    $UPCF2(sez, ups, AN, UP),
    fc.exponent) for fc in factors fr]

package INTPACK AnnaNumericalIntegrationPackage

— AnnaNumericalIntegrationPackage.input —

)set break resume
)sys rm -f AnnaNumericalIntegrationPackage.output
)spool AnnaNumericalIntegrationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AnnaNumericalIntegrationPackage
--R
--R AnnaNumericalIntegrationPackage is a package constructor
CHAPTER 2. CHAPTER A

--R Abbreviation for AnnaNumericalIntegrationPackage is INTPACK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTPACK
--R
--R------------------------------- Operations --------------------------------
--R integrate : (Expression(Float),Segment(OrderedCompletion(Float)),Float,Float,RoutinesTable) -> Result
--R integrate : NumericalIntegrationProblem -> Result
--R integrate : (Expression(Float),Segment(OrderedCompletion(Float)),Float,Float) -> Result
--R integrate : (Expression(Float),Segment(OrderedCompletion(Float)),Float) -> Result
--R integrate : (Expression(Float),Segment(OrderedCompletion(Float))) -> Result
--R integrate : (Expression(Float),List(Segment(OrderedCompletion(Float)))) -> Result
--R integrate : (Expression(Float),List(Segment(OrderedCompletion(Float))),Float) -> Result
--R integrate : (Expression(Float),List(Segment(OrderedCompletion(Float))),Float,Float) -> Result
--R integrate : (Expression(Float),List(Segment(OrderedCompletion(Float))),Float,Float,RoutinesTable) -> Result
--R integrate : (Expression(Float),SegmentBinding(OrderedCompletion(Float)),String) -> Union(Result,"failed")
--R integrate : (Expression(Float),SegmentBinding(OrderedCompletion(Float)),Symbol) -> Union(Result,"failed")
--R measure : NumericalIntegrationProblem -> Record(measure: Float,name: String,explanations: List(String),extra: Result)
--R measure : (NumericalIntegrationProblem,RoutinesTable) -> Record(measure: Float,name: String,explanations: List(String),extra: Result)

--E 1

)spool
)lisp (bye)

—— AnnaNumericalIntegrationPackage.help ——

====================================================================
AnnaNumericalIntegrationPackage examples
====================================================================

AnnaNumericalIntegrationPackage is a package of functions for the
category NumericalIntegrationCategory with measure and integrate.

See Also:
o )show AnnaNumericalIntegrationPackage
AnnaNumericalIntegrationPackage (INTPACK)

Exports:
integrate  measure

--- package INTPACK AnnaNumericalIntegrationPackage ---

)abbrev package INTPACK AnnaNumericalIntegrationPackage
++ Author: Brian Dupee
++ Date Created: August 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{AnnaNumericalIntegrationPackage} is a \axiom{package}
++ of functions for the \axiom{category}
++ \axiomType{NumericalIntegrationCategory}
++ with \axiom{measure}, and \axiom{integrate}.

AnnaNumericalIntegrationPackage(): EE == II where

EDF ==> Expression DoubleFloat
DF ==> DoubleFloat
EF ==> Expression Float
F ==> Float
INT ==> Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
SBDCF ==> SegmentBinding OrderedCompletion Float
LSDCF ==> List Segment OrderedCompletion Float
SCOF ==> Segment OrderedCompletion Float
OCF ==> OrderedCompletion Float
LS ==> List Symbol
S ==> Symbol
LST ==> List String
ST ==> String
RT ==> RoutinesTable
NIA ==> Record(var:S, fn:EDF, range:SOCDF, abserr:DF, relerr:DF)
MDNIA ==> Record(fn:EDF,range:List SOCDF,abserr:DF,relerr:DF)
IFL ==> List(Record(ifail:Integer,instruction:String))
Entry ==> Record(chapter:String, type:String, domainName: String,
Measure ==> Record(measure:F, name:ST, explanations:LST, extra:Result)
EE ==> with
  integrate: (EF,SOCF,F,F,RT) -> Result
    ++ integrate(exp, a..b, epsrel, routines) is a top level ANNA function
    ++ to integrate an expression, \{\tt exp\}, over a given range \{\tt a\}
    ++ to \{\tt b\} to the required absolute and relative accuracy using
    ++ the routines available in the RoutinesTable provided.
    ++
    ++ It iterates over the \axiom{domains} of
    ++ \axiomType{NumericalIntegrationCategory}
    ++ to get the name and other
    ++ relevant information of the the (domain of the) numerical
    ++ routine likely to be the most appropriate,
    ++ i.e. have the best \axiom{measure}.
    ++
    ++ It then performs the integration of the given expression
    ++ on that \axiom{domain}.

integrate: NumericalIntegrationProblem -> Result
  ++ integrate(IntegrationProblem) is a top level ANNA function
  ++ to integrate an expression over a given range or ranges
  ++ to the required absolute and relative accuracy.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ It then performs the integration of the given expression
  ++ on that \axiom{domain}.

integrate: (EF,SOCF,F,F) -> Result
  ++ integrate(exp, a..b, epsabs, epsrel) is a top level ANNA function
  ++ to integrate an expression, \{\tt exp\}, over a given range \{\tt a\}
  ++ to \{\tt b\} to the required absolute and relative accuracy.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ It then performs the integration of the given expression
  ++ on that \axiom{domain}.

integrate: (EF,SOCF,F) -> Result
  ++ integrate(exp, a..b, epsrel) is a top level ANNA function
  ++ to integrate an expression, \{\tt exp\}, over a given range \{\tt a\}
  ++ to \{\tt b\} to the required absolute and relative accuracy.
  ++
  ++ It iterates over the \axiom{domains} of
  ++ \axiomType{NumericalIntegrationCategory} to get the name and other
  ++ relevant information of the the (domain of the) numerical
  ++ routine likely to be the most appropriate,
  ++ i.e. have the best \axiom{measure}.
  ++
  ++ It then performs the integration of the given expression
  ++ on that \axiom{domain}.
++ function to integrate an expression, \{tt exp\}, over a given
++ range \{tt a\} to \{tt b\} to the required relative accuracy.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ If epsrel = 0, a default absolute accuracy is used.

\begin{verbatim}
integrate: (EF,SCOF) -> Result
integrate(exp, a..b) is a top
++ level ANNA function to integrate an expression, \{tt exp\},
++ over a given range \{tt a\} to \{tt b\}.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ Default values for the absolute and relative error are used.
\end{verbatim}

\begin{verbatim}
integrate:(EF,LSOCF) -> Result
integrate(exp, [a..b,c..d,...]) is a top
++ level ANNA function to integrate a multivariate expression, \{tt exp\},
++ over a given set of ranges.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ Default values for the absolute and relative error are used.
\end{verbatim}

\begin{verbatim}
integrate:(EF,LSOCF,F) -> Result
integrate(exp, [a..b,c..d,...], epsrel) is a top
++ level ANNA function to integrate a multivariate expression, \{tt exp\},
++ over a given set of ranges to the required relative
++ accuracy.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ If epsrel = 0, a default absolute accuracy is used.

\texttt{integrate:(EF,LSOCF,F,F) -> Result}
++ integrate(exp, [a..b,c..d,...], epsabs, epsrel) is a top
++ level ANNA function to integrate a multivariate expression, {\tt exp},
++ over a given set of ranges to the required absolute and relative
++ accuracy.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.

\texttt{integrate:(EF,LSOCF,F,F,RT) -> Result}
++ integrate(exp, [a..b,c..d,...], epsabs, epsrel, routines) is a top
++ level ANNA function to integrate a multivariate expression, {\tt exp},
++ over a given set of ranges to the required absolute and relative
++ accuracy, using the routines available in the RoutinesTable provided.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.

\texttt{measure:NumericalIntegrationProblem -> Measure}
++ measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine for solving the numerical integration
++ problem defined by \axiom{prob}.
++
++ It calls each \axiom{domain} of \axiom{category}
++ \axiomType{NumericalIntegrationCategory} in turn to calculate all measures
++ and returns the best
++ i.e. the name of the most appropriate domain and any other relevant
++ information.
measure:(NumericalIntegrationProblem,RT) -> Measure
++ measure(prob,R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical integration
++ problem defined by \axiom{prob}.
++
++ It calls each \axiom{domain} listed in \axiom{R} of \axiom{category}
++ \axiomType{NumericalIntegrationCategory} in turn to calculate all measures
++ and returns the best
++ i.e. the name of the most appropriate domain and any other relevant
++ information.
integrate:(EF,SBOCF,ST) -> Union(Result,"failed")
++ integrate(exp, x = a..b, "numerical") is a top level ANNA function to
++ integrate an expression, \tt{exp}, over a given range, \tt{a}
++ to \tt{b}.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ Default values for the absolute and relative error are used.
++
++ It is an error if the last argument is not \tt{"numerical"}.
integrate:(EF,SBOCF,S) -> Union(Result,"failed")
++ integrate(exp, x = a..b, numerical) is a top level ANNA function to
++ integrate an expression, \tt{exp}, over a given range, \tt{a}
++ to \tt{b}.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalIntegrationCategory} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ It then performs the integration of the given expression
++ on that \axiom{domain}.
++
++ Default values for the absolute and relative error are used.
++
++ It is an error if the last argument is not \tt{numerical}.

II ==> add
import ExpertSystemToolsPackage

integrateConstantList(exp:EF, ras:LSOCF): Result ==
  c: OCF := ((retract(exp)@F)$EF)::OCF
  b := [hi(j) - lo(j) for j in ras]
  c := c*reduce((x,y) +-> x*y,b)
  a := coerce(c)$AnyFunctions1(OCF)
  text := coerce("Constant Function")$AnyFunctions1(ST)
  construct([[result@S,a], [method@S,text]])$Result

integrateConstant(exp:EF, ra: SOCF): Result ==
  c := (retract(exp)@F)$EF
  r: OCF := (c::OCF)*(hi(ra)-lo(ra))
  a := coerce(r)$AnyFunctions1(OCF)
  text := coerce("Constant Function")$AnyFunctions1(ST)
  construct([[result@S,a], [method@S,text]])$Result

zeroMeasure(m: Measure): Result ==
  a := coerce(0$DF)$AnyFunctions1(DF)
  text := coerce("Constant Function")$AnyFunctions1(String)
  r := construct([[result@Symbol,a], [method@Symbol,text]])$Result
  concat(measure2Result m, r)$ExpertSystemToolsPackage

scriptedVariables?(mdnia: MDNIA): Boolean ==
  vars: List Symbol := variables(mdnia.fn)$EDF
  var1 := first(vars)$List Symbol
  not scripted?(var1) => false
  name1 := name(var1)$Symbol
  for i in 2..# vars repeat
    not ((scripted?(vars.i)$Symbol) and (name1 = name(vars.i)$Symbol)) =>
      true
      return false
  true

preAnalysis(args: Union(nia: NIA, mdnia: MDNIA), t: RT): RT ==
  import RT
  r: RT := selectIntegrationRoutines t
  args case nia =>
arg := args.nia

rangeIsFinite(arg) =>
    selectFiniteRoutines
    selectNonFiniteRoutines
    selectMultiDimensionalRoutines

changeName(ans, name) =>
    sy := coerce(name "Answer")
    anyAns := coerce(ans)$AnyFunctions1
    construct([[sy, anyAns]])

measureSpecific(name, R, args) =>
    args case nia =>
        arg := args.nia
        name = "d01ajfAnnaType" => measure(R, arg)$d01ajfAnnaType
        name = "d01akfAnnaType" => measure(R, arg)$d01akfAnnaType
        name = "d01alfAnnaType" => measure(R, arg)$d01alfAnnaType
        name = "d01amfAnnaType" => measure(R, arg)$d01amfAnnaType
        name = "d01amfAnnaType" => measure(R, arg)$d01amfAnnaType
        name = "d01amfAnnaType" => measure(R, arg)$d01amfAnnaType
        name = "d01apfAnnaType" => measure(R, arg)$d01apfAnnaType
        name = "d01apfAnnaType" => measure(R, arg)$d01apfAnnaType
        name = "d01apfAnnaType" => measure(R, arg)$d01apfAnnaType
        name = "d01aafAnnaType" => measure(R, arg)$d01aafAnnaType
        name = "d01aafAnnaType" => measure(R, arg)$d01aafAnnaType
        name = "d01aafAnnaType" => measure(R, arg)$d01aafAnnaType
        name = "d01TransformFunctionType" => measure(R, arg)$d01TransformFunctionType
        error("measureSpecific","invalid type name: " name)$ErrorFunctions
    args case mdnia =>
        arg2 := args.mdnia
        name = "d01gbfAnnaType" => measure(R, arg2)$d01gbfAnnaType
        name = "d01fcfAnnaType" => measure(R, arg2)$d01fcfAnnaType
        error("measureSpecific","invalid type name: " name)$ErrorFunctions
        error("measureSpecific","invalid type name")$ErrorFunctions

measure(a, R) =>
    args := retract(a)$NumericalIntegrationProblem
   sofar := 0
    best := "none"
    routs := copy R
    preAnalysis(args, routs)
    empty?(routs) =>
        error("measure", "no routines found")$ErrorFunctions
    rout := inspect(routs)
    e := retract(rout.entry)$AnyFunctions1
    ext := empty()
    for i in 1..# routs repeat
        rout := extract!(routs)
        e := retract(rout.entry)$AnyFunctions1
        n := e.domainName
        if e.defaultMin > sofar then
m := measureSpecific(n,R,args)
if m.measure > sofar then
  sofar := m.measure
  best := n
ext := concat(m.extra,ext)$ExpertSystemToolsPackage
str:LST := [string(rout.key)$S "measure: " outputMeasure(m.measure)
  " - " m.explanations]
else
  str:LST := [string(rout.key)$S " is no better than other routines"]
meth := append(meth,str)$LST
[sofar,best,meth,ext]

measure(a:NumericalIntegrationProblem):Measure ==
  measure(a,routines()$RT)

integrateSpecific(args:Union(nia:NIA,mdnia:MDNIA),n:ST,ex:Result):Result ==
  args case nia =>
    arg:NIA := args.nia
    n = "d01ajfAnnaType" => numericalIntegration(arg,ex)$d01ajfAnnaType
    n = "d01TransformFunctionType" =>
      numericalIntegration(arg,ex)$d01TransformFunctionType
    n = "d01amfAnnaType" => numericalIntegration(arg,ex)$d01amfAnnaType
    n = "d01apfAnnaType" => numericalIntegration(arg,ex)$d01apfAnnaType
    n = "d01aqfAnnaType" => numericalIntegration(arg,ex)$d01aqfAnnaType
    n = "d01alfAnnaType" => numericalIntegration(arg,ex)$d01alfAnnaType
    n = "d01akfAnnaType" => numericalIntegration(arg,ex)$d01akfAnnaType
    n = "d01anfAnnaType" => numericalIntegration(arg,ex)$d01anfAnnaType
    n = "d01asfAnnaType" => numericalIntegration(arg,ex)$d01asfAnnaType
    error("integrateSpecific","invalid type name: " n)$ErrorFunctions
  args case mdnia =>
    arg2:MDNIA := args.mdnia
    n = "d01gbfAnnaType" => numericalIntegration(arg2,ex)$d01gbfAnnaType
    n = "d01fcfAnnaType" => numericalIntegration(arg2,ex)$d01fcfAnnaType
    error("integrateSpecific","invalid type name: " n)$ErrorFunctions
    error("integrateSpecific","invalid type name: " n)$ErrorFunctions

better?(r:Result,s:Result):Boolean ==
  a1 := search("abserr":S,r)$Result
  a1 case "failed" => false
  abserr1 := retract(a1)$AnyFunctions1(DF)
  negative?(abserr1) => false
  a2 := search("abserr":S,s)$Result
  a2 case "failed" => true
  abserr2 := retract(a2)$AnyFunctions1(DF)
  negative?(abserr2) => true
  (abserr1 < abserr2) -- true if r.abserr better than s.abserr

recoverAfterFail(n:Union(nia:NIA,mdnia:MDNIA),routs:RT,m:Measure,iint:INT,
  r:Result):Record(a:Result,b:Measure) ==
  bestName := m.name
while positive?(iint) repeat
  routineName := m.name
  s := recoverAfterFail(routs, routineName(1..6), iint)$RoutinesTable
  s case "failed" => iint := 0
  if s = "changeEps" then
    nn := n.nia
    zero?(nn.abserr) =>
      nn.abserr := 1.0e-8 :: DF
    m := measure(n::NumericalIntegrationProblem, routs)
    zero?(m.measure) => iint := 0
    r := integrateSpecific(n, m.name, m.extra)
    iint := 0
  rn := routineName(1..6)
  buttVal := getButtonValue(rn, "functionEvaluations")$AttributeButtons
  if (s = "incrFunEvals") and (buttVal < 0.8) then
    increase(rn, "functionEvaluations")$AttributeButtons
  if s = "increase tolerance" then
    (n.nia).relerr := (n.nia).relerr*(10.0::DF)
  if s = "decrease tolerance" then
    (n.nia).relerr := (n.nia).relerr/(10.0::DF)
  fl := coerce(s)$AnyFunctions1(ST)
  flrec:Record(key:S, entry:Any):=[failure@S, fl]
  m2 := measure(n::NumericalIntegrationProblem, routs)
  zero?(m2.measure) => iint := 0
  r2:Result := integrateSpecific(n, m2.name, m2.extra)
  better?(r, r2) =>
    m.name := m2.name
    insert!(flrec, r)$Result
    bestName := m2.name
    m := m2
    insert!(flrec, r2)$Result
    r := concat(r2, changeName(r, routineName))$ExpertSystemToolsPackage
  iany := search(ifail@S, r2)$Result
  iany case "failed" => iint := 0
  iint := retract(iany)$AnyFunctions1(INT)
  m.name := bestName
[r, m]

integrateArgs(prob::NumericalIntegrationProblem, t:RT):Result ==
  args:Union(nia:NIA, mdnia:MDNIA) := retract(prob)$NumericalIntegrationProblem
  routs := copy(t)$RT
  if args case mdnia then
    arg := args.mdnia
    v := (# variables(arg.fn))
    not scriptedVariables?(arg) =>
      error("MultiDimensionalNumericalIntegrationPackage",
        "invalid variable names")$ErrorFunctions
    (v ~= # arg.range)@Boolean =>
      error("MultiDimensionalNumericalIntegrationPackage",
        "number of variables do not match number of ranges")$ErrorFunctions
m := measure(prob,routs)
zero?(m.measure) => zeroMeasure m
r := integrateSpecific(args,m.name,m.extra)
iany := search@ifail@S,r$Result
iint := 0$INT
if (iany case Any) then
  iint := retract(iany)$AnyFunctions1(INT)
if positive?(iint) then
  tu:Record(a:Result,b:Measure) := recoverAfterFail(args,routs,m,iint,r)
  r := tu.a
  m := tu.b
r := concat(measure2Result m,r)$ExpertSystemToolsPackage
n := m.name
nn:ST :=
  (# n > 14) => "d01transform"
  n(1..6)
expl := getExplanations(routs,nn)$RoutinesTable
expla := coerce(expl)$AnyFunctions1(LST)
explaa:Record(key:Symbol,entry:Any) := ["explanations"::Symbol,expla]
r := concat(construct([explaa]),r)
args case nia =>
  att := showAttributes(args.nia)$IntegrationFunctionsTable
  att case "failed" => r
  concat(att2Result att,r)$ExpertSystemToolsPackage
r
integrate(args:NumericalIntegrationProblem):Result ==
  integrateArgs(args,routines()$RT)
  Var:LS := variables(exp)$EF
  empty?(Var)$LS => integrateConstant(exp,ra)
  args:NIA := [first(Var)$LS,ef2edf exp,socf2socdf ra,f2df epsabs,f2df epsrel]
  integrateArgs(args::NumericalIntegrationProblem,r)
  integrate(exp,ra,0$F,1.0E-5)
integrate(exp:EF,ra:SOCF):Result ==
  integrate(exp,ra,0$F,1.0E-5)
integrate(exp:EF,ra:SOCF, err:F):Result ==
  positive?(err)$F => integrate(exp,ra,0$F, err)
  integrate(exp,ra,1.0E-5, err)
integrate(exp:EF,ra:SOCF):Result ==
  integrate(exp,ra,0$F,1.0E-5)
integrate(exp:EF,ra:SOCF, err:F):Result ==
  positive?(err)$F => integrate(exp,ra,0$F, err)
  integrate(exp,ra,1.0E-5, err)
integrate(exp:EF,ra:SOCF):Result ==
  integrate(exp,ra,0$F,1.0E-5)
integrate(exp:EF,ra:SOCF, err:F):Result ==
  positive?(err)$F => integrate(exp,ra,0$F, err)
  integrate(exp,ra,1.0E-5, err)
integrate(exp:EF,ra:SOCF):Result ==
  integrate(exp,ra,0$F,1.0E-5)
"failed"

\[
\]
\[
\text{vars := variables(exp)$EF}
\]
\[
\text{empty?(vars)$LS => integrateConstantList(exp,ra)}
\]
\[
\text{args:MDNIA := [ef2edf exp,convert ra,f2df epsabs,f2df epsrel]}\]
\[
\text{integrateArgs(args::NumericalIntegrationProblem,r)}
\]
\[
\text{integrate(exp:EF,ra:LSOCF,epsabs:F,epsrel:F):Result ==}
\]
\[
\text{integrate(exp,ra,epsabs,epsrel,routines()$RT)}
\]
\[
\text{integrate(exp:EF,ra:LSOCF,epsrel:F):Result ==}
\]
\[
\text{zero? epsrel => integrate(exp,ra,1.0e-6,epsrel)}\]
\[
\text{integrate(exp,ra,0$F,epsrel)}
\]
\[
\text{integrate(exp:EF,ra:LSOCF):Result == integrate(exp,ra,1.0e-4)}
\]

---

— INTPACK.dotabb —

"INTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPACK"]

"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]

"INTPACK" -> "TBAGG"

---

package OPTPACK AnnaNumericalOptimizationPackage

— AnnaNumericalOptimizationPackage.input —

)set break resume
)sys rm -f AnnaNumericalOptimizationPackage.output
)spool AnnaNumericalOptimizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AnnaNumericalOptimizationPackage
--R
--R AnnaNumericalOptimizationPackage is a package constructor
--R Abbreviation for AnnaNumericalOptimizationPackage is OPTPACK
--R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for OPTPACK
---R
---R------------------------------------- Operations -----------------------------------
---R goodnessOfFit : NumericalOptimizationProblem -> Result
---R goodnessOfFit : (List(Expression(Float)),List(Float)) -> Result
---R measure : NumericalOptimizationProblem -> Record(measure: Float,name: String,explanations: List(String))
---R measure : (NumericalOptimizationProblem,RoutinesTable) -> Record(measure: Float,name: String,explanations: List(String))
---R optimize : (NumericalOptimizationProblem,RoutinesTable) -> Result
---R optimize : NumericalOptimizationProblem -> Result
---R optimize : (Expression(Float),List(Float),List(OrderedCompletion(Float)),List(Expression(Float)),List(OrderedCompletion(Float))) -> Result
---R optimize : (Expression(Float),List(Float),List(OrderedCompletion(Float)),List(OrderedCompletion(Float))) -> Result
---R optimize : (Expression(Float),List(Float)) -> Result
---R optimize : (List(Expression(Float)),List(Float)) -> Result
---E 1

)spool
)lisp (bye)

---

AnnaNumericalOptimizationPackage.help

AnnaNumericalOptimizationPackage examples

AnnaNumericalOptimizationPackage is a package of functions for the NumericalOptimizationCategory with measure and optimize.

See Also:
  o )show AnnaNumericalOptimizationPackage

---
AnnaNumericalOptimizationPackage (OPTPACK)

Exports:
  goodnessOfFit  measure  optimize

— package OPTPACK AnnaNumericalOptimizationPackage —

)abbrev package OPTPACK AnnaNumericalOptimizationPackage
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{AnnaNumericalOptimizationPackage} is a \axiom{package} of
++ functions for the \axiomType{NumericalOptimizationCategory}
++ with \axiom{measure} and \axiom{optimize}.

AnnaNumericalOptimizationPackage(): EE == II where

  EDF  ==> Expression DoubleFloat
  LEDF ==> List Expression DoubleFloat
  LDF  ==> List DoubleFloat
  MDF  ==> Matrix DoubleFloat
  DF   ==> DoubleFloat
  LOCDF ==> List OrderedCompletion DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat
  LOCF ==> List OrderedCompletion Float
  OCF  ==> OrderedCompletion Float
  LEF  ==> List Expression Float
  EF   ==> Expression Float
  LF   ==> List Float
  F    ==> Float
  LS   ==> List Symbol
  LST  ==> List String
  INT  ==> Integer
  NOA  ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)
  LSA  ==> Record(lfn:LEDF, init:LDF)
  IFL  ==> List(Record(ifail:Integer,instruction:String))
CHAPTER 2. CHAPTER A

Entry ==> Record(chapter:String, type:String, domainName: String,
Measure ==> Record(measure:F, name:String, explanations:List String)
Measure2 ==> Record(measure:F, explanations:String)
RT ==> RoutinesTable
UNOALSA ==> Union(noa:NOA, lsa:LSA)

EE ==> with
measure: NumericalOptimizationProblem -> Measure
++ measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical optimization problem defined by
++ \texttt{\textbackslash axiom\{}prob\texttt{\}\} by checking various attributes of the functions and
++ calculating a measure of compatibility of each routine to these
++ attributes.
++
++ It calls each \texttt{\textbackslash axiom\{}domain\texttt{\}} of \texttt{\textbackslash axiom\{}category\texttt{\}}
++ \texttt{\textbackslash axiomType\{}NumericalOptimizationCategory\texttt{\}} in turn to calculate all
++ measures and returns the best i.e. the name of the most
++ appropriate domain and any other relevant information.

measure: NumericalOptimizationProblem, RT) -> Measure
++ measure(prob, R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical optimization problem defined by
++ \texttt{\textbackslash axiom\{}prob\texttt{\}\} by checking various attributes of the functions and
++ calculating a measure of compatibility of each routine to these
++ attributes.
++
++ It calls each \texttt{\textbackslash axiom\{}domain\texttt{\}} listed in \texttt{\textbackslash axiom\{}R\texttt{\}} of \texttt{\textbackslash axiom\{}category\texttt{\}}
++ \texttt{\textbackslash axiomType\{}NumericalOptimizationCategory\texttt{\}} in turn to calculate all
++ measures and returns the best i.e. the name of the most
++ appropriate domain and any other relevant information.

optimize: NumericalOptimizationProblem, RT) -> Result
++ optimize(prob, routines) is a top level ANNA function to
++ minimize a function or a set of functions with any constraints
++ as defined within \texttt{\textbackslash axiom\{}prob\texttt{\}}.
++
++ It iterates over the \texttt{\textbackslash axiom\{}domains\texttt{\}} listed in \texttt{\textbackslash axiom\{}routines\texttt{\}} of
++ \texttt{\textbackslash axiomType\{}NumericalOptimizationCategory\texttt{\}}
++ to get the name and other relevant information of the best
++ \texttt{\textbackslash axiom\{}measure\texttt{\}} and then optimize the function on that \texttt{\textbackslash axiom\{}domain\texttt{\}}.

optimize: NumericalOptimizationProblem -> Result
++ optimize(prob) is a top level ANNA function to
++ minimize a function or a set of functions with any constraints
++ as defined within \texttt{\textbackslash axiom\{}prob\texttt{\}}.
++
++ It iterates over the \texttt{\textbackslash axiom\{}domains\texttt{\}} of
++ \axiomType{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \axiom{measure} and then optimize the function on that \axiom{domain}.

goodnessOfFit: NumericalOptimizationProblem -> Result
++ goodnessOfFit(prob) is a top level ANNA function to
++ check to goodness of fit of a least squares model
++ as defined within \axiom{prob}.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \axiom{measure} and then optimize the function on that \axiom{domain}.
++ It then calls the numerical routine \axiom{E04YCF} to get estimates
++ of the variance-covariance matrix of the regression coefficients of
++ the least-squares problem.
++
++ It thus returns both the results of the optimization and the
++ variance-covariance calculation.

optimize:(EF, LF, LOCF, LEF, LOCF) -> Result
++ optimize(f,start,lower,cons,upper) is a top level ANNA function to
++ minimize a function, \axiom{f}, of one or more variables with the
++ given constraints.
++
++ These constraints may be simple constraints on the variables
++ in which case \axiom{cons} would be an empty list and the bounds on
++ those variables defined in \axiom{lower} and \axiom{upper}, or a
++ mixture of simple, linear and non-linear constraints, where
++ \axiom{cons} contains the linear and non-linear constraints and
++ the bounds on these are added to \axiom{upper} and \axiom{lower}.
++
++ The parameter \axiom{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \axiom{measure} and then optimize the function on that \axiom{domain}.

optimize:(EF, LF, LOCF, LOCF) -> Result
++ optimize(f,start,lower,upper) is a top level ANNA function to
++ minimize a function, \axiom{f}, of one or more variables with
++ simple constraints. The bounds on
++ the variables are defined in \axiom{lower} and \axiom{upper}.
++
++ The parameter \axiom{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \axiom{domains} of
++ \texttt{NumericalOptimizationCategory} to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

\texttt{optimize:(EF,LF) -> Result}
++ \texttt{optimize}(f,start) is a top level ANNA function to
++ minimize a function, \texttt{f}, of one or more variables without
++ constraints.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

\texttt{optimize:(LEF,LF) -> Result}
++ \texttt{optimize}(lf,start) is a top level ANNA function to
++ minimize a set of functions, \texttt{lf}, of one or more variables
++ without constraints i.e. a least-squares problem.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.

\texttt{goodnessOfFit:(LEF,LF) -> Result}
++ \texttt{goodnessOfFit}(lf,start) is a top level ANNA function to
++ check to goodness of fit of a least squares model i.e. the minimization
++ of a set of functions, \texttt{lf}, of one or more variables without
++ constraints.
++
++ The parameter \texttt{start} is a list of the initial guesses of the
++ values of the variables.
++
++ It iterates over the \texttt{domains} of
++ \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.
++
++ It thus returns both the results of the optimization and the
++ variance-covariance calculation.
++ goodnessOfFit(lf,start) is a top level function to iterate over
++ the \texttt{domains} of \texttt{NumericalOptimizationCategory}
++ to get the name and other relevant information of the best
++ \texttt{measure} and then optimize the function on that \texttt{domain}.
++ It then checks the goodness of fit of the least squares model.

II ==> add

\begin{verbatim}
preAnalysis:RT \rightarrow RT
optimizeSpecific:(UNOALSA, String) \rightarrow Result
measureSpecific:(String, RT, UNOALSA) \rightarrow Measure2
changeName:(Result, String) \rightarrow Result
recoverAfterFail:(UNOALSA, RT, Measure, INT, Result) \rightarrow Record(a: Result, b: Measure)
custom:UNOALSA \rightarrow Union(DF, "failed")
optimizeConstant:DF \rightarrow Result
\end{verbatim}

import ExpertSystemToolsPackage, e04AgentsPackage, NumericalOptimizationProblem

constant(args:UNOALSA):Union(DF, "failed") ==
  args case noa =>
    Args := args.noa
    f := Args.fn
    retractIfCan(f)@Union(DoubleFloat, "failed")
  "failed"

optimizeConstant(c:DF): Result ==
  a := coerce(c)$AnyFunctions1(DF)
  text := coerce("Constant Function")$AnyFunctions1(String)
  construct([[objf@Symbol,a],[method@Symbol,text]])$Result

preAnalysis(args:UNOALSA, t:RT):RT ==
  r := selectOptimizationRoutines(t)$RT
  args case lsa =>
    selectSumOfSquaresRoutines(r)$RT
  r

zeroMeasure(m:Measure):Result ==
  a := coerce(0$F)$AnyFunctions1(F)
  text := coerce("Zero Measure")$AnyFunctions1(String)
  r := construct([[objf@Symbol,a],[method@Symbol,text]])$Result
  concat(measure2Result m, r)

measureSpecific(name:String,R:RT, args:UNOALSA): Measure2 ==
  args case noa =>
    arg:NOA := args.noa
    name = "e04dgfAnnaType" => measure(R, arg)$e04dgfAnnaType
    name = "e04fdfAnnaType" => measure(R, arg)$e04fdfAnnaType
    name = "e04gcfAnnaType" => measure(R, arg)$e04gcfAnnaType
    name = "e04jafAnnaType" => measure(R, arg)$e04jafAnnaType
name = "e04mbfAnnaType" => measure(R,arg)$e04mbfAnnaType
name = "e04nafAnnaType" => measure(R,arg)$e04nafAnnaType
name = "e04ucfAnnaType" => measure(R,arg)$e04ucfAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions
args case lsa =>
  arg2:LSA := args.lsa
  name = "e04fdfAnnaType" => measure(R,arg2)$e04fdfAnnaType
  name = "e04gcfAnnaType" => measure(R,arg2)$e04gcfAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions
error("measureSpecific","invalid argument type")$ErrorFunctions

measure(Args:NumericalOptimizationProblem,R:RT):Measure ==
  args:UNOALSA := retract(Args)$NumericalOptimizationProblem
 sofar := 0$F
  best := "none" :: String
  routs := copy R
  routs := preAnalysis(args,routs)
  empty?(routs)$RT =>
    error("measure", "no routines found")$ErrorFunctions
  rout := inspect(routs)$RT
  e := retract(rout.entry)$AnyFunctions1(Entry)
  meth := empty()$(List String)
  for i in 1..# routs repeat
    rout := extract!(routs)$RT
    e := retract(rout.entry)$AnyFunctions1(Entry)
    n := e.domainName
    if e.defaultMin > sofar then
      m := measureSpecific(n,R,args)
      if m.measure > sofar then
        sofar := m.measure
        best := n
      str := [concat(concat([string(rout.key)$Symbol,"measure: ",
          outputMeasure(m.measure)," - "],
          m.explanations)$((List String))$String]
    else
      str := [concat([string(rout.key)$Symbol
            ,", is no better than other routines"])$String]
      meth := append(meth,str)$((List String)
      [sofar,best,meth]
  measure(args:NumericalOptimizationProblem):Measure == measure(args,routines()$RT)

optimizeSpecific(args:UNOALSA,name:String):Result ==
  args case noa =>
    arg:NOA := args.noa
    name = "e04dgfAnnaType" => numericalOptimization(arg)$e04dgfAnnaType
    name = "e04fdfAnnaType" => numericalOptimization(arg)$e04fdfAnnaType
    name = "e04gcfAnnaType" => numericalOptimization(arg)$e04gcfAnnaType
    name = "e04jafAnnaType" => numericalOptimization(arg)$e04jafAnnaType
    name = "e04mbfAnnaType" => numericalOptimization(arg)$e04mbfAnnaType
name = "e04nafAnnaType" => numericalOptimization(arg)$e04nafAnnaType
name = "e04ucfAnnaType" => numericalOptimization(arg)$e04ucfAnnaType
error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
args case lsa =>
  arg2:LSA := args.lsa
  name = "e04fdfAnnaType" => numericalOptimization(arg2)$e04fdfAnnaType
  name = "e04gcfAnnaType" => numericalOptimization(arg2)$e04gcfAnnaType
  error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
  error("optimizeSpecific","invalid type name: " name)$ErrorFunctions
changeName(ans:Result,name:String):Result ==
  st:String := concat([name,"Answer"])$String
  sy:Symbol := coerce(st)$Symbol
  anyAns:Any := coerce(ans)$AnyFunctions1(Result)
  construct([[sy,anyAns]])$Result
recoverAfterFail(args:UNOALSA,routs:RT,m:Measure,iint:INT,r:Result):Record(a:Result,b:Measure) ==
  while positive?(iint) repeat
    routineName := m.name
    s := recoverAfterFail(routs,routineName(1..6),iint)$RT
    s case "failed" => iint := 0
    (s = "no action")@Boolean => iint := 0
    fl := coerce(s)$AnyFunctions1(String)
    flrec:Record(key:Symbol,entry:Any):=[failure@Symbol,fl]
    m2 := measure(args::NumericalOptimizationProblem,routs)
    zero?(m2.measure) => iint := 0
    r2:Result := optimizeSpecific(args,m2.name)
    m := m2
    insert!(flrec,r2)$Result
    r := concat(r2,changeName(r,routineName))
    iany := search(ifail@Symbol,r2)$Result
    iany case "failed" => iint := 0
    iint := retract(iany)$AnyFunctions1(INT)
  [r,m]
optimize(Args:NumericalOptimizationProblem,t:RT):Result ==
  args:UNOALSA := retract(Args)$NumericalOptimizationProblem
  routs := copy(t)$RT
  c:Union(DF,"failed") := constant(args)
  c case DF => optimizeConstant(c)
  m := measure(Args,routs)
  zero?(m.measure) => zeroMeasure m
  r := optimizeSpecific(args,n := m.name)
  iany := search(ifail@Symbol,r)$Result
  iint := 0$INT
  if (iany case Any) then
    iint := retract(iany)$AnyFunctions1(INT)
  if positive?(iint) then
    tu:Record(a:Result,b:Measure) := recoverAfterFail(args,routs,m,iint,r)
r := tu.a
m := tu.b
r := concat(measure2Result m, r)
expl := getExplanations(routs, n(1..6))$RoutinesTable
expla := coerce(expl)$AnyFunctions1(LST)
explaa:Record(key:Symbol, entry:Any) := ["explanations": Symbol, expla]
r := concat(construct([explaa]), r)
att:List String := optAttributes(args)
atta := coerce(att)$AnyFunctions1(List String)
attr:Record(key:Symbol, entry:Any) := [attributes@Symbol, atta]
insert!(attr, r)$Result

optimize(args: NumericalOptimizationProblem): Result == optimize(args, routines()$RT)

goodnessOfFit(Args: NumericalOptimizationProblem): Result ==
  r := optimize(Args)
  args1: UNOALSA := retract(Args)$NumericalOptimizationProblem
  args1 case noa => error("goodnessOfFit", "Not an appropriate problem")
  args: LSA := args1.lsa
  lf := args.lfn
  n: INT := #(variables(args))
  m: INT := # lf
  me := search(method, r)$Result
  me case "failed" => r
  meth := retract(me)$AnyFunctions1(Result)
  na := search(nameOfRoutine, meth)$Result
  na case "failed" => r
  name := retract(na)$AnyFunctions1(String)
  temp: INT := (n*(n-1)) quo 2
  ns: INT :=
      name = "e04fdfAnnaType" => 6*n+(2+n)*m+1+max(1, temp)
      8*n+(n+2)*m+temp+1+max(1, temp)
  nv: INT := ns+1
  ww := search(w, r)$Result
  ww case "failed" => r
  ws: MDF := retract(ww)$AnyFunctions1(MDF)
  fr := search(objf, r)$Result
  fr case "failed" => r
  f := retract(fr)$AnyFunctions1(DF)
  s := subMatrix(ws, 1, 1, ns, nv-1)$MDF
  v := subMatrix(ws, 1, 1, nv, ns+1-nv)$MDF
  r2 := e04ycf(0, m, n, f, s, v, -1)$NagOptimisationPackage
  concat(r, r2)

optimize(f: EF, start: LF, lower: LDCF, cons: LEF, upper: LDCF): Result ==
  args: NOA := [ef2edf(f), [f2df i for i in start], [ocf2ocdf j for j in lower],
                   [ef2edf k for k in cons], [ocf2ocdf l for l in upper]]
  optimize(args: NumericalOptimizationProblem)

optimize(f: EF, start: LF, lower: LDCF, upper: LDCF): Result ==
optimize(f,start,lower,empty()$LEF,upper)

optimize(f:EF,start:LF):Result ==
optimize(f,start,empty()$LOCF,empty()$LOCF)

optimize(1f:LEF,start:LF):Result ==
    args:LSA := [[ef2edf i for i in 1f],[f2df j for j in start]]
optimize(args::NumericalOptimizationProblem)

goodnessOfFit(1f:LEF,start:LF):Result ==
    args:LSA := [[ef2edf i for i in 1f],[f2df j for j in start]]
goodnessOfFit(args::NumericalOptimizationProblem)

package ODEPACK AnnaOrdinaryDifferentialEquationPackage

—— AnnaOrdinaryDifferentialEquationPackage.input ——

)set break resume
)sys rm -f AnnaOrdinaryDifferentialEquationPackage.output
)spool AnnaOrdinaryDifferentialEquationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AnnaOrdinaryDifferentialEquationPackage
--R
--R AnnaOrdinaryDifferentialEquationPackage is a package constructor
--R Abbreviation for AnnaOrdinaryDifferentialEquationPackage is ODEPACK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODEPACK
--R
CHAPTER 2. CHAPTER A

--R--------------------------------- Operations ---------------------------------
--R measure : NumericalODEProblem -> Record(measure: Float, name: String, explanations: List(String))
--R measure : (NumericalODEProblem, RoutinesTable) -> Record(measure: Float, name: String, explanations: List(String))
--R solve : NumericalODEProblem -> Result
--R solve : (NumericalODEProblem, RoutinesTable) -> Result
--R solve : (Vector(Expression(Float)), Float, Float, List(Float)) -> Result
--R solve : (Vector(Expression(Float)), Float, Float, List(Float), Float) -> Result
--R solve : (Vector(Expression(Float)), Float, Float, List(Float), Expression(Float), Float) -> Result
--R solve : (Vector(Expression(Float)), Float, Float, List(Float), List(Float), Float) -> Result
--R solve : (Vector(Expression(Float)), Float, Float, List(Float), Expression(Float), List(Float), Float) -> Result
--R
--E 1

)spool
)lisp (bye)

--- AnnaOrdinaryDifferentialEquationPackage.help ---
AnnaOrdinaryDifferentialEquationPackage (ODEPACK)

Exports:
  measure   solve

— package ODEPACK AnnaOrdinaryDifferentialEquationPackage —

)abbrev package ODEPACK AnnaOrdinaryDifferentialEquationPackage
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: December 1997
++ Description:
++ \texttt{AnnaOrdinaryDifferentialEquationPackage} is a \texttt{package}
++ of functions for the \texttt{category}
++ \texttt{OrdinaryDifferentialEquationsSolverCategory}
++ with \texttt{measure}, and \texttt{solve}.

AnnaOrdinaryDifferentialEquationPackage(): EE == II where

EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
FI ==> Fraction Integer
EFI ==> Expression Fraction Integer
SOCDF ==> Segment OrderedCompletion DoubleFloat
VEDF ==> Vector Expression DoubleFloat
VEF ==> Vector Expression Float
EF ==> Expression Float
LF ==> List Float
F ==> Float
VDF ==> Vector DoubleFloat
VMF ==> Vector MachineFloat
MF ==> MachineFloat
LS ==> List Symbol
ST ==> String
LST ==> List String
INT ==> Integer
RT ==> RoutinesTable
IFL ==> List(Record(ifail:Integer, instruction:String))
Entry ==> Record(chapter:String, type:String, domainName: String, _defaultMin:F,, measure:F,, failList:IFL,, explList:LST)
Measure ==> Record(measure:F,,name:String, explanations:List String)

EE ==> with
solve:(NumericalODEProblem) -> Result
++ solve(odeProblem) is a top level ANNA function to solve numerically a
++ system of ordinary differential equations i.e. equations for the
++ derivatives y[1]’..y[n]’ defined in terms of x,y[1]..y[n], together
++ with starting values for x and y[1]..y[n] (called the initial
++ conditions), a final value of x, an accuracy requirement and any
++ intermediate points at which the result is required.
++
++ It iterates over the \axiomType{domains} of
++ \axiomType{OrdinaryDifferentialEquationsSolverCategory}
++ to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiomType{measure}.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of ODE’s and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting ‘best’ routine.

solve:(NumericalODEProblem,RT) -> Result
++ solve(odeProblem,R) is a top level ANNA function to solve numerically a
++ system of ordinary differential equations i.e. equations for the
++ derivatives y[1]’..y[n]’ defined in terms of x,y[1]..y[n], together
++ with starting values for x and y[1]..y[n] (called the initial
++ conditions), a final value of x, an accuracy requirement and any
++ intermediate points at which the result is required.
++
++ It iterates over the \axiomType{domains} of
++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
++ the table of routines \axiomType{R} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiomType{measure}.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of ODE's and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting 'best' routine.
solve:(VEF,F,F,LF) -> Result
++ solve(f,xStart,xEnd,yInitial) is a top level ANNA function to solve
++ numerically a system of ordinary differential equations i.e. equations
++ for the derivatives y[1]'..y[n]' defined in terms of x,y[1]..y[n],
++ together with a starting value for x and y[1]..y[n] (called the initial
++ conditions) and a final value of x. A default value
++ is used for the accuracy requirement.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
++ the table of routines \axiom{R} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of ODE's and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting 'best' routine.
solve:(VEF,F,F,LF,EF,F) -> Result
++ solve(f,xStart,xEnd,yInitial,G,tol) is a top level ANNA function to
++ numerically a system of ordinary differential equations, \axiom{f},
++ i.e. equations for the derivatives y[1]'..y[n]' defined in terms
++ of x,y[1]..y[n] from \axiom{xStart} to \axiom{xEnd} with the initial
++ values for y[1]..y[n] \axiom{yInitial}) to a tolerance \axiom{tol}.
++
++ It iterates over the \axiom{domains} of
++ \axiomType{OrdinaryDifferentialEquationsSolverCategory} contained in
++ the table of routines \axiom{R} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \axiom{measure}.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of ODE's and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting 'best' routine.
solve:(VEF,F,F,LF,EF,F) -> Result
++ solve(f,xStart,xEnd,yInitial,G,tol) is a top level ANNA function to
solve numerically a system of ordinary differential equations,
++ \mbox{\texttt{axiom(f)}}, \text{i.e. equations for the derivatives } y[1]'..y[n] \text{ defined in}
++ terms of } x,y[1]..y[n] \text{ from } \texttt{axiom(xStart)} \text{ to } \texttt{axiom(xEnd)} \text{ with the}
++ initial values for } y[1]..y[n] \text{ (} \texttt{axiom(yInitial)} \text{) to a tolerance}
++ \texttt{axiom(tol)}. \text{The calculation will stop if the function}
++ \mbox{G(x,y[1],..,y[n]) evaluates to zero before } x = xEnd.
++
++ It iterates over the \texttt{axiom(domains)} of
++ \texttt{axiomType\{OrdinaryDifferentialEquationsSolverCategory\}} contained in
++ the table of routines \texttt{axiom\{R\}} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ \text{i.e. have the best } \texttt{axiom\{measure\}}.
++
++ The method used to perform the numerical process will be one of the
++ routines contained in the NAG numerical Library. \text{The function}
++ predicts the likely most effective routine by checking various
++ attributes of the system of ODE's and calculating a measure of
++ compatibility of each routine to these attributes.
++
++ It then calls the resulting 'best' routine.

\texttt{solve:(VEF,F,F,LF,LF,F) \rightarrow Result}

++ \texttt{solve(f,xStart,xEnd,yInitial,intVals,tol)} is a top level ANNA function
++ to solve numerically a system of ordinary differential equations,
++ \texttt{axiom(f)}, \text{i.e. equations for the derivatives } y[1]'..y[n] \text{ defined in}
++ terms of } x,y[1]..y[n] \text{ from } \texttt{axiom(xStart)} \text{ to } \texttt{axiom(xEnd)} \text{ with the}
++ initial values for } y[1]..y[n] \text{ (} \texttt{axiom(yInitial)} \text{) to a tolerance}
++ \texttt{axiom(tol)}. \text{The values of } y[1]..y[n] \text{ will be output for the values}
++ of } x \text{ in } \texttt{axiom\{intVals\}}.
++
++ It iterates over the \texttt{axiom(domains)} of
++ \texttt{axiomType\{OrdinaryDifferentialEquationsSolverCategory\}} contained in
++ the table of routines \texttt{axiom\{R\}} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ \text{i.e. have the best } \texttt{axiom\{measure\}}.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. \text{The function predicts the likely most effective routine}
++ by checking various attributes of the system of ODE's and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting 'best' routine.

\texttt{solve:(VEF,F,F,LF,EF,LF,F) \rightarrow Result}

++ \texttt{solve(f,xStart,xEnd,yInitial,G,intVals,tol)} is a top level ANNA function
++ to solve numerically a system of ordinary differential
equations, \texttt{axiom(f)}, \text{i.e. equations for the derivatives } y[1]'..y[n]'
++ defined in terms of } x,y[1]..y[n] \text{ from } \texttt{axiom(xStart)} \text{ to } \texttt{axiom(xEnd)}
++ with the initial values for } y[1]..y[n] \text{ (} \texttt{axiom(yInitial)} \text{) to a}
++ tolerance \texttt{tol}. The values of \(y_1\)\ldots\(y_n\) will be output for
++ the values of \(x\) in \texttt{intVals}. The calculation will stop if the
++ function \(G(x,y_1,\ldots,y_n)\) evaluates to zero before \(x = \texttt{xEnd}\).
++ It iterates over the \texttt{domains} of
++ \texttt{OrdinaryDifferentialEquationsSolverCategory} contained in
++ the table of routines \texttt{R} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{measure}.
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of ODE's and calculating
++ a measure of compatibility of each routine to these attributes.
++ It then calls the resulting 'best' routine.

\begin{verbatim}
solve:(VEF,F,F,LF,EF,LF,F,F) \to \text{Result}
solve(f,\texttt{xStart,xEnd,yInitial,G,intVals,epsabs,epsrel}) is a top level
ANNA function to solve numerically a system of ordinary differential
++ equations, \texttt{OrdinaryDifferentialEquationsSolverCategory}, i.e.
++ equations for the derivatives \(y_1'\ldots y_n'\) defined in terms
++ of \(x,y_1\ldots y_n\) from \texttt{xStart} to \texttt{xEnd} with the initial
++ values for \(y_1\ldots y_n\) \texttt{(yInitial)} to an absolute error
++ requirement \texttt{epsabs} and relative error \texttt{epsrel}.
++ The values of \(y_1\ldots y_n\) will be output for the values of \(x\) in
++ \texttt{intVals}. The calculation will stop if the function
++ \(G(x,y_1,\ldots,y_n)\) evaluates to zero before \(x = \texttt{xEnd}\).
++ It iterates over the \texttt{domains} of
++ \texttt{OrdinaryDifferentialEquationsSolverCategory} contained in
++ the table of routines \texttt{R} to get the name and other
++ relevant information of the the (domain of the) numerical
++ routine likely to be the most appropriate,
++ i.e. have the best \texttt{measure}.
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of ODE's and calculating
++ a measure of compatibility of each routine to these attributes.
++ It then calls the resulting 'best' routine.
\end{verbatim}

\begin{verbatim}
measure:(NumericalODEProblem) \to \text{Measure}
measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical ODE
++ problem defined by \texttt{prob}.
++ It then calls the resulting 'best' routine.
\end{verbatim}
++ It calls each axiom\{domain\} of axiom\{category\}
++ axiomType\{OrdinaryDifferentialEquationsSolverCategory\} in turn to
++ calculate all measures and returns the best i.e. the name of
++ the most appropriate domain and any other relevant information.
++ It predicts the likely most effective NAG numerical
++ Library routine to solve the input set of ODEs
++ by checking various attributes of the system of ODEs and calculating
++ a measure of compatibility of each routine to these attributes.

measure:\langle\text{NumericalODEProblem},\text{RT}\rangle \rightarrow \text{Measure}
++ measure(prob,R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical ODE
++ problem defined by axiom\{prob\}.
++
++ It calls each axiom\{domain\} listed in axiom\{R\} of axiom\{category\}
++ axiomType\{OrdinaryDifferentialEquationsSolverCategory\} in turn to
++ calculate all measures and returns the best i.e. the name of
++ the most appropriate domain and any other relevant information.
++ It predicts the likely most effective NAG numerical
++ Library routine to solve the input set of ODEs
++ by checking various attributes of the system of ODEs and calculating
++ a measure of compatibility of each routine to these attributes.

II \Rightarrow \text{add}

\text{import} \text{ODEA, NumericalODEProblem}

f2df:F \rightarrow \text{DF}
\text{ef2edf}:\text{EF} \rightarrow \text{EDF}
\text{preAnalysis}:\langle\text{ODEA},\text{RT}\rangle \rightarrow \text{RT}
\text{zeroMeasure}:\text{Measure} \rightarrow \text{Result}
\text{measureSpecific}:\langle\text{ST, RT, ODEA}\rangle \rightarrow \text{Record(measure:F, explanations:ST)}
\text{solveSpecific}:\langle\text{ODEA, ST}\rangle \rightarrow \text{Result}
\text{changeName}:\langle\text{Result, ST}\rangle \rightarrow \text{Result}
\text{recoverAfterFail}:\langle\text{ODEA, RT, Measure, Integer, Result}\rangle \rightarrow \text{Record(a:Result, b:Measure)}

\text{f2df}(f:F):\text{DF} \equiv (\text{convert}(f)@\text{DF})@F

\text{ef2edf}(f:EF):\text{EDF} \equiv \text{map}(\text{f2df}, f)@\text{ExpressionFunctions2}(F, \text{DF})

\text{preAnalysis}(\text{args:ODEA}, \text{t:RT}):\text{RT} \equiv
\quad \text{rt} := \text{selectODEIVPRoutines}(\text{t})@\text{RT}
\quad \text{if positive?(# variables(\text{args.g})) then}
\quad \quad \text{changeMeasure}(\text{rt}, \text{d02bhf@Symbol}, \text{getMeasure}(\text{rt}, \text{d02bhf@Symbol})*0.8)
\quad \text{if positive?(# \text{args.intvals}) then}
\quad \quad \text{changeMeasure}(\text{rt}, \text{d02bhf@Symbol}, \text{getMeasure}(\text{rt}, \text{d02bhf@Symbol})*0.8)
\quad \text{rt}

\text{zeroMeasure}(\text{m:Measure}):\text{Result} \equiv
\quad \text{a} := \text{coerce}(0$F)$@\text{AnyFunctions1}(F)
text := coerce("Zero Measure")$AnyFunctions1(ST)
r := construct([[result@Symbol,a],[method@Symbol,text]])$Result
concat(measure2Result m,r)$ExpertSystemToolsPackage

measureSpecific(name:ST,R:RT,ode:ODEA):Record(measure:F,explanations:ST) ==
name = "d02bbfAnnaType" => measure(R,ode)$d02bbfAnnaType
name = "d02bhfAnnaType" => measure(R,ode)$d02bhfAnnaType
name = "d02cjfAnnaType" => measure(R,ode)$d02cjfAnnaType
name = "d02ejfAnnaType" => measure(R,ode)$d02ejfAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions

measure(Ode:NumericalODEProblem,R:RT):Measure ==
ode:ODEA := retract(Ode)$NumericalODEProblem
sofar := 0$F
best := "none" :: ST
routs := copy R
routs := preAnalysis(ode,routs)
empty?(routs)$RT =>
error("measure", "no routines found")$ErrorFunctions
rout := inspect(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
meth := empty()$LST
for i in 1..# routs repeat
rout := extract!(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
n := e.domainName
if e.defaultMin > sofar then
m := measureSpecific(n,R,ode)
if m.measure > sofar then
sofar := m.measure
best := n
str:LST := [string(rout.key)$Symbol "measure: ",
outputMeasure(m.measure)$ExpertSystemToolsPackage " - "
m.explanations]
else
str := [string(rout.key)$Symbol " is no better than other routines"]
meth := append(meth,str)$LST
[sofar,best,meth]

measure(ode:NumericalODEProblem):Measure == measure(ode,routines()$RT)
solveSpecific(ode:ODEA,n:ST):Result ==
n = "d02bbfAnnaType" => ODESolve(ode)$d02bbfAnnaType
n = "d02bhfAnnaType" => ODESolve(ode)$d02bhfAnnaType
n = "d02cjfAnnaType" => ODESolve(ode)$d02cjfAnnaType
n = "d02ejfAnnaType" => ODESolve(ode)$d02ejfAnnaType
error("solveSpecific","invalid type name: " n)$ErrorFunctions

changeName(ans:Result,name:ST):Result ==
sy:Symbol := coerce(name "Answer")$Symbol
anyAns: Any := coerce(ans)$AnyFunctions1(Result)
construct([[sy, anyAns]])$Result

recoverAfterFail(ode: ODEA, routs: RT, m: Measure, iint: Integer, r: Result):
  Record(a: Result, b: Measure) ==
  while positive?(iint) repeat
    routineName := m.name
    s := recoverAfterFail(routs, routineName(1..6), iint)$RT
    s case "failed" => iint := 0
    if s = "increase tolerance" then
      ode.relerr := ode.relerr*(10.0::DF)
      ode.abserr := ode.abserr*(10.0::DF)
    if s = "decrease tolerance" then
      ode.relerr := ode.relerr/(10.0::DF)
      ode.abserr := ode.abserr/(10.0::DF)
    (a = "no action")@Boolean => iint := 0
    fl := coerce(s)$AnyFunctions1(ST)
    flrec: Record(key: Symbol, entry: Any) := [failure@Symbol, fl]
    m2 := measure(ode:: NumericalODEProblem, routs)
    zero?(m2.measure) => iint := 0
    r2: Result := solveSpecific(ode, m2.name)
    m := m2
    insert!(flrec, r2)$Result
    r := concat(r2, changeName(r, routineName))$ExpertSystemToolsPackage
    iany := search(ifail@Symbol, r)$Result
    iany case "failed" => iint := 0
    iint := retract(iany)$AnyFunctions1(Integer)
  [r, m]

solve(Ode: NumericalODEProblem, t: RT): Result ==
  ode: ODEA := retract(Ode)$NumericalODEProblem
  routs := copy(t)$RT
  m := measure(Ode, routs)
  zero?(m.measure) => zeroMeasure m
  r := solveSpecific(ode, m := m.name)
  iany := search(ifail@Symbol, r)$Result
  iint := 0$Integer
  if (iint case Any) then
    iint := retract(iany)$AnyFunctions1(Integer)
  if positive?(iint) then
    tu: Record(a: Result, b: Measure) := recoverAfterFail(ode, routs, m, iint, r)
    r := tu.a
    m := tu.b
  r := concat(measure2Result m, r)$ExpertSystemToolsPackage
  expla := getExplanations(routs, n(1..6))$RoutinesTable
  expla := coerce(expla)$AnyFunctions1(LST)
  explaa: Record(key: Symbol, entry: Any) := ["explanations":: Symbol, expla]
  r := concat(construct([explaa]), r)
  iflist := showIntensityFunctions(ode)$ODEIntensityFunctionsTable
  iflist case "failed" => r
concat(iflist2Result iflist, r)$ExpertSystemToolsPackage

solve(ode: NumericalODEProblem): Result == solve(ode, routines()$RT)

   d: ODEA := [f2df xStart, f2df xEnd, vector([ef2edf e for e in members f])$VEDF,
               [f2df i for i in yInitial], [f2df j for j in intVals],
               ef2edf G, f2df epsabs, f2df epsrel]
   solve(d:: NumericalODEProblem, routines()$RT)

   solve(f, xStart, xEnd, yInitial, G, intVals, tol, tol)

solve(f: VEF, xStart: F, xEnd: F, yInitial: LF, tol: F): Result ==
   solve(f, xStart, xEnd, yInitial, 1$EF, intVals, tol)

   solve(f, xStart, xEnd, y, G, empty()$LF, tol)

solve(f: VEF, xStart: F, xEnd: F, yInitial: LF): Result ==
   solve(f, xStart, xEnd, yInitial, 1.0e-4)

<table>
<thead>
<tr>
<th>ODEPACK.dotabb</th>
</tr>
</thead>
</table>

"ODEPACK" [color="#FF4488", href="bookvol10.4.pdf#nameddest=ODEPACK"]
"ALIST" [color="#88FF44", href="bookvol10.3.pdf#nameddest=ALIST"]
"ODEPACK" -> "ALIST"

package PDEPACK AnnaPartialDifferentialEquationPackage

| AnnaPartialDifferentialEquationPackage.input |

)set break resume
)sys rm -f AnnaPartialDifferentialEquationPackage.output
)spool AnnaPartialDifferentialEquationPackage.output
)set message test on
set message auto off
clear all

show AnnaPartialDifferentialEquationPackage

AnnaPartialDifferentialEquationPackage is a package constructor
Abbreviation for AnnaPartialDifferentialEquationPackage is PDEPACK
This constructor is exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for PDEPACK

Operations

measure : NumericalPDEProblem -> Record(measure: Float,name: String,explanations: List(String))
measure : (NumericalPDEProblem,RoutinesTable) -> Record(measure: Float,name: String,explanations: List(String))
solve : NumericalPDEProblem -> Result
solve : (NumericalPDEProblem,RoutinesTable) -> Result
solve : (Float,Float,Float,Float,NonNegativeInteger,NonNegativeInteger,List(Expression(Float)),List(List(Expression(Float))),String,DoubleFloat) -> Result
solve : (Float,Float,Float,Float,NonNegativeInteger,NonNegativeInteger,List(Expression(Float)),List(List(Expression(Float))),String) -> Result

AnnaPartialDifferentialEquationPackage.help

AnnaPartialDifferentialEquationPackage examples

AnnaPartialDifferentialEquationPackage is an uncompleted package for the interface to NAG PDE routines. It has been realised that a new approach to solving PDEs will need to be created.

See Also:
oshow AnnaPartialDifferentialEquationPackage
AnnaPartialDifferentialEquationPackage (PDEPACK)

Exports:
measure solve

--- package PDEPACK AnnaPartialDifferentialEquationPackage ---

)abbrev package PDEPACK AnnaPartialDifferentialEquationPackage
++ Author: Brian Dupee
++ Date Created: June 1996
++ Date Last Updated: December 1997
++ Description:
++ AnnaPartialDifferentialEquationPackage is an uncompleted
++ package for the interface to NAG PDE routines. It has been realised that
++ a new approach to solving PDEs will need to be created.

AnnaPartialDifferentialEquationPackage(): EE == II where
LEDF ==> List Expression DoubleFloat
EDF ==> Expression DoubleFloat
LDF ==> List DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
LEF ==> List Expression Float
EF ==> Expression Float
MEF ==> Matrix Expression Float
LF ==> List Float
F ==> Float
LS ==> List Symbol
ST ==> String
LST ==> List String
INT ==> Integer
NNI ==> NonNegativeInteger
RT ==> RoutinesTable
PDEC ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
                  dStart:MDF, dFinish:MDF)
PDEB ==> Record(pde:LEDF, constraints:List PDEC,
                f:List LEDF, st:ST, tol:DF)
CHAPTER 2. CHAPTER A

IFL ==> List(Record(ifail:INT, instruction:ST))
Measure ==> Record(measure:F, name:ST, explanations:LST)

EE ==> with
solve:(NumericalPDEProblem) -> Result
++ solve(PDEProblem) is a top level ANNA function to solve numerically a system of partial differential equations.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of PDE’s and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting ‘best’ routine.
++
++ ** At the moment, only Second Order Elliptic Partial Differential
++ Equations are solved **

solve:(NumericalPDEProblem,RT) -> Result
++ solve(PDEProblem,routines) is a top level ANNA function to solve numerically a system of partial differential equations.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of PDE’s and calculating
++ a measure of compatibility of each routine to these attributes.
++
++ It then calls the resulting ‘best’ routine.
++
++ ** At the moment, only Second Order Elliptic Partial Differential
++ Equations are solved **

solve:(F,F,F,F,NNI,NNI,LEF,List LEF,ST,DF) -> Result
++ solve(xmin,ymin,xmax,ymax,ngx,ngy,pde,bounds,st,tol) is a top level ANNA function to solve numerically a system of partial differential
++ equations. This is defined as a list of coefficients (\axiom{pde}),
++ a grid (\axiom{xmin}, \axiom{ymin}, \axiom{xmax}, \axiom{ymax},
++ \axiom{ngx}, \axiom{ngy}), the boundary values (\axiom{bounds}) and a
++ tolerance requirement (\axiom{tol}). There is also a parameter
++ (\axiom{st}) which should contain the value "elliptic" if the PDE is
++ known to be elliptic, or "unknown" if it is uncertain. This causes the
++ routine to check whether the PDE is elliptic.
++
++ The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of PDE’s and calculating
++ a measure of compatibility of each routine to these attributes.
++ It then calls the resulting 'best' routine.
++ ** At the moment, only Second Order Elliptic Partial Differential
++ Equations are solved **
solve:(F,F,F,F,NNI,NNI,LEF,List LEF,ST) -> Result
++ solve(xmin,ymin,xmax,ymax,ngx,ngy,pde,bounds,st) is a top level
++ ANNA function to solve numerically a system of partial differential
++ equations. This is defined as a list of coefficients \( \text{pde} \),
++ a grid \( (\text{xmin}, \text{ymin}, \text{xmax}, \text{ymax}, \text{ngx}, \text{ngy}) \) and the boundary values \( \text{bounds} \).
++ A default value for tolerance is used. There is also a parameter
++ \( \text{st} \) which should contain the value "elliptic" if the PDE is
++ known to be elliptic, or "unknown" if it is uncertain. This causes the
++ routine to check whether the PDE is elliptic.
++ ** The method used to perform the numerical
++ process will be one of the routines contained in the NAG numerical
++ Library. The function predicts the likely most effective routine
++ by checking various attributes of the system of PDE's and calculating
++ a measure of compatibility of each routine to these attributes.
++ ** It then calls the resulting 'best' routine.
++ ** At the moment, only Second Order Elliptic Partial Differential
++ Equations are solved **

measure:(NumericalPDEProblem) -> Measure
++ measure(prob) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical PDE
++ problem defined by \text{prob}.
++ ** It calls each \text{domain} of \text{category}
++ \text{PartialDifferentialEquationsSolverCategory} in turn to
++ calculate all measures and returns the best i.e. the name of
++ the most appropriate domain and any other relevant information.
++ It predicts the likely most effective NAG numerical
++ Library routine to solve the input set of PDEs
++ by checking various attributes of the system of PDEs and calculating
++ a measure of compatibility of each routine to these attributes.

measure:(NumericalPDEProblem,RT) -> Measure
++ measure(prob,R) is a top level ANNA function for identifying the most
++ appropriate numerical routine from those in the routines table
++ provided for solving the numerical PDE
++ problem defined by \text{prob}.
++ ** It calls each \text{domain} listed in \text{R} of \text{category}
++ \text{PartialDifferentialEquationsSolverCategory} in turn to
++ calculate all measures and returns the best i.e. the name of
++ the most appropriate domain and any other relevant information.
++ It predicts the likely most effective NAG numerical 
++ Library routine to solve the input set of PDEs 
++ by checking various attributes of the system of PDEs and calculating 
++ a measure of compatibility of each routine to these attributes.

II ==> add

import PDEB, d03AgentsPackage, ExpertSystemToolsPackage, NumericalPDEProblem

zeroMeasure:Measure -> Result
measureSpecific:(ST,RT,PDEB) -> Record(measure:F,explanations:ST)
solveSpecific:(PDEB,ST) -> Result
changeName:(Result,ST) -> Result
recoverAfterFail:(PDEB,RT,Measure,Integer,Result) -> Record(a:Result,b:Measure)

zeroMeasure(m:Measure):Result ==
a := coerce(0$F)$AnyFunctions1(F)
text := coerce("No available routine appears appropriate")$AnyFunctions1(ST)
r := construct([result@Symbol,a],[method@Symbol,text])$Result
concat(measure2Result m,r)$ExpertSystemToolsPackage

measureSpecific(name:ST,R:RT,p:PDEB):Record(measure:F,explanations:ST) ==
name = "d03eefAnnaType" => measure(R,p)$d03eefAnnaType
--name = "d03fafAnnaType" => measure(R,p)$d03fafAnnaType
error("measureSpecific","invalid type name: " name)$ErrorFunctions

measure(P:NumericalPDEProblem,R:RT):Measure ==
p:PDEB := retract(P)$NumericalPDEProblem
sofar := 0$F
best := "none" :: ST
routs := copy R
routs := selectPDERoutines(routs)$RT
empty?(routs)$RT =>
   error("measure", "no routines found")$ErrorFunctions
rout := inspect(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
meth := empty()$LST
for i in 1..# routs repeat
   rout := extract!(routs)$RT
e := retract(rout.entry)$AnyFunctions1(Entry)
n := e.domainName
if e.defaultMin > sofar then
   m := measureSpecific(n,R,p)
   if m.measure > sofar then
      sofar := m.measure
      best := n
str:LST := [string(rout.key)$Symbol "measure: "]
outputMeasure(m.measure)$ExpertSystemToolsPackage " - "
m.explanations]
else
    str := [string(rout.key)$Symbol "$ is no better than other routines"]
    meth := append(meth,str)$LST
    [sofar,best,meth]

measure(P:NumericalPDEProblem):Measure == measure(P,routines()$RT)

solveSpecific(p:PDEB,n:ST):Result ==
    n = "d03eefAnnaType" => PDESolve(p)$d03eefAnnaType
    --n = "d03fafAnnaType" => PDESolve(p)$d03fafAnnaType
    error("solveSpecific","invalid type name: " n)$ErrorFunctions

changeName(ans:Result,name:ST):Result ==
    sy:Symbol := coerce(name "Answer")$Symbol
    anyAns:Any := coerce(ans)$AnyFunctions1(Result)
    construct([[sy,anyAns]])$Result

recoverAfterFail(p:PDEB,routs:RT,m:Measure,iint:Integer,r:Result):
    Record(a:Result,b:Measure) ==
        while positive?(iint) repeat
            routineName := m.name
            s := recoverAfterFail(routs,routineName(1..6),iint)$RT
            s case "failed" => iint := 0
            (s = "no action")@Boolean => iint := 0
            fl := coerce(s)$AnyFunctions1(ST)
            flrec:Record(key:Symbol,entry:Any):=[failure@Symbol,fl]
            m2 := measure(p::NumericalPDEProblem,routs)
            zero?(m2.measure) => iint := 0
            r2:Result := solveSpecific(p,m2.name)
            m := m2
            insert!(flrec,r2)$Result
            r := concat(r2,changeName(r,routineName))$ExpertSystemToolsPackage
            iany := search(ifail@Symbol,r)$Result
            iint := retract(iany)$AnyFunctions1(Integer)
            [r,m]

solve(P:NumericalPDEProblem,t:RT):Result ==
    routs := copy(t)$RT
    m := measure(P,routs)
    p:PDEB := retract(P)$NumericalPDEProblem
    zero?(m.measure) => zeroMeasure m
    r := solveSpecific(p,n := m.name)
    iany := search(ifail@Symbol,r)$Result
    iint := 0$Integer
    if (iany case Any) then
        iint := retract(iany)$AnyFunctions1(Integer)
        if positive?(iint) then
            tu:Record(a:Result,b:Measure) := recoverAfterFail(p,routs,m,iint,r)
            r := tu.a
\[ m := \text{tu.b} \]
\[ \text{expl := getExplanations(routs,n(1..6))$RoutinesTable} \]
\[ \text{expla := coerce(expl)$AnyFunctions1(LST)} \]
\[ \text{explaa:Record(key:Symbol,entry:Any) := ["explanations":Symbol,expla]} \]
\[ r := \text{concat(construct([explaa]),r)} \]
\[ \text{concat(measure2Result m,r)$ExpertSystemToolsPackage} \]

\[
\text{solve(P:NumericalPDEProblem):Result == solve(P,routines()$RT)}
\]

\[
\text{ LEF,s:ST,to:DF):Result == }
\]
\[
\text{cx:PDEC := [f2df xmi, f2df xma, nx, 1, empty()$MDF, empty()$MDF]}
\]
\[
\text{cy:PDEC := [f2df ymi, f2df yma, ny, 1, empty()$MDF, empty()$MDF]}
\]
\[
\text{p:PDEB := [[ef2edf e for e in pe],[cx,cy],[}
\text{ [ef2edf u for u in w] for w in bo],s,to]} \]
\[
\text{solve(p::NumericalPDEProblem,routines()$RT)}
\]

\[
\text{ LEF,s:ST):Result == }
\]
\[
\text{solve(xmi,xma,ymi,yma,nx,ny,pe,bo,s,0.0001::DF)}
\]

---

---

\text{PDEPACK.dotabb ---}

"PDEPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PDEPACK"]
"TBAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TBAGG"]
"PDEPACK" -> "TBAGG"

---

package ANY1 AnyFunctions1

--- AnyFunctions1.input ---

\( \text{set break resume} \)
\( \text{sys rm -f AnyFunctions1.output} \)
\( \text{spool AnyFunctions1.output} \)
\( \text{set message test on} \)
\( \text{set message auto off} \)
\( \text{clear all} \)
\( --S 1 of 1 \)
\( \text{show AnyFunctions1} \)
AnyFunctions1 (ANY1)
Exports:
coerce retract retractable? retractIfCan

— package ANY1 AnyFunctions1 —

)abbrev package ANY1 AnyFunctions1
++ Description:
++ \spadtype{AnyFunctions1} implements several utility functions for
++ working with \spadtype{Any}. These functions are used to go back
++ and forth between objects of \spadtype{Any} and objects of other
++ types.

AnyFunctions1(S:Type): with
coerce : S -> Any
  ++ coerce(s) creates an object of \spadtype{Any} from the
  ++ object \spad{s} of type \spad{S}.
retractIfCan: Any -> Union(S, "failed")
  ++ retractIfCan(a) tries change \spad{a} into an object
  ++ of type \spad{S}. If it can, then such an object is
  ++ returned. Otherwise, "failed" is returned.
retractable?: Any -> Boolean
  ++ retractable?(a) tests if \spad{a} can be converted
  ++ into an object of type \spad{S}.
retract : Any -> S
  ++ retract(a) tries to convert \spad{a} into an object of
  ++ type \spad{S}. If possible, it returns the object.
  ++ Error: if no such retraction is possible.

== add
import NoneFunctions1(S)

Sexpr:SExpression := devaluate(S)$Lisp

retractable? a == dom(a) = Sexpr
coerce(s:S):Any == any(Sexpr, s::None)

retractIfCan a ==
  retractable? a => obj(a) pretend S
  "failed"

retract a ==
  retractable? a => obj(a) pretend S
  error "Cannot retract value."

________
package API ApplicationProgramInterface

getDomains 'Collection
  --R
  --R (1)
  --R {AssociationList, Bits, CharacterClass, ComplexDoubleFloatVector, DataList,
  --R DoubleFloatVector, EqTable, FlexibleArray, GeneralPolynomialSet,
  --R GeneralSparseTable, GeneralTriangularSet, HashTable, IndexedBits,
  --R IndexedFlexibleArray, IndexedList, IndexedOneDimensionalArray,
  --R IndexedString, IndexedVector, InnerTable, KeyedAccessFile, Library, List,
  --R ListMultiDictionary, Multiset, NeitherSparseOrDensePowerSeries,
  --R OneDimensionalArray, Point, PrimitiveArray, RegularChain,
  --R RegularTriangularSet, Result, RoutinesTable, Set, SparseTable,
  --R SquareFreeRegularTriangularSet, Stream, String, StringTable, Table,
  --R U16Vector, U32Vector, USVector, Vector, WuWenTsunTriangularSet}
  --R Type: Set(Symbol)
--E 1

difference(getDomains 'IndexedAggregate,getDomains 'Collection)
  --R
  --R (2)
  --R {DirectProduct, DirectProductMatrixModule, DirectProductModule,
  --R HomogeneousDirectProduct, OrderedDirectProduct,
  --R SplitHomogeneousDirectProduct}
  --R Type: Set(Symbol)
An alphabetical listing of contributors to AXIOM:

- Cyril Alberga  Roy Adler
- Richard Anderson  George Andrews
- Henry Baker  Martin Baker
- Yuri Baransky  David R. Barton
- Gerald Baumgartner  Gilbert Baumsagl
- Nelson H. F. Beebe  Jay Belanger
- Fred Blair  Vladimir Bondarenko
- Raoul Bourquin  Alexandre Bouyer
- Peter A. Broadbery  Martin Brock
- Stephen Buchwald  Florian Bundschuh
- William Burge  Ralph Byers
- Robert Caviness  Bruce Char
- Tzu-Yi Chen  Cheekai Chin
- Gregory V. Chudnovsky  Mark Clements
- Jia Zhao Cong  Josh Cohen
- Don Coppersmith  George Corliss
- Gary Cornell  Meino Cramer
- David Cyganski  Nathaniel Daly
- Timothy Daly Jr.  James H. Davenport
- James Demmel  Didier Deshommes
- Jack Dongarra  Jean Della Dora
- Claire DiCrescendo  Sam Dooley
- Iain Duff  Lee Duhem
- Brian Dupee  Dominique Duval
- Heow Eide-Goodman  Lars Erickson
- Bertfried Fauser  Stuart Feldman
- Brian Ford  Albrecht Fortenbacher
- Constantine Frangos  Timothy Freeman
- Marc Gaetano  Rudiger Gebauer
- Kathy Gerber  Patricia Gianni
- Samantha Goldrich  Holger Gollan
- Laureano Gonzalez-Vega  Stephen Gortler
- Matt Grayson  Klaus Ebbe Grue
- Vladimir Grinberg  Oswald Gschmitzer
- Jocelyn Guidry  Gaetan Hache
- Satoshi Hamaguchi  Sven Hammarling
- Richard Hanson  Richard Harke
- Vilya Harvey  Martin Hassner
- Dan Hatton  Waldek Hebisch
- Ralf Hemmecke  Henderson
- Roger House  Gernot Hueber
- Alejandro Jakubi  Richard Jenks
- Kyriakos Kalorkoti  Kai Kaminski
- Wilfrid Kendall  Tony Kennedy
CHAPTER 2. CHAPTER A

---E 4

---S 5 of 9
)show API
--R ApplicationProgramInterface is a package constructor
--R Abbreviation for ApplicationProgramInterface is API
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for API
--R
--R------------------------------- Operations --------------------------------
--R credits : () -> Void getAncestors : Symbol -> Set(Symbol)
--R getDomains : Symbol -> Set(Symbol) summary : () -> Void
--R reportInstantiations : Boolean -> Void
--R
--E 5

---S 6 of 9
getAncestors 'IndexedAggregate
--R
--R
--R (5)
--R {Aggregate, BasicType, CoercibleTo, Eltable, EltableAggregate, Evalable,
--R HomogeneousAggregate, InnerEvalable, SetCategory, Type}
--R
--E 6

---S 7 of 9
reportInstantiations(true)
--R
--R
--R# instantiated/# dropped/domain name
--R------------------------------------
--RTotals: 0 instantiated
--R 0 inside coerceInteractive
--R 0 inside canCoerceFrom
--R 0 inside evalMmCond
--R 0 reinstantiated
--R 0 dropped
--R 0 distinct domains instantiated/dropped
--E 7

---S 8 of 9
1
--R
--R
--R (7) 1
--R
--R# instantiated/# dropped/domain name
--R------------------------------------
--R------------------------------------
--R1 0 PositiveInteger
This package contains useful functions that expose Axiom system internals.

The ApplicationProgramInterface exposes Axiom internal functions which might be useful for understanding, debugging, or creating tools.

The getDomains function takes the name of a category and returns a set of domains which inherit from that category:

getDomains 'Collection

{AssociationList, Bits, CharacterClass, ComplexDoubleFloatVector, DataList, DoubleFloatVector, EqTable, FlexibleArray, GeneralPolynomialSet,
This can be used to form the set-difference of two categories:

\[
\text{difference(getDomains 'IndexedAggregate, getDomains 'Collection)}
\]

\[
\{\text{DirectProduct, DirectProductMatrixModule, DirectProductModule, HomogeneousDirectProduct, OrderedDirectProduct, SplitHomogeneousDirectProduct}\}
\]

The credits function prints a list of the people who have contributed to the development of Axiom. This is equivalent to the )credits command.

The summary function prints a short list of useful console commands.

The getAncestors function takes the name of a domain or category and returns a list of categories from which it inherits.

\[
\text{getAncestors 'IndexedAggregate}
\]

\[
\{\text{Aggregate, BasicType, CoercibleTo, Eltable, EltableAggregate, Evalable, HomogeneousAggregate, InnerEvalable, SetCategory, Type}\}
\]

The reportInstantiations function information about what domains are instantiated by an expression. In a clean Axiom you'll see

\[
(1) \rightarrow \text{reportInstantiations(true)}
\]

# instantiated/# dropped/domain name

```
# instantiated/# dropped/domain name
-----------------------------
 1 0 Void
 1 0 String
 1 0 SingleInteger
 1 0 PrimitiveArray(OutputForm)
 1 0 OutputForm
 1 0 List(OutputForm)
 1 0 Integer
 1 0 Character

Totals: 8 instantiated
 0 inside coerceInteractive
```
Which shows that 8 domains were instantiated.
If a new domain is requested, more will be instantiated.

(2) -> 1

(2) 1

# instantiated/# dropped/domain name
---------------------------------------------
1 0 PositiveInteger
1 0 NonNegativeInteger

Totals: 2 instantiated
0 inside canCoerceFrom
0 inside evalMmCond
0 reinstantiated
0 dropped
2 distinct domains instantiated/dropped

However, this happens only once. Invoking already instantiated domains does nothing since they already exist.

(3) -> 1

(3) 1

# instantiated/# dropped/domain name
---------------------------------------------

Totals: 0 instantiated
0 inside coerceInteractive
0 inside canCoerceFrom
0 inside evalMmCond
0 reinstantiated
0 dropped
0 distinct domains instantiated/dropped

We call the function with false to turn off this information.

(4) -> reportInstantiations(false)

Type: Void
ApplicationProgramInterface (API)

Exports:

— package API ApplicationProgramInterface —

)abbrev package API ApplicationProgramInterface
++ Author: Timothy Daly, Martin Rubey
++ Date Created: 3 March 2009
++ Date Last Updated: 24 March 2012
++ Description:
++ This package contains useful functions that expose Axiom system internals

ApplicationProgramInterface(): Exports == Implementation where
Exports ==> with
  getDomains : Symbol -> Set Symbol
  ++ The getDomains(s) takes a category and returns the list of domains
  ++ that have that category
  ++
  ++X getDomains 'IndexedAggregate
  getAncestors : Symbol -> Set Symbol
  ++ The getAncestor(s) takes a category and returns the list of domains
  ++ that have that category as ancestors
  ++
  ++X getAncestors 'IndexedAggregate
  credits : () -> Void
  ++ credits() prints a list of people who contributed to Axiom
  ++
  ++X credits()
  summary : () -> Void
  ++ summary() prints a short list of useful console commands
  ++
  ++X summary()
reportInstantiations: Boolean -> Void
  ++ A debugging tool to show instantiation information
  ++
  ++X reportInstantiations(true)
  ++X 1
  ++X reportInstantiations(false)

Implementation ==> add
getDomains(cat:Symbol):Set(Symbol) ==
  set [symbol car first destruct a _
    for a in (destruct domainsOf(cat,NIL$Lisp)$Lisp)::List(SExpression)]

getAncestors(cat:Symbol):Set(Symbol) ==
  set [symbol car first destruct a _
    for a in (destruct ancestorsOf(cat,NIL$Lisp)$Lisp)::List(SExpression)]

credits() == ( credits()$Lisp ; void() )

summary() == ( summary()$Lisp ; void() )

reportInstantiations(b:Boolean): Void ==
  REPORTINSTANTIATIONS(b)$Lisp
  void

/package APPRULE APPLYRULES

— API.dotabb —

"API" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPRULE"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"API" -> "ORDSET"

/package APPRULE ApplyRules

— ApplyRules.input —

)set break resume
/sys rm -f ApplyRules.output
/spool ApplyRules.output
)set message test on
)set message auto off
)clear all
---S 1 of 1
)show ApplyRules
--R
--R ApplyRules(Base: SetCategory,R: Join(Ring,PatternMatchable(Base),OrderedSet,ConvertibleTo(Pattern(Base))),F: Join(FunctionSpace(R),PatternMatchable(Base),ConvertibleTo(Pattern(Base)))) is a package constructor
--R Abbreviation for ApplyRules is APPRULE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for APPRULE
--R
--R------------------------------- Operations --------------------------------
--R localUnquote : (F,List(Symbol)) -> F
--R applyRules : (List(RewriteRule(Base,R,F)),F) -> F
--R applyRules : (List(RewriteRule(Base,R,F)),F,PositiveInteger) -> F
--R
--E 1

)spool
)lisp (bye)

——

— ApplyRules.help —

====================================================================
ApplyRules examples
====================================================================

This package apply rewrite rules to expressions, calling the pattern matcher.

See Also:

- )show ApplyRules

——
ApplyRules (APPRULE)

Exports:
  applyRules  localUnquote

— package APPRULE ApplyRules —

)abbrev package APPRULE ApplyRules
++ Applications of rules to expressions
++ Author: Manuel Bronstein
++ Date Created: 20 Mar 1990
++ Date Last Updated: 5 Jul 1990
++ Description:
++ This package apply rewrite rules to expressions, calling
++ the pattern matcher.

ApplyRules(Base, R, F): Exports == Implementation where
  Base : SetCategory
  R     : Join(Ring, PatternMatchable Base, OrderedSet,
                   ConvertibleTo Pattern Base)
  F     : Join(FunctionSpace R, PatternMatchable Base,
                   ConvertibleTo Pattern Base)

P ==> Pattern Base
PR ==> PatternMatchResult(Base, F)
RR ==> RewriteRule(Base, R, F)
K ==> Kernel F

Exports ==> with
  applyRules : (List RR, F) -> F
    ++ applyRules([r1,...,rn], expr) applies the rules
    ++ r1,...,rn to f an unlimited number of times, i.e. until
    ++ none of r1,...,rn is applicable to the expression.
  applyRules : (List RR, F, PositiveInteger) -> F
    ++ applyRules([r1,...,rn], expr, n) applies the rules
    ++ r1,...,rn to f a most n times.
  localUnquote: (F, List Symbol) -> F
++ localUnquote(f,ls) is a local function.

Implementation ==> add
import PatternFunctions1(Base, F)

splitRules: List RR -> Record(lker: List K, lval: List F, rl: List RR)
localApply : (List K, List F, List RR, F, PositiveInteger) -> F
rewrite : (F, PR, List Symbol) -> F
app : (List RR, F) -> F
applist : (List RR, List F) -> List F
isit : (F, P) -> PR
isitwithpred: (F, P, List P, List PR) -> PR

applist(lrule, arglist) == [app(lrule, arg) for arg in arglist]

splitRules l ==
ncr := empty()$List(RR)
lk := empty()$List(K)
lv := empty()$List(F)
for r in l repeat
  if (u := retractIfCan(r)@Union(Equation F, "failed"))
    case "failed" then ncr := concat(r, ncr)
    else
      lk := concat(retract(lhs(u::Equation F))@K, lk)
      lv := concat(rhs(u::Equation F), lv)
[lk, lv, ncr]

applyRules(l, s) ==
  rec := splitRules l
  repeat
    (new:= localApply(rec.lker,rec.lval,rec.rl,s,1)) = s => return s
    s := new
applyRules(l, s, n) ==
  rec := splitRules l
  localApply(rec.lker, rec.lval, rec.rl, s, n)

localApply(lk, lv, lrule, subject, n) ==
  for i in 1..n repeat
    for k in lk for v in lv repeat
      subject := eval(subject, k, v)
    subject := app(lrule, subject)

rewrite(f, res, l) ==
  lk := empty()$List(K)
  lv := empty()$List(F)
  for rec in destruct res repeat
    lk := concat(kernel(rec.key), lk)
    lv := concat(rec.entry, lv)
localUnquote(eval(f, lk, lv), l)

if R has ConvertibleTo InputForm then
  localUnquote(f, l) ==
  empty? l => f
  eval(f, l)
else
  localUnquote(f, l) == f

isitwithpred(subject, pat, vars, bad) ==
  failed?(u := patternMatch(subject, pat, new()$PR)) => u
  satisfy?(u, pat)::Boolean => u
  member?(u, bad) => failed()
  for v in vars repeat addBadValue(v, getMatch(v, u)::F)
  isitwithpred(subject, pat, vars, concat(u, bad))

isit(subject, pat) ==
  hasTopPredicate? pat =>
  for v in (l := variables pat) repeat resetBadValues v
  isitwithpred(subject, pat, l, empty())
  patternMatch(subject, pat, new()$PR)

app(lrule, subject) ==
  for r in lrule repeat
    not failed?(u := isit(subject, pattern r)) =>
    return rewrite(rhs r, u, quotedOperators r)
  (k := retractIfCan(subject)@Union(K, "failed")) case K =>
  operator(k::K) applist(lrule, argument(k::K))
  (l := isPlus subject) case List(F) => +/applist(lrule,l::List(F))
  (l := isTimes subject) case List(F) => */applist(lrule,l::List(F))
  (e := isPower subject) case Record(val:F, exponent:Integer) =>
    ee := e::Record(val:F, exponent:Integer)
    f := app(lrule, ee.val)
    positive?(ee.exponent) => f ** (ee.exponent)::NonNegativeInteger
    recip(f)::F ** (-ee.exponent)::NonNegativeInteger
    subject
package APPLYORE ApplyUnivariateSkewPolynomial

— ApplyUnivariateSkewPolynomial.input —

)set break resume
)sys rm -f ApplyUnivariateSkewPolynomial.output
)spool ApplyUnivariateSkewPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ApplyUnivariateSkewPolynomial
--R
--R ApplyUnivariateSkewPolynomial(R: Ring,M: LeftModule(R),P: UnivariateSkewPolynomialCategory(R)) is a package constructor
--R Abbreviation for ApplyUnivariateSkewPolynomial is APPLYORE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for APPLYORE
--R
--R-------------------------------- Operations --------------------------------
--R apply : (P,(M -> M),M) -> M
--R
--E 1

)spool
)lisp (bye)

———

— ApplyUnivariateSkewPolynomial.help —

========================================================
ApplyUnivariateSkewPolynomial examples
========================================================

ApplyUnivariateSkewPolynomial (internal) allows univariate skew polynomials to be applied to appropriate modules.

See Also:
  o )show ApplyUnivariateSkewPolynomial

———
ApplyUnivariateSkewPolynomial (APPLYORE)

Exports:
apply

--- package APPLYORE ApplyUnivariateSkewPolynomial ---

)abbrev package APPLYORE ApplyUnivariateSkewPolynomial
++ Author: Manuel Bronstein
++ Date Created: 7 December 1993
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{ApplyUnivariateSkewPolynomial} (internal) allows univariate
++ skew polynomials to be applied to appropriate modules.

ApplyUnivariateSkewPolynomial(R:Ring, M: LeftModule R,
  P: UnivariateSkewPolynomialCategory R): with
  apply: (P, M -> M, M) -> M
  ++ apply(p, f, m) returns \spad{p(m)} where the action is given
  ++ by \spad{x m = f(m)}.
  ++ \spad{f} must be an R-pseudo linear map on M.
++ add
  apply(p, f, m) ==
  w:M := 0
  mn:M := m
  for i in 0..degree p repeat
    w := w + coefficient(p, i) * mn
    mn := f mn
  w

--- APPLYORE.dotabb ---

"APPLYORE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=APPLYORE"]
package ASSOCEQ AssociatedEquations

AssociatedEquations examples

AssociatedEquations provides functions to compute the associated equations needed for factoring operators
AssociatedEquations (ASSOCEQ)

Exports:
associatedEquations  associatedSystem  uncouplingMatrices

--- package ASSOCEQ AssociatedEquations ---

)abbrev package ASSOCEQ AssociatedEquations
++ Author: Manuel Bronstein
++ Date Created: 10 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{AssociatedEquations} provides functions to compute the
++ associated equations needed for factoring operators

AssociatedEquations(R, L):Exports == Implementation where
  R: IntegralDomain
  L: LinearOrdinaryDifferentialOperatorCategory R

PI  ==> PositiveInteger
N   ==> NonNegativeInteger
MAT ==> Matrix R
REC ==> Record(minor: List PI, eq: L, minors: List List PI, ops: List L)

Exports == with
  associatedSystem: (L, PI) -> Record(mat: MAT, vec: Vector List PI)
    ++ associatedSystem(op, m) returns \spad{[M, w]} such that the
    ++ m-th associated equation system to L is \spad{w' = M \cdot w}.  

See Also:
o )show AssociatedEquations
uncouplingMatrices: MAT -> Vector MAT
++ uncouplingMatrices(M) returns \spad{[A_1, ..., A_n]} such that if
++ \spad{y = [y_1, ..., y_n]} is a solution of \spad{y' = M y}, then
++ \spad{[y_j', y_j'', ..., y_j^{(n)}] = A_j y} for all j's.

if R has Field then
  associatedEquations: (L, PI) -> REC
  ++ associatedEquations(op, m) returns \spad{[w, eq, lw, lop]} such that \spad{eq(w) = 0} where w is the given minor, and
  ++ \spad{lw_i = lop_i(w)} for all the other minors.

Implementation ==> add
makeMatrix: (Vector MAT, N) -> MAT

diff:L := D()
makeMatrix(v, n) == matrix [parts row(v.i, n) for i in 1..#v]

associatedSystem(op, m) ==
  eq: Vector R
  S := SetOfMIntegersInOneToN(m, n := degree(op)::PI)
  w := enumerate()$S
  s := size()$S
  ww: Vector List PI := new(s, empty())
  M: MAT := new(s, s, 0)
  m1 := (m::Integer - 1)::PI
  an := leadingCoefficient op
  a: Vector(R) := [- (coefficient(op, j) exquo an)::R for j in 0..n - 1]
  for i in 1..s repeat
    eq := new(s, 0)
    wi := w.i
    ww.i := elements wi
    for k in 1..m1 repeat
      u := incrementKthElement(wi, k::PI)$S
      if member?(n, wi) then
        for j in 1..n | a.j ^= 0 repeat
          u := replaceKthElement(wi, m, j::PI)
          if u case S then
            eq(lookup(u::S)) := (odd? delta(wi, m, j::PI) => -a.j; a.j)
          else
            u := incrementKthElement(wi, m)$S
            if u case S then
              eq(lookup(u::S)) := 1
        setRow_!(M, i, eq)
    [M, ww]

uncouplingMatrices m ==
  n := nrows m
  v: Vector MAT := new(n, zero(1, 0)$MAT)
  v.1 := mi := m
  for i in 2..n repeat v.i := mi := map((z1:R):R +-> diff z1, mi) + mi * m
[\text{makeMatrix}(v, i) \text{ for } i \text{ in } 1..n]

\text{if } R \text{ has Field then}
\quad \text{import } \text{PrecomputedAssociatedEquations}(R, L)

\text{makeop: } \text{Vector } R \rightarrow L
\text{makeeq: } (\text{Vector List } PI, \text{MAT}, N, N) \rightarrow \text{REC}
\text{computeIt: } (L, PI, N) \rightarrow \text{REC}

\text{makeeq}(v, m, i, n) =
\quad [v.i, \text{makeop row}(m, i) - 1, [v.j \text{ for } j \text{ in } 1..n | j \neq i],
\quad \text{[makeop row}(m, j) \text{ for } j \text{ in } 1..n | j \neq i)]

\text{associatedEquations}(op, m) =
\quad (u := \text{firstUncouplingMatrix}(op, m)) \text{ case } "failed" \Rightarrow \text{computeIt}(op,m,1)
\quad (v := \text{inverse}(u::\text{MAT})) \text{ case } "failed" \Rightarrow \text{computeIt}(op,m,2)
\quad S := \text{SetOfMIntegersInOneToN}(m, \text{degree}(op)::PI)
\quad w := \text{enumerate}()\$S
\quad s := \text{size}()\$S
\quad ww: \text{Vector List } PI := \text{new}(s, \text{empty}())
\quad \text{for } i \text{ in } 1..s \text{ repeat } ww.i := \text{elements}(w.i)
\quad \text{makeeq}(ww, v::\text{MAT}, 1, s)

\text{computeIt}(op, m, k) =
\quad \text{rec := associatedSystem}(op, m)
\quad a := \text{uncouplingMatrices}(\text{rec.mat})
\quad n := \#a
\quad \text{for } i \text{ in } k..n \text{ repeat}
\quad \quad (u := \text{inverse}(a.i)) \text{ case } \text{MAT} \Rightarrow \text{return makeeq}(\text{rec.vec}, u::\text{MAT}, i, n)
\quad \text{error } "\text{associatedEquations: full degenerate case}"

\text{makeop } v =
\quad \text{op:L := 0}
\quad \text{for } i \text{ in } 1..\#v \text{ repeat op := op + monomial}(v \text{ i, i})
\quad \text{op}
package PMPRED AttachPredicates

--- AttachPredicates.input ---

)set break resume
)sys rm -f AttachPredicates.output
)spool AttachPredicates.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AttachPredicates
--R
--R AttachPredicates(D: Type) is a package constructor
--R Abbreviation for AttachPredicates is PMPRED
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMPRED
--R
--R----------------------------- Operations -----------------------------
--R suchThat : (Symbol,(D -> Boolean)) -> Expression(Integer)
--R suchThat : (Symbol,List((D -> Boolean))) -> Expression(Integer)
--R
--E 1

)spool
)lisp (bye)

---------

--- AttachPredicates.help ---

====================================================================
AttachPredicates examples
====================================================================

Attaching predicates to symbols for pattern matching.

See Also:
  o )show AttachPredicates

---------
AttachPredicates (PMPRED)

Exports:
suchThat

--- package PMPRED AttachPredicates ---

)abbrev package PMPRED AttachPredicates
++ Predicates for pattern-matching
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching predicates to symbols for pattern matching.

AttachPredicates(D:Type): Exports == Implementation where
  FE ==> Expression Integer

Exports ==> with
  suchThat: (Symbol, D -> Boolean) -> FE
    ++ suchThat(x, foo) attaches the predicate foo to x.
  suchThat: (Symbol, List(D -> Boolean)) -> FE
    ++ suchThat(x, [f1, f2, ..., fn]) attaches the predicate
    ++ f1 and f2 and ... and fn to x.

Implementation ==> add
  import FunctionSpaceAttachPredicates(Integer, FE, D)

  suchThat(p:Symbol, f:D -> Boolean) == suchThat(p::FE, f)
  suchThat(p:Symbol, l:List(D -> Boolean)) == suchThat(p::FE, l)

--- PMPRED.dotabb ---
package AXSERV AxiomServer

--- AxiomServer.input ---

)set break resume
)sys rm -f AxiomServer.output
)spool AxiomServer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show AxiomServer

--R
--R AxiomServer is a package constructor
--R Abbreviation for AxiomServer is AXSERV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for AXSERV
--R
--R------------------------------------- Operations -------------------------------------
--R multiServ : SExpression -> Void
--R axServer : (Integer,(SExpression -> Void)) -> Void
--R getDatabase : (String,String) -> String
--R
--E 1

)spool
)lisp (bye)

--- AxiomServer.help ---

====================================================================
AxiomServer examples
====================================================================
This package provides a function to support a web server for the new Axiom Browser functions.

See Also:
- `)show AxiomServer`

---

**AxiomServer (AXSERV)**

Exports:
- `multiServ`, `axServer`, `getDatabase`

--- package AXSERV AxiomServer ---

```lisp
)abbrev package AXSERV AxiomServer
++ Author: Arthur Ralfe, Timothy Daly, Alfredo Portes
++ Description:
++ This package provides a function to support a web server for the new Axiom Browser functions.

AxiomServer: public == private where

public == with

  `axServer`: (Integer, SExpression -> Void) -> Void
  `multiServ`: SExpression -> Void
  `getDatabase`: (String, String) -> String

private == add

  `getFile`: (SExpression, String) -> Void
```
getCommand: (SExpression, String) -> Void
getDescription: String -> String
getInterp: (SExpression, String) -> Void
getLisp: (SExpression, String) -> Void
getShow: (SExpression, String) -> Void
lastStep: () -> String
lastType: () -> String
formatMessages: String -> String
makeErrorPage: String -> String
getSourceFile: (String, String, String) -> String
makeDBPage: String -> String
getContentType: String -> String
readTheFile: SExpression -> String
outputToSocket: (SExpression, String, String) -> Void

getDatabase(constructor: String, key: String): String ==
  answer := string GETDATABASE(INTERN$Lisp constructor, INTERN$Lisp key)$Lisp
  answer

The axServer function handles the socket connection on the given port. When it gets an input
on the socket it calls the server function on the socket input.

— package AXSERV AxiomServer —

axServer(port: Integer, serverfunc: SExpression -> Void): Void ==
  WriteLine$Lisp "listening on port 8085"
  s := SiSock(port, serverfunc)$Lisp
  -- To listen for just one connection and then close the socket
  -- uncomment i := 0.
  i: Integer := 1
  while (i > 0) repeat
    if not null?(SiListen(s)$Lisp)$SExpression then
      w := SiAccept(s)$Lisp$SExpression
      serverfunc(w)
    --
      i := 0

The multiServ function parses the socket input. It expects either a GET or POST request.
A GET request fetches a new page, calling “getFile”. A POST request starts with

- “command=” which expects axiom interpreter commands. When this is recognized we
call the “getCommand” function.

- “lispcall=” which expects lisp interpreter input When this is recognized we call the
  “getLisp” function.
--- package AXSERV AxiomServer ---

multiServ(s:SExpression):Void ==
  -- WriteLine("multiServ begin")$Lisp
  headers:String := ""
  char:String
  -- read in the http headers
  while (char :=
    STRING(READ_-CHAR_-NO_-HANG(s,NIL$Lisp,'EOF)$Lisp)$Lisp) ^= "EOF"_
  repeat
    headers := concat [headers,char]
  -- sayTeX$Lisp headers
  StringMatch("([ "]*)", headers)$Lisp
  u:UniversalSegment(Integer)
  u := segment(MatchBeginning(1)$Lisp+1,
    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
  reqtype:String := headers.u
  -- sayTeX$Lisp concat ['request type: ",reqtype]
  if reqtype = "GET" then
    StringMatch("GET ([ "]*)",headers)$Lisp
    u:UniversalSegment(Integer)
    u := segment(MatchBeginning(1)$Lisp+1,
      MatchEnd(1)$Lisp)$UniversalSegment(Integer)
    getFile(s,headers.u)
  if reqtype = "POST" and StringMatch("command=(.*)",headers)$Lisp > 0
    then
      u:UniversalSegment(Integer)
      u := segment(MatchBeginning(1)$Lisp+1,
        MatchEnd(1)$Lisp)$UniversalSegment(Integer)
      getCommand(s,headers.u)
  if reqtype = "POST" and StringMatch("interpcall=(.*)",headers)$Lisp > 0
    then
      u:UniversalSegment(Integer)
      u := segment(MatchBeginning(1)$Lisp+1,
        MatchEnd(1)$Lisp)$UniversalSegment(Integer)
      getInterp(s,headers.u)
  if reqtype = "POST" and StringMatch("lispcall=(.*)",headers)$Lisp > 0
    then
      u:UniversalSegment(Integer)
      u := segment(MatchBeginning(1)$Lisp+1,
        MatchEnd(1)$Lisp)$UniversalSegment(Integer)
      getLisp(s,headers.u)
  if reqtype = "POST" and StringMatch("showcall=(.*)",headers)$Lisp > 0
    then
      u:UniversalSegment(Integer)
      u := segment(MatchBeginning(1)$Lisp+1,
        MatchEnd(1)$Lisp)$UniversalSegment(Integer)
      getShow(s,headers.u)
-- WriteLine("multiServ end")$Lisp
-- WriteLine(""")$Lisp

getFile

Given a socket and the URL of the file we create an input stream that contains the file. If the filename contains a question mark then we need to parse the parameters and dynamically construct the file contents.

-- package AXSERV AxiomServer --

getFile(s: SExpression, pathvar: String): Void ==
-- WriteLine(""")$Lisp
WriteLine$Lisp concat ["getFile: ", pathvar]
params:=split(pathvar, char ")?
if #params = 1
then if not null? PATHNAME_-NAME(PATHNAME(pathvar)$Lisp)$Lisp
then
  contentType:String := getContentType(pathvar)
  q:=Open(pathvar)$Lisp
  if null? q
  then
    q := MAKE_-STRING_-INPUT_-STREAM(_
             makeErrorPage("File doesn't exist")(p$Lisp)
  else
    q:=MAKE_-STRING_-INPUT_-STREAM(_
             makeErrorPage("Problem with file path")$Lisp
  else
    q:=MAKE_-STRING_-INPUT_-STREAM(makeDBPage(pathvar))$Lisp
outputToSocket(s,readTheFile(q),contentType)

makeErrorPage

-- package AXSERV AxiomServer --

makeErrorPage(msg:String): String ==
page:String:="<!DOCTYPE html PUBLIC "
page:=page "-//W3C//DTD XHTML 1.0 Strict//EN_"
page:=page "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
page:=page "<html xmlns="http://www.w3.org/1999/xhtml">
page:=page "<head><title>Error</title></head><body>" msg "" msg "" msg "" msg ""
getDescription

We need to fish around in the data structure to return the piece of documentation for the
domain. We have to call the lisp version of GETDATABASE because the version above
returns a string object. The string object is missing quotes and cannot be properly read. So
we need to get the lisp object and work with it in native form first.
The doc string also contains spad markup which we need to replace with html.

---
package AXSERV AxiomServer ---

getDescription(dom: String): String ==
d := CADR(CADAR(GETDATABASE(INTERN(dom)$Lisp,'DOCUMENTATION)$Lisp)$Lisp)$Lisp
string d

---

getSourceFile

During build we construct a hash table that takes the chunk name as the key and returns
the filename. We reconstruct the chunk name here and do a lookup for the source file.

---
package AXSERV AxiomServer ---

ggetSourceFile(constructorkind: String, _
abbreviation: String, _
dom: String): String ==
sourcekey:="@<< " constructorkind " " abbreviation " " dom ">>"
-- WriteLine(sourcekey)$Lisp
sourcefile:=lowerCase last split(getDatabase(dom,"SOURCEFILE"),char "/")
sourcefile:=sourcefile ".pamphlet"

---

makeDBPage

---
package AXSERV AxiomServer ---

makeDBPage(pathvar: String): String ==
params: List(String) := split(pathvar, char "?")
for i in 1..#params repeat WriteLine$Lisp concat ["params: ",params.i]
pathparts: List(String):=split(params.1,char "/")
for i in 1..#pathparts repeat
    WriteLine$Lisp concat ["pathparts: ", pathparts.i]
    pagename:=last pathparts
    WriteLine$Lisp concat ["pagename: ", pagename]
    cmd:=first split(pagename, char ".")
    WriteLine$Lisp concat ["cmd: ", cmd]
    args:List(String):=split(params.2, char "&")
for i in 1..#args repeat WriteLine$Lisp concat ["args: ", args.i]
page:='"<!DOCTYPE html PUBLIC 
    "-//W3C//DTD XHTML 1.0 Strict//EN",
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
    "html xmlns="http://www.w3.org/1999/xhtml">
    "head">
    "meta http-equiv="Content-Type" content="text/html">
    "charset="us-ascii">
    "title">
    "cmd " args.1 "</title></head>
    "<style> html { background-color: #ECEA81; } </style>
    "<body>
    cmd = "db" =>
    dom:=args.1
domi:=INTERN(dom)$Lisp
-- category, domain, or package?
constructorkind:=getDatabase(dom,"CONSTRUCTORKIND")
abbreviation:=getDatabase(dom, "ABBREVIATION")
sourcefile:=getDatabase(dom, "SOURCEFILE")
constructorkind.1:=toUpperCase constructorkind.1
description:=getDescription(dom)
page:="<div align="center">
    "<img align="middle" src="doctitle.png" /></div><hr/>
    "<table>
    "<tr><td valign="top">Description: </td>
    "<td>
    "<table>
    "<tr><td valign="top">Description: </td>
    "<td>
    "<table>
    "<tr><td valign="top">Abbreviation: </td>
    "<td>
    "<table>
    "<tr><td valign="top">Source File: </td>
    "<td>
    "<table>
    "<tr>
    "<td>
    "<a href="?" dom ">Ancestors</a>
    "</td>
    "<td>
    "<a href="?" dom ">Dependents</a>
    "</td>
    "<td>
    "<a href="?" dom ">Exports</a>
    "</td>
    "<td>
    "<a href="?" dom ">Parents</a>"
<table>
<thead>
<tr>
<th>Description:</th>
<th>Attributes</th>
<th>Examples</th>
<th>Operations</th>
<th>Search Path</th>
<th>Uses</th>
</tr>
</thead>
</table>

```lisp
--- category, domain, or package?
dom = getDatabase(dom, "CONSTRUCTOR_KIND")
abbreviation = getDatabase(dom, "ABBREVIATION")
sourcefile = getDatabase(dom, "SOURCEFILE")
constructorkind.1 = upperCase constructorkind.1
description = getDescription(dom)
```

--- 

```html
<table>
  <tr><td>Description: </td><td>"</td></tr>
  <tr><td>Abbreviation: </td><td>"</td></tr>
  <tr><td>Source File: </td><td>"</td></tr>
</table>
```
readTheFile

We have q which is a stream which contains the file. We read the file into a string-stream to get it all into one string. We return the string.

--- package AXSERV AxiomServer ---

readTheFile(q:SExpression):String ==
  -- WriteLine("begin reading file")$Lisp
  r := MAKE_-STRING_-OUTPUT_-STREAM()$Lisp
  SiCopyStream(q,r)$Lisp
  filestream:String := GET_-OUTPUT_-STREAM_-STRING(r)$Lisp
  CLOSE(r)$Lisp
  CLOSE(q)$Lisp
  -- WriteLine("end reading file")$Lisp
  filestream
(outputToSocket)

We have “s” which is the socket, “filestream” which is the text of the file to output, and “contentType” which is the HTML Content-Type. We construct the HTML header information according to the standard and prepend it to the file. The resulting string is output to the socket.

---

(outputToSocket(s:SExpression,filestream:String,contentType:String):Void ==
  filelength:String := string(#filestream)
  file:String := ""
  nl:String:=STRING(NewLine$Lisp)$Lisp
  file := concat ["Content-Length: ",filelength,nl,nl,file]
  file := concat ["Connection: close",nl,file]
  file := concat ["Content-Type: " ,contentType,nl,file]
  file := concat ["HTTP/1.1 200 OK",nl,file]
  file := concat [file,filestream]
  -- WriteLine(file)$Lisp
  f:=MAKE_STRING_INPUT_STREAM(file)$Lisp
  SiCopyStream(f,s)$Lisp
  CLOSE(f)$Lisp
  CLOSE(s)$Lisp

---

(getCommand)

The getCommand function is invoked when the HTTP request is a POST and contains the string "command". Essentially the game here is to rebind the various output streams used by Axiom so we can capture the normal output. This function returns a set of HTML 5 div blocks:

1. stepnum, the value of lastStep()
2. command, the value of the command variable
3. algebra, the value of the algebra variable
4. mathml, the value of the mathml variable
5. type, the value of lastType()

The HTML functions in the hyperdoc browser depend on the order of these variables so do not change this without changing the corresponding functions in the browser HTML.

---
getCommand(s: SExpression, command: String): Void ==
    WriteLine$Lisp concat ["getCommand: ", command]
    SETQ(tmpmathml$Lisp, MAKE_-STREAM_-OUTPUT_-STREAM()$Lisp)
    SETQ(tmpalgebra$Lisp, MAKE_-STREAM_-OUTPUT_-STREAM()$Lisp)
    SETQ(savemathml$Lisp, _$texOutputStream$Lisp)
    SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)
    SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)
    SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)
    ans := string parseAndEvalToStringEqNum$Lisp command
    SETQ(resultmathml$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$texOutputStream$Lisp)$Lisp)
    SETQ(resultalgebra$Lisp,_
         GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)
    SETQ(_$texOutputStream$Lisp,savemathml$Lisp)
    SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)
    CLOSE(tmpmathml$Lisp)
    CLOSE(tmpalgebra$Lisp)
    -- Since strings returned from axiom are going to be displayed in html I
    -- should really check for the characters &,<,> and replace them with
    -- &,<,>.
    -- At present I only check for ampersands in formatMessages.
    mathml:String := string(resultmathml$Lisp)
    algebra:String := string(resultalgebra$Lisp)
    algebra := formatMessages(algebra)
    -- At this point mathml contains the mathml for the output but does not
    -- include step number or type information.
    -- We should also save the command.
    -- I get the type and step number from the $internalHistoryTable
    axans:String :=
        concat ["<div class=_'stepnum_'>", lastStep(), "</div>_
         <div class=_'command_'>", command, "</div>_
         <div class=_'algebra_'>", algebra, "</div>_
         <div class=_'mathml_'>", mathml, "</div>_
         <div class=_'type_'>", lastType(), "</div>]
    -- WriteLine$Lisp concat ['mathml answer: ",mathml]
    -- WriteLine$Lisp concat ['algebra answer: ",algebra]
    q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
    SiCopyStream(q,s)$Lisp
    CLOSE(q)$Lisp
    CLOSE(s)$Lisp

getInterp

The getInterp function is invoked when the HTTP request is a POST and contains the
string "command". Essentially the game here is to rebind the various output streams used
by Axiom so we can capture the normal output. This function returns a set of HTML 5 div
PACKAGEx AXSERV AXIOMSERVER

blocks:

1. stepnum, the value of lastStep()
2. command, the value of the command variable
3. algebra, the value of the algebra variable
4. mathml, the value of the mathml variable
5. type, the value of lastType()

The HTML functions in the hyperdoc browser depend on the order of these variables so do not change this without changing the corresponding functions in the browser HTML.

```lisp
getInterp(s:SExpression,command:String):Void ==
  WriteLine$Lisp concat ['"getInterp: ",command]
  SETQ(tmpmathml$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)
  SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)
  SETQ(savemathml$Lisp, _$texOutputStream$Lisp)
  SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)
  SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
  SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
  ans := string parseAndEvalToStringEqNum$Lisp command
  SETQ(resultmathml$Lisp,)
    GET_-OUTPUT_-STREAM_-STRING(_$texOutputStream$Lisp)$Lisp)
  SETQ(resultalgebra$Lisp,)
    GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)
  SETQ(_$texOutputStream$Lisp,savemathml$Lisp)
  SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)
  CLOSE(tmpmathml$Lisp)$Lisp
  CLOSE(tmpalgebra$Lisp)$Lisp
  -- Since strings returned from axiom are going to be displayed in html I
  -- should really check for the characters $,<$,> and replace them with
  -- &amp;,&lt;,&gt;
  -- At present I only check for ampersands in formatMessages.
  mathml:String := string(resultmathml$Lisp)
  algebra:String := string(resultalgebra$Lisp)
  algebra := formatMessages(algebra)
  -- At this point mathml contains the mathml for the output but does not
  -- include step number or type information.
  -- We should also save the command.
  -- I get the type and step number from the $internalHistoryTable
  axans:String :=
    concat ['"div class="_stepnum_"">", lastStep(), "</div>
"div class="_command_"">", command, "</div>
"div class="_algebra_"">",algebra,"</div>
"div class="_mathml_"">",mathml,"</div>
"div class="_type_"">",lastType(),"</div>"
```
getLisp

The getLisp function is invoked when the HTTP request is a POST and contains the string "lispcall".

--- package AXSERV AxiomServer ---

getLisp(s:SExpression,command:String):Void ==
  WriteLine$Lisp concat ["getLisp: ",command]
  evalresult:=EVAL(READ_-FROM_-STRING(command)$Lisp)$Lisp
  mathml:String:=string(evalresult)
  -- WriteLine$Lisp concat ["getLisp: after ",mathml]
  -- WriteLine$Lisp concat ["getLisp output: ",mathml]
  SETQ(tmpalgebra$Lisp, MAKE_-STRING_-OUTPUT_-STREAM()$Lisp)$Lisp
  SETQ(savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
  SETQ(savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
  SETQ(_$texOutputStream$Lisp,tmpmathml$Lisp)$Lisp
  SETQ(_$algebraOutputStream$Lisp,tmpalgebra$Lisp)$Lisp
  SETQ(resultalgebra$Lisp,_
        GET_-OUTPUT_-STREAM_-STRING(_$algebraOutputStream$Lisp)$Lisp)$Lisp
  SETQ(_$texOutputStream$Lisp,savemathml$Lisp)$Lisp
  SETQ(_$algebraOutputStream$Lisp,savealgebra$Lisp)$Lisp
  CLOSE(tmpalgebra$Lisp)$Lisp
  -- Since strings returned from axiom are going to be displayed in html I
  -- should really check for the characters &,<,> and replace them with
  -- &amp;,&lt;&gt;.
  -- At present I only check for ampersands in formatMessages.
  algebra:String := string(resultalgebra$Lisp)
  algebra := formatMessages(algebra)
  -- At this point mathml contains the mathml for the output but does not
  -- include step number or type information.
  -- We should also save the command.
  -- I get the type and step number from the $internalHistoryTable
  axans:String := _
        concat ['"<div class=_'stepnum_">", lastStep(), "]
        <div class=_'command_">", command, "]
        <div class=_'algebra_">",algebra,"</div>
        <div class=_'mathml_">",mathml,"</div>
        <div class=_'type_">",lastType(),"</div>"

  -- WriteLine$Lisp concat ["mathml answer: ",mathml]
getShow

The getShow function is invoked when the HTTP request is a POST and contains the string "showcall". The )show command generates output to lisp's "standard-output" so we wrap that stream to capture it. The resulting string needs to be transformed into html-friendly form. This is done in the call to replace-entities (see http.lisp)

```lisp
| package AXSERV AxiomServer |

getShow

getShow(s:SExpression, showarg: String): Void ==
  WriteLine$Lisp concat ["getShow: ", showarg]
  realarg:=SUBSEQ(showarg,6)$Lisp
  show:=
  "(progn (setq |$options| '((|operations|))) (|show| '|" realarg "))"
  WriteLine$Lisp concat ["getShow: ", show]
  SETQ(SAVESTREAM$Lisp, *STANDARD_OUTPUT_*$Lisp)$Lisp
  SETQ(*$STANDARD_OUTPUT_*$Lisp,)
  MAKE_STRING_OUTPUT_STREAM()$Lisp)$Lisp
  evalresult:=EVAL(READ_FROM_STRING(show)$Lisp)$Lisp
  SETQ(evalresult,)
  GET_OUTPUT_STREAM_STRING($STANDARD_OUTPUT_*$Lisp)$Lisp
  SETQ($STANDARD_OUTPUT_*$Lisp,SAVESTREAM$Lisp)$Lisp
  mathml: String:=string(REPLACE_ENTITIES(evalresult)$Lisp)$Lisp
  SETQ($standard_output_stream$Lisp, MAKE_STRING_OUTPUT_STREAM()$Lisp)$Lisp
  SETQ($savemathml$Lisp, _$texOutputStream$Lisp)$Lisp
  SETQ($savealgebra$Lisp, _$algebraOutputStream$Lisp)$Lisp
  SETQ($algebraOutputStream$Lisp,$texOutputStream$Lisp)$Lisp
  SETQ($algebraOutputStream$Lisp,$texOutputStream$Lisp)$Lisp
  SETQ($algebraOutputStream$Lisp,$texOutputStream$Lisp)$Lisp
  SETQ($algebraOutputStream$Lisp,$texOutputStream$Lisp)$Lisp
  -- Since strings returned from axiom are going to be displayed in html I
  -- should really check for the characters &,<,> and replace them with
  -- &amp;,&lt;,&gt;.
  -- At present I only check for ampersands in formatMessages.
  algebra: String := string(resultalgebra$Lisp)$Lisp
  algebra := formatMessages(algebra)
  -- At this point mathml contains the mathml for the output but does not
  -- include step number or type information.
```
-- We should also save the command.
-- I get the type and step number from the $internalHistoryTable
axans:String := _
concat ["<div class=_'stepnum_'">", lastStep(), "</div>_
  <div class=_'command_'">", showarg, "</div>_
  <div class=_'algebra_'">",algebra,"</div>_
  <div class=_'mathml_'">",mathml,"</div>_
  <div class=_'type_'">",lastType(),"</div>_
] -- WriteLine$Lisp concat ["mathml answer: ",mathml]
q:=MAKE_-STRING_-INPUT_-STREAM(axans)$Lisp
SiCopyStream(q,s)$Lisp
CLOSE(q)$Lisp
CLOSE(s)$Lisp

_________

lastType

To examine the $internalHistoryTable use the following line

)lisp |$internalHistoryTable|

We need to pick out first member of internalHistoryTable and then pick out the element with % as first element. Here is an example showing just the first element of the list, which correponds to the last command.

Note that the last command does not necessarily correspond to the last element of the first element of $internalHistoryTable as it is in this example.

(4 NIL
 (x (value (BasicOperator) WRAPPED . #<vector 09a93bd0>))
 (y (value (BasicOperator) WRAPPED . #<vector 09a93bb4>))
 (% (value (Matrix (Polynomial (Integer))) WRAPPED . #<vector 0982e0e0>))
 )
 ... )

We also need to check for input error in which case the $internalHistoryTable is not changed and the type retrieved would be that for the last correct input.

— package AXSERV AxiomServer —

lastType():String ==
  SETQ(first$Lisp,FIRST($internalHistoryTable$Lisp)$Lisp)$Lisp
  count:Integer := 0
  hisLength:Integer := LIST_-LENGTH($internalHistoryTable$Lisp)$Lisp
  length:Integer := LIST_-LENGTH(first$Lisp)$Lisp
-- This initializes stepSav. The test is a bit of a hack, maybe I’ll
-- figure out the right way to do it later.
if string stepSav$Lisp = "#<OBJNULL>" then SETQ(stepSav$Lisp, 0$Lisp)$Lisp
-- If hisLength = 0 then the history table has been reset to NIL
-- and we’re starting numbering over
if hisLength = 0 then SETQ(stepSav$Lisp, 0$Lisp)$Lisp
if hisLength > 0 and
  car(car($internalHistoryTable$Lisp)$Lisp)$Lisp = stepSav$Lisp
  then SETQ(stepSav$Lisp, car(car($internalHistoryTable$Lisp)$Lisp)$Lisp)
  while count < length repeat
    position(char "%", string FIRST(first$Lisp)$Lisp) = 2 =>
      count := length+1
    count := count + 1
    SETQ(first$Lisp, REST(first$Lisp)$Lisp)$Lisp
  count = length + 1 =>
    string SECOND(SECOND(FIRST(first$Lisp)$Lisp)$Lisp)$Lisp

lastStep():String ==
string car($internalHistoryTable$Lisp)$Lisp

formatMessages(str:String):String ==
  -- WriteLine("formatMessages")$Lisp
  -- I need to replace any ampersands with &amp; and may also need to
  -- replace < and > with &lt; and &gt;
  strlist:List String
  -- WriteLine(str)$Lisp
  strlist := split(str, char "&")
  str := ""
  -- oops, if & is the last character in the string this method
  -- will eliminate it. Need to redo this.
  for s in strlist repeat
    str := concat [str, s, "&"]
  strlen:Integer := #str
  str := str.(1..(#str - 5))
  -- WriteLine(str)$Lisp
  -- Here I split the string into lines and put each line in a "div".
  strlist := split(str, char string NewlineChar$Lisp)
  str := ""
  -- WriteLine("formatMessages1")$Lisp
  -- WriteLine(concat strlist)$Lisp
  for s in strlist repeat
    -- WriteLine(s)$Lisp
    str := concat [str, "<div>", s, "</div>"]
  str

getContentType(pathvar:String):String ==
  -- WriteLine("getContentType begin")$Lisp
  -- set default content type
contentType: String := "text/plain"
-- need to test for successful match?
StringMatch(".\.(.*)", pathvar)$Lisp
u: UniversalSegment(Integer)
u := segment(MatchBeginning(1)$Lisp + 1, _
    MatchEnd(1)$Lisp)$UniversalSegment(Integer)
extension: String := pathvar.u
-- WriteLine$Lisp concat ["file extension: ", extension]
-- test for extensions: html, htm, xml, xhtml, js, css
if extension = "html" then
    contentType: String := "text/html"
else if extension = "htm" then
    contentType: String := "text/html"
else if extension = "xml" then
    contentType: String := "text/xml"
else if extension = "xhtml" then
    contentType: String := "application/xhtml+xml"
else if extension = "js" then
    contentType: String := "text/javascript"
else if extension = "css" then
    contentType: String := "text/css"
else if extension = "png" then
    contentType: String := "image/png"
else if extension = "jpg" then
    contentType: String := "image/jpeg"
else if extension = "jpeg" then
    contentType: String := "image/jpeg"
-- WriteLine$Lisp concat ["Content-Type: ", contentType]
-- WriteLine("getContentType end")$Lisp
contentType

—— AXSERV.dotabb ——

"AXSERV" [color="#FF4488", href="bookvol10.4.pdf#nameddest=AXSERV"]
"STRING" [color="#88FF44", href="bookvol10.3.pdf#nameddest=STRING"]
"AXSERV" -> "STRING"
Chapter 3

Chapter B

package BALFACT BalancedFactorisation

| BalancedFactorisation.input |

)set break resume
)sys rm -f BalancedFactorisation.output
)spool BalancedFactorisation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BalancedFactorisation
--R
--R BalancedFactorisation(R: Join(GcdDomain,CharacteristicZero),UP: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for BalancedFactorisation is BALFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for BALFACT
--R
--R-------------------------------------- Operations ----------------------------------
--R balancedFactorisation : (UP,UP) -> Factored(UP)
--R balancedFactorisation : (UP,List(UP)) -> Factored(UP)
--R
--E 1

)spool
)lisp (bye)
BalancedFactorisation (BALFACT)

Exports:
balancedFactorisation

— package BALFACT BalancedFactorisation —

)abbrev package BALFACT BalancedFactorisation
++ Author: Manuel Bronstein
++ Date Created: 1 March 1991
++ Date Last Updated: 11 October 1991
++ Description:
++ This package provides balanced factorisations of polynomials.

BalancedFactorisation(R, UP): Exports == Implementation where
  R : Join(GcdDomain, CharacteristicZero)
  UP : UnivariatePolynomialCategory R

Exports ==> with
balancedFactorisation: (UP, UP) -> Factored UP
  ++ balancedFactorisation(a, b) returns
  ++ a factorisation \spad{a = p_1^{e_1} \ldots p_m^{e_m}} such that each
  ++ \spad{p_i} is balanced with respect to b.
balancedFactorisation: (UP, List UP) -> Factored UP
  ++ balancedFactorisation(a, [b_1,\ldots,b_n]) returns
  ++ a factorisation \spad{a = p_1^{e_1} \ldots p_m^{e_m}} such that each
  ++ p_i is balanced with respect to \spad{[b_1,\ldots,b_m]}.

Implementation ==> add
balSqr : (UP, Integer, List UP) -> Factored UP
balSqr1: (UP, Integer, UP) -> Factored UP

balancedFactorisation(a:UP, b:UP) == balancedFactorisation(a, [b])
balSqr1(a, n, b) ==
g := gcd(a, b)
fa := sqfrFactor((a exquo g)::UP, n)
ground? g => fa
fa * balSqr1(g, n, (b exquo (g ** order(b, g)))::UP)

balSqr(a, n, l) ==
b := first l
empty? rest l => balSqr1(a, n, b)
*/[balSqr1(f.factor, n, b) for f in factors balSqr(a,n,rest l)]

balancedFactorisation(a:UP, l:List UP) ==
empty?(ll := select(z1 +-> z1 ^= 0, l)) =>
  error "balancedFactorisation: 2nd argument is empty or all 0"
sa := squareFree a
unit(sa) * */[balSqr(f.factor,f.exponent,ll) for f in factors sa]
This package exports functions to set some commonly used properties of operators, including properties which contain functions.

See Also:
  - )show BasicOperatorFunctions1
BasicOperatorFunctions1 (BOP1)

Exports:
constantOpIfCan constantOperator derivative evaluate

<table>
<thead>
<tr>
<th>package BOP1 BasicOperatorFunctions1</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ Author: Manuel Bronstein</td>
</tr>
<tr>
<td>++ Date Created: 28 Mar 1988</td>
</tr>
<tr>
<td>++ Date Last Updated: 15 May 1990</td>
</tr>
<tr>
<td>++ Description:</td>
</tr>
<tr>
<td>++ This package exports functions to set some commonly used properties</td>
</tr>
<tr>
<td>++ of operators, including properties which contain functions.</td>
</tr>
</tbody>
</table>

BasicOperatorFunctions1(A:SetCategory): Exports == Implementation where

OP  ==> BasicOperator
EVAL ==> "%eval"
CONST ==> "%constant"
DIFF  ==> "%diff"
OUT   ==> OutputForm
IN    ==> InputForm

Exports ==> with

evaluate : (OP, List A) -> Union(A, "failed")
++ evaluate(op, [a1,...,an]) checks if op has an "%eval"
++ property f. If it has, then \spad{f(a1,...,an)} is returned, and
++ "failed" otherwise.
evaluate : (OP, List A -> A) -> OP
++ evaluate(op, foo) attaches foo as the "%eval" property
++ of op. If op has an "%eval" property f, then applying op
++ to \spad{\{a1,...,an\}} returns the result of \spad{f(a1,...,an)}.
evaluate : (OP, A -> A) -> OP
++ evaluate(op, foo) attaches foo as the "%eval" property
++ of op. If op has an "%eval" property f, then applying op
++ to a returns the result of \spad{f(a)}. Argument op must be unary.
evaluate : OP -> Union(List A -> A, "failed")
++ evaluate(op) returns the value of the "%eval" property of
++ op if it has one, and "failed" otherwise.
derivative : (OP, List (List A -> A)) -> OP
++ derivative(op, [foo1,...,foon]) attaches [foo1,...,foon] as
++ the "%diff" property of op. If op has an "%diff" property
++ \spad{[f1,...,fn]} then applying a derivation D
++ to \spad{op(a1,...,an)}
++ returns \spad{f1(a1,...,an) * D(a1) + ... + fn(a1,...,an) * D(an)}.
derivative : (OP, A -> A) -> OP
++ derivative(op, foo) attaches foo as the "%diff" property
++ of op. If op has an "%diff" property f, then applying a
++ derivation D to op(a) returns \spad{f(a) * D(a)}.
++ Argument op must be unary.
derivative : OP -> Union(List(List A -> A), "failed")
++ derivative(op) returns the value of the "%diff" property of
++ op if it has one, and "failed" otherwise.
if A has OrderedSet then
  constantOperator: A -> OP
  ++ constantOperator(a) returns a nullary operator op
  ++ such that \spad{op()} always evaluate to \spad{a}.
  constantOpIfCan : OP -> Union(A, "failed")
  ++ constantOpIfCan(op) returns \spad{a} if op is the constant
  ++ nullary operator always returning \spad{a}, "failed" otherwise.

Implementation ==> add
evaluate(op:OP, func:A -> A) ==
  evaluate(op, (ll:List(A)):A +-> func first ll)
evaluate op ==
  (func := property(op, EVAL)) case "failed" => "failed"
  (func::None) pretend (List A -> A)
evaluate(op:OP, args:List A) ==
  (func := property(op, EVAL)) case "failed" => "failed"
  ((func::None) pretend (List A -> A)) args
evaluate(op:OP, func:List A -> A) ==
  setProperty(op, EVAL, func pretend None)
derivative op ==
  (func := property(op, DIFF)) case "failed" => "failed"
  ((func::None) pretend List(List A -> A))
derivative(op:OP, grad:List(List A -> A)) ==
  setProperty(op, DIFF, grad pretend None)
derivative(op:OP, f:A -> A) ==
  unary? op or nary? op =>
  derivative(op, [ll:List(A)]:A +-> f first ll)$List(List(A -> A))
error "Operator is not unary"
if A has OrderedSet then
  cdisp : (OUT, List OUT) -> OUT
  csex : (IN, List IN) -> IN
  eqconst?: (OP, OP) -> Boolean
  ltconst?: (OP, OP) -> Boolean
  constOp : A -> OP

  opconst:OP :=
    comparison(equality(operator("constant"::Symbol, 0), eqconst?),
          ltconst?)

  cdisp(a, l) == a
  csex(a, l) == a

  eqconst?(a, b) ==
    (va := property(a, CONST)) case "failed" => not has?(b, CONST)
    (vb := property(b, CONST)) case None) and
    ((va::None) pretend A) = ((vb::None) pretend A)

  ltconst?(a, b) ==
    (va := property(a, CONST)) case "failed" => has?(b, CONST)
    (vb := property(b, CONST)) case None) and
    ((va::None) pretend A) < ((vb::None) pretend A)

  constOp a ==
    setProperty(
      display(copy opconst, (ll:List(OUT)):OUT +-> cdisp(a::OUT, ll)),
      CONST, a pretend None)

  constantOpIfCan op ==
    is?(op, "constant"::Symbol) and
    ((u := property(op, CONST)) case None) => (u::None) pretend A
    "failed"

if A has ConvertibleTo IN then
  constantOperator a ==
    input(constOp a, (ll:List(IN)):IN +-> csex(convert a, ll))
else
  constantOperator a == constOp a
package BEZIER Bezier

-- Bezier.input --

)set break resume
)sys rm -f Bezier.output
)spool Bezier.output
)set message test on
)set message auto off
)clear all

--S 1 of 10
n:=linearBezier([2.0,2.0],[4.0,4.0])
--R
--I (1) theMap(BEZIER;linearBezier;2LM;1!0,707)
--R Type: (Float -> List(Float))
--E 1

--S 2 of 10
[n(t/10.0) for t in 0..10 by 1]
--R
--R (2)
--R [[2.0,2.0], [2.2,2.2], [2.4,2.4], [2.6,2.6], [2.8,2.8], [3.0,3.0],
--R [3.2,3.2], [3.4,3.4], [3.6,3.6], [3.8,3.8], [4.0,4.0]]
--R Type: List(List(Float))
--E 2

--S 3 of 10
n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
--R
--I (3) theMap(BEZIER;quadraticBezier;3LM;2!0,291)
--R Type: (Float -> List(Float))
--E 3

--S 4 of 10
[n(t/10.0) for t in 0..10 by 1]
--R
--R (4)
--R [[2.0,2.0], [2.4,2.36], [2.8,2.64], [3.2,2.84], [3.6,2.96], [4.0,3.0],
--R [4.4,2.96], [4.8,2.84], [5.2,2.64], [5.6,2.36], [6.0,2.0]]
--R Type: List(List(Float))
--E 4
n := cubicBezier([2.0, 2.0], [2.0, 4.0], [6.0, 4.0], [6.0, 2.0])

functions := [quadraticBezier([2.0, 2.0], m, [6.0, 2.0]) for m in line]

graphs := [[point(((functions.i)(j/100.0))::LIST(DFLOAT)) :: LIST(DFLOAT)) :: LIST(DFLOAT)) for j in 0..100 for i in 1..9]
[1.1424000000000003, 2.8843999999999999],
[1.2383999999999999, 2.8704000000000001],
[1.3375999999999999, 2.8555999999999999],
[1.4399999999999999, 2.8399999999999999],
[1.5456000000000003, 2.8235999999999999],
[1.6544000000000003, 2.8064],
[1.7664, 2.7884000000000002],
[1.8815999999999999, 2.7696000000000005],
[2., 2.75],
[2.1215999999999999, 2.7296000000000005],
[2.2464, 2.7084000000000001],
[2.3744000000000005, 2.6863999999999999],
[2.5055999999999998, 2.6636000000000006],
[2.6400000000000006, 2.6400000000000006],
[2.7760000000000005, 2.6156000000000006],
[2.9184000000000001, 2.5903999999999998],
[3.0623999999999998, 2.5644],
[3.2096, 2.5375999999999999],
[3.3599999999999999, 2.5099999999999999],
[3.5135999999999998, 2.4815999999999999],
[3.6703999999999999, 2.4523999999999999],
[3.804, 2.4224000000000006],
[4.1600000000000001, 2.3915999999999999],
[4.3296000000000001, 2.3679999999999999],
[4.5023999999999997, 2.2944],
[4.6783999999999997, 2.2256],
[5.0400000000000000, 2.1536],
[5.4139999999999997, 2.1164000000000005],
[5.6063999999999998, 2.0783999999999998],
[5.8015999999999996, 2.0396000000000001],
[6., 2.]]
CHAPTER 3. CHAPTER B

--R [3.9952000000000005, 2.4815999999999998],
--R [4.1227999999999998, 2.4523999999999999],
--R [4.2527999999999997, 2.4224000000000006],
--R [4.3852000000000001, 2.3915999999999999],
--R [4.5199999999999996, 2.3599999999999999],
--R [4.6571999999999996, 2.3275999999999999],
--R [4.7968000000000011, 2.2944],
--R [4.9387999999999996, 2.2604000000000006],
--R [5.0831999999999997, 2.2256],
--R [5.2300000000000004, 2.1899999999999999],
--R [5.3792, 2.1536],
--R [5.530800000000001, 2.1164000000000005],
--R [5.6848000000000011, 2.0783999999999998],
--R [5.8411999999999997, 2.0396000000000001], [6., .2]
--R ,
--R [2., .2], [1.9410000000000003, 2.0396000000000001], [1.8399999999999999, 2.0783999999999999], [1.7760000000000002, 2.1536], [1.7250000000000001, 2.1899999999999999],
--R [1.6759999999999999, 2.2256], [1.629, 2.2604000000000006], [1.5840000000000001, 2.2944], [1.5409999999999999, 2.3275999999999999], [1.5, 2.3599999999999999], [1.4610000000000003, 2.3915999999999999],
--R [1.4239999999999999, 2.4224000000000006], [1.3890000000000002, 2.4523999999999999],
--R [1.3560000000000003, 2.4815999999999998], [1.325, 2.5099999999999999], [1.2960000000000002, 2.5375999999999999], [1.2699999999999999, 2.5644],
--R [1.244, 2.5903999999999999], [1.2100000000000001, 2.6156000000000006], [1.2, 2.6400000000000006], [1.181, 2.6636000000000006], [1.1639999999999999, 2.6863999999999999], [1.149, 2.7084000000000001], [1.1359999999999999, 2.7296000000000005], [1.125, 2.75],
--R [1.1160000000000001, 2.7696000000000005], [1.109, 2.7840000000000002], [1.1040000000000001, 2.8064], [1.101, 2.8359999999999999],
--R [1.0000000000000001, 2.8399999999999999], [1.01, 2.8559999999999999],
--R [1.1040000000000001, 2.8704000000000001], [1.109, 2.8843999999999999],
--R [1.1160000000000001, 2.8976000000000006], [1.125, 2.9100000000000001], [1.1359999999999999, 2.9216000000000006], [1.149, 2.9323999999999999],
--R [1.1639999999999999, 2.9424000000000001], [1.181, 2.9516], [1.2, 2.96],
--R [1.2210000000000001, 2.9676], [1.244, 2.9744000000000002], [1.2689999999999999, 2.9803999999999999],
--R [1.2960000000000003, 2.9859999999999999], [1.325, 2.9900000000000002],
--R [1.3560000000000003, 2.9935999999999999], [1.3890000000000002, 2.9964],
--R [1.4239999999999999, 2.9984000000000002], [1.4610000000000003, 2.9996],
--R [1.5, .3], [1.5409999999999999, 2.9996],
--R [1.5840000000000001, 2.9984000000000002], [1.629, 2.9964],
--R [1.6759999999999999, 2.9935999999999999],
--R [1.7250000000000001, 2.9900000000000002],
--R [1.7760000000000002, 2.9859999999999999], [1.829, 2.9803999999999999],
--R [1.8399999999999999, 2.9744000000000002], [1.94100000000000003, 2.9676],
--R [2., 2.96], [2.0609999999999999, 2.9516],
--R [2.1240000000000006, 2.9424000000000002], [2.1890000000000001, 2.9323999999999999],
--R [2.2559999999999999, 2.9216000000000006],
--R [2.3250000000000002, 2.9100000000000001],
--R [4.4800000000000004, 2.9424000000000001],
--R [4.5199999999999996, 2.9323999999999999],
--R [4.5599999999999996, 2.9160000000000006],
--R [4.5999999999999996, 2.9000000000000001],
--R [4.6399999999999997, 2.8976000000000006],
--R [4.6799999999999997, 2.8843999999999999],
--R [4.7199999999999998, 2.8704000000000001],
--R [4.7599999999999999, 2.8559999999999999],
--R [4.7999999999999999, 2.8399999999999999],
--R [4.8399999999999999, 2.8235999999999999],
--R [4.8799999999999999, 2.8064],
--R [4.9199999999999999, 2.7884000000000002],
--R [5., 2.75],
--R [5.0800000000000001, 2.7796000000000005],
--R [5.1200000000000001, 2.7696000000000005],
--R [5.1600000000000001, 2.7596000000000005],
--R [5.2000000000000001, 2.7496000000000005],
--R [5.2400000000000001, 2.7396000000000005],
--R [5.2800000000000001, 2.7296000000000005],
--R [5.3200000000000003, 2.5644],
--R [5.3600000000000001, 2.5544],
--R [5.4000000000000004, 2.5444],
--R [5.4400000000000001, 2.5344],
--R [5.4800000000000004, 2.5244],
--R [5.5200000000000004, 2.5144],
--R [5.5600000000000004, 2.5044],
--R [5.6000000000000004, 2.4944],
--R [5.6400000000000004, 2.4844],
--R [5.6800000000000004, 2.4744],
--R [5.7200000000000004, 2.4644],
--R [5.7600000000000004, 2.4544],
--R [5.8000000000000004, 2.4444],
--R [5.8400000000000004, 2.4344],
--R [5.8800000000000004, 2.4244],
--R [5.9200000000000004, 2.4144],
--R [5.9600000000000004, 2.4044],
--R [6., 2.]]

--R Type: List(List(Point(DoubleFloat)))

/R 9

S 10 of 10
)show Bezier

R Bezier(R: Ring) is a package constructor
R Abbreviation for Bezier is BEZIER
R This constructor is exposed in this frame.
R Issue )edit bookvol10.4.pamphlet to see algebra source code for BEZIER

------------- Operations -----------------------------
R cubicBezier : (List(R),List(R),List(R),List(R)) -> (R -> List(R))
R linearBezier : (List(R),List(R)) -> (R -> List(R))
R quadraticBezier : (List(R),List(R),List(R)) -> (R -> List(R))
R
E 10
-- We do not do these during testing since graphics is not available
-- The resulting image is in the Bezier section of Volume 10.4

```plaintext
--d1:=draw(graphs.1,title=="Bezier Control Point Motion")
--others:=[graphs.i for i in 2..9]
--for i in 2..9 for graph in others repeat putGraph(d1,[graph],i)
--vp:=makeViewport2D(d1)
```

--- Bezier.help ---

BezoutMatrix contains functions for computing resultants and
discriminants using Bezout matrices.

A linear Bezier curve is a simple interpolation between the
starting point and the ending point based on a parameter t.

Given a start point a=[x1,y1] and an endpoint b=[x2,y2]
f(t) == [(1-t)*x1 + t*x2, (1-t)*y1 + t*y2]

n:=linearBezier([2.0,2.0],[4.0,4.0])
  theMap(BEZIER;linearBezier;2LM;110,707)

[n(t/10.0) for t in 0..10 by 1]
A quadratic Bezier curve is a simple interpolation between the starting point, a middle point, and the ending point based on a parameter t.

Given a start point \( a = [x_1, y_1] \), a middle point \( b = [x_2, y_2] \), and an endpoint \( c = [x_3, y_3] \)

\[
f(t) = [(1-t)^2 x_1 + 2t(1-t) x_2 + t^2 x_3, \]
\[
(1-t)^2 y_1 + 2t(1-t) y_2 + t^2 y_3]
\]

```
n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
theMap(BEZIER;quadraticBezier;3LM;2!0,291)

[n(t/10.0) for t in 0..10 by 1]
[[2.0,2.0], [2.4,2.36], [2.8,2.64], [3.2,2.84], [3.6,2.96], [4.0,3.0],
 [4.4,2.96], [4.8,2.84], [5.2,2.64], [5.6,2.36], [6.0,2.0]]
```

A cubic Bezier curve is a simple interpolation between the starting point, a left-middle point, a right-middle point, and the ending point based on a parameter t.

Given a start point \( a = [x_1, y_1] \), the left-middle point \( b = [x_2, y_2] \), the right-middle point \( c = [x_3, y_3] \), and an endpoint \( d = [x_4, y_4] \)

\[
f(t) = [(1-t)^3 x_1 + 3t(1-t)^2 x_2 + 3t^2 (1-t) x_3 + t^3 x_4,
(1-t)^3 y_1 + 3t(1-t)^2 y_2 + 3t^2 (1-t) y_3 + t^3 y_4]
\]

```
n:=cubicBezier([2.0,2.0],[2.0,4.0],[6.0,4.0],[6.0,2.0])
theMap(BEZIER;cubicBezier;4LM;3!0,915)

[n(t/10.0) for t in 0..10 by 1]
[[2.0,2.0], [2.112,2.54], [2.416,2.96], [2.864,3.26], [3.408,3.44],
 [4.0,3.5], [4.592,3.44], [5.136,3.26], [5.584,2.96], [5.888,2.54],
 [6.0,2.0]]
```

Bezier curves "move" based on moving their control points, which in the case of the three components of a quadratic Bezier curve are the three points given. To see this motion we can show what happens when you "drag" the middle control point along the line from \([-4,4]\) to \([4,4]\) by increments of 1.

First, we form the line as a list of Floats, 9 in all.

```
line:=[[i::Float,4.0] for i in -4..4 by 1]
```

Next, we form a list of functions, each with a different center control point.
Notice that the endpoints remain fixed so we expect that the curve will begin and end at the same point but that the midpoint is pulled around.

\[
\text{functions} := [\text{quadraticBezier}([2.0,2.0], m, [6.0,2.0]) \text{ for } m \text{ in line}]
\]

Then we form a list of the graphs by calling each function in the above list. Each function call happens 101 times (to include both endpoints). Thus we get a List of Lists of Points of DoubleFloats

\[
\text{graphs} := [\text{point}((\text{functions}.i)(j/100.0))::\text{LIST}(\text{DFLOAT}) \text{ for } j \text{ in } 0..100 \text{ for } i \text{ in } 1..9]
\]

We draw the first graph to see if it looks reasonable:

\[
\text{d1} := \text{draw}(\text{graphs}.1)
\]

Since it does we add the other 8 graphs. The 2D graphs can hold up to 9 simultaneous graphs.

\[
\text{others} := [\text{graphs}.i \text{ for } i \text{ in } 2..9]
\]

for i in 2..9 for graph in others repeat \text{putGraph}(\text{d1},[\text{graph}],i)

and now we display them all on one graph.

\[
\text{vp} := \text{makeViewport2D}(\text{d1})
\]

See Also:
- )show Bezier
- )show TwoDimensionalViewport
- )d op draw
- )d op point
- )d op putGraph
- )d op makeViewport2D
Beziers (BEZIER)

Exports:
linearBezier  quadraticBezier  cubicBezier

--- package BEZIER Bezir ---

)abbrev package BEZIER Bezier
++ Author: Timothy Daly
++ Date Created: 14 April 2009
++ Description:
++ Provide linear, quadratic, and cubic spline bezier curves

Bezier(R:Ring): with
  linearBezier: (x:List R,y:List R) -> Mapping(List R,R)
    ++ A linear Bezir curve is a simple interpolation between the
    ++ starting point and the ending point based on a parameter t.
    ++ Given a start point a=[x1,y1] and an endpoint b=[x2,y2]
    ++ f(t) == [(1-t)*x1 + t*x2, (1-t)*y1 + t*y2]
    ++
    ++X n:=linearBezier([2.0,2.0],[4.0,4.0])
    ++X [n(t/10.0) for t in 0..10 by 1]
  quadraticBezier: (x:List R,y:List R,z:List R) -> Mapping(List R,R)
    ++ A quadratic Bezir curve is a simple interpolation between the
    ++ starting point, a middle point, and the ending point based on
    ++ a parameter t.
    ++ Given a start point a=[x1,y1], a middle point b=[x2,y2],
    ++ and an endpoint c=[x3,y3]
    ++ f(t) == [(1-t)^2 x1 + 2t(1-t) x2 + t^2 x3,
                (1-t)^2 y1 + 2t(1-t) y2 + t^2 y3]
    ++
    ++X n:=quadraticBezier([2.0,2.0],[4.0,4.0],[6.0,2.0])
    ++X [n(t/10.0) for t in 0..10 by 1]
    ++ A cubic Bezir curve is a simple interpolation between the
    ++ starting point, a left-middle point, a right-middle point,
    ++ and the ending point based on a parameter t.
Given a start point \( a = [x_1, y_1] \), the left-middle point \( b = [x_2, y_2] \),
the right-middle point \( c = [x_3, y_3] \) and an endpoint \( d = [x_4, y_4] \)
\[ f(t) = [(1-t)^3 x_1 + 3t(1-t)^2 x_2 + 3t^2 (1-t) x_3 + t^3 x_4,
(1-t)^3 y_1 + 3t(1-t)^2 y_2 + 3t^2 (1-t) y_3 + t^3 y_4] \]

\[ X n := \text{cubicBezier}([2.0, 2.0], [2.0, 4.0], [6.0, 4.0], [6.0, 2.0]) \]
\[ X \text{n(t/10.0) for t in 0..10 by 1} \]

---

Web: "BEZIER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BEZIER"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"BEZIER" -> "LMODULE"
"BEZIER" -> "SGROUP"

---

package BEZOUT BezoutMatrix

--- BezoutMatrix.input ---

)set break resume
)sys rm -f BezoutMatrix.output
)spool BezoutMatrix.output
)set message test on
)set message auto off
)clear all
bezoutMatrix : (UP,UP) -> M  
sylvesterMatrix : (UP,UP) -> M  
bezoutDiscriminant : UP -> R if R has commutative(*)  
bezoutResultant : (UP,UP) -> R if R has commutative(*)

---

bezoutMatrix examples

BezoutMatrix contains functions for computing resultants and discriminants using Bezout matrices.

See Also:
- )show BezoutMatrix
BezoutMatrix (BEZOUT)

Exports:
bezoutDiscriminant  bezoutMatrix  bezoutResultant  sylvesterMatrix

— package BEZOUT BezoutMatrix —

)abbrev package BEZOUT BezoutMatrix
++ Author: Clifton J. Williamson
++ Date Created: 2 August 1988
++ Date Last Updated: 3 November 1993
++ Reference: Knuth, The Art of Computer Programming, 2nd edition,
++ Description:
++ \spadtype{BezoutMatrix} contains functions for computing resultants and
++ discriminants using Bezout matrices.

BezoutMatrix(R,UP,M,Row,Col): Exports == Implementation where
R  : Ring
UP : UnivariatePolynomialCategory R
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M  : MatrixCategory(R,Row,Col)
I  => Integer
lc => leadingCoefficient

Exports ==> with
  sylvesterMatrix: (UP,UP) -> M
++ sylvesterMatrix(p,q) returns the Sylvester matrix for the two
++ polynomials p and q.
  bezoutMatrix: (UP,UP) -> M
++ bezoutMatrix(p,q) returns the Bezout matrix for the two
++ polynomials p and q.

if R has commutative("*") then
  bezoutResultant: (UP,UP) -> R
  ++ bezoutResultant(p,q) computes the resultant of the two
++ polynomials p and q by computing the determinant of a Bezout matrix.

bezoutDiscriminant: UP -> R
++ bezoutDiscriminant(p) computes the discriminant of a polynomial p
++ by computing the determinant of a Bezout matrix.

Implementation => add

---

The Sylvester matrix is used to compute the resultant of two polynomials. The Sylvester matrix is formed from the coefficients of the two polynomials. Given a polynomial with degree \( m \) and another of degree \( n \) form an \( (m + n) \times (m + n) \) matrix by filling the matrix from the upper left corner with the coefficients of the first polynomial then shifting down one row and one column to the right and filling in the coefficients starting there until they hit the right column. Starting at the next row, do the same process for the second polynomial. The determinant of this matrix is the resultant of the two polynomials.

For example, given \( a_3x^3 + a_2x^2 + a_1x + a_0 \) and \( b_2x^2 + b_1x + b_0 \) the Sylvester matrix is a \((3 + 2) \times (3 + 2)\) matrix:

\[
\begin{bmatrix}
  a_3 & a_2 & a_1 & a_0 & 0 \\
  0 & a_3 & a_2 & a_1 & a_0 \\
  b_2 & b_1 & b_0 & 0 & 0 \\
  0 & b_2 & b_1 & b_0 & 0 \\
  0 & 0 & b_2 & b_1 & b_0 \\
\end{bmatrix}
\]

The resultant of these two polynomials (assuming a leading coefficient of 1), is the product of the differences \( p_i - q_i \) between the roots of the polynomials. If there are roots in common then the product will contain a 0 and the whole equation reduces to 0. This can be used to determine if two polynomials have common roots.

In symbolic form the resultant can show the multiplicity of roots.

---

| package BEZOUT BezoutMatrix |

sylvesterMatrix(p,q) ==

  n1 := degree p; n2 := degree q; n := n1 + n2
  sylmat : M := new(n,n,0)
  minR := minRowIndex sylmat; minC := minColIndex sylmat
  maxR := maxRowIndex sylmat; maxC := maxColIndex sylmat
  p0 := p
  -- fill in coefficients of 'p'
  while not zero? p0 repeat
    coef := lc p0; deg := degree p0; p0 := reductum p0
    -- put bk = coef(p,k) in sylmat(minR + i,minC + i + (n1 - k))
    for i in 0..n2 - 1 repeat
      qsetelt_!(sylmat,minR + i,minC + n1 - deg + i,coef)
  q0 := q
bezoutMatrix(p,q) ==
-- This function computes the Bezout matrix for 'p' and 'q'.
-- One must have deg(p) >= deg(q), so the arguments are reversed
-- if this is not the case.
n1 := degree p; n2 := degree q; n := n1 + n2
n1 < n2 => bezoutMatrix(q,p)
m1 : i := n1 - 1; m2 : i := n2 - 1; m : i := n - 1
-- 'sylmat' will be a matrix consisting of the first n1 columns
-- of the standard Sylvester matrix for 'p' and 'q'
sylmat : M := new(n,n1,0)
minR := minRowIndex sylmat; minC := minColIndex sylmat
maxR := maxRowIndex sylmat; maxC := maxColIndex sylmat
p0 := p
-- fill in coefficients of 'p'
while not ground? p0 repeat
  coef := lc p0; deg := degree p0; p0 := reductum p0
  -- put bk = coef(p,k) in sylmat(minR + i,minC + i + (n1 - k))
  -- for i = 0...
  -- quit when i > m2 or when i + (n1 - k) > m1, whichever happens first
  for i in 0..min(m2,deg - 1) repeat
    qsetelt_!(sylmat,minR + i,minC + n1 - deg + i,coef)
q0 := q
-- fill in coefficients of 'q'
while not zero? q0 repeat
  coef := lc q0; deg := degree q0; q0 := reductum q0
  -- put ak = coef(q,k) in sylmat(minR + n1 + i,minC + i + (n2 - k))
  -- for i = 0...
  -- quit when i > m1 or when i + (n2 - k) > m1, whichever happens first
  -- since n2 - k >= 0, we quit when i + (n2 - k) > m1
  for i in 0..(deg + n1 - n2 - 1) repeat
    qsetelt_!(sylmat,minR + n1 + i,minC + n1 - deg + i,coef)
q0 := q
-- 'bezmat' will be the 'Bezout matrix' as described in Knuth
bezmat : M := new(n1,n1,0)
for i in 0..m2 repeat
  -- replace A_i by (b_0 A_i + ... + b_{n_2-1-i} A_{n_2 - 1-i}) -
  -- (a_0 B_i + ... + a_{n_2-1-i} B_{n_2-1-i}), as in Knuth
  bound : i := n2 - i; q0 := q
  while not zero? q0 repeat
    deg := degree q0
    if (deg < bound) then
      -- add b_deg A_{n_2 - deg} to the new A_i
      coef := lc q0
for k in minC..maxC repeat
    c := coef * qelt(sylmat,minR + m2 - i - deg,k) +
        qelt(bezmat,minR + m2 - i,k)
    qsetelt_!(bezmat,minR + m2 - i,k,c)
q0 := reductum q0
p0 := p
while not zero? p0 repeat
    deg := degree p0
    if deg < bound then
        coef := lc p0
        -- subtract a_deg B_{n_2 - deg} from the new A_i
        for k in minC..maxC repeat
            c := -coef * qelt(sylmat,minR + m - i - deg,k) +
                qelt(bezmat,minR + m2 - i,k)
            qsetelt_!(bezmat,minR + m2 - i,k,c)
        p0 := reductum p0
    for i in n2..m1 repeat for k in minC..maxC repeat
        qsetelt_!(bezmat,minR + i,k,qelt(sylmat,minR + i,k))
bezmat

if R has commutative("*") then
    bezoutResultant(f,g) == determinant bezoutMatrix(f,g)

if R has IntegralDomain then
    bezoutDiscriminant f ==
        degMod4 := (degree f) rem 4
        (degMod4 = 0) or (degMod4 = 1) =>
            bezoutResultant(f,differentiate f) exquo (lc f)) :: R
        -((bezoutResultant(f,differentiate f) exquo (lc f)) :: R)
    else
        bezoutDiscriminant f ==
            lc f = 1 =>
                degMod4 := (degree f) rem 4
                (degMod4 = 0) or (degMod4 = 1) =>
                    bezoutResultant(f,differentiate f)
                    bezoutResultant(f,differentiate f)
                    error "bezoutDiscriminant: leading coefficient must be 1"

-------

— BEZOUT.dotabb —
package BLUPPACK BlowUpPackage

--- BlowUpPackage.input ---

)set break resume
)sys rm -f BlowUpPackage.output
)spool BlowUpPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BlowUpPackage
--R
--R BlowUpPackage(K: Field,symb: List(Symbol),PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategory(#(symb),NonNegativeInteger),BLMET: BlowUpMethodCategory) is a package constructor
--R Abbreviation for BlowUpPackage is BLUPPACK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for BLUPPACK
--R
--R-------------------------------------------------------- Operations --------------------------------
--R applyTransform : (PolyRing,BLMET) -> PolyRing
--R biringToPolyRing : (DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),BLMET) -> PolyRing
--R newtonPolySlope : DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K) -> List(List(NonNegativeInteger))
--R polyRingToBlUpRing : (PolyRing,BLMET) -> DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K)
--R quadTransform : (DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),NonNegativeInteger,BLMET)
--R stepBlowUp : (DistributedMultivariatePolynomial([construct,QUOTEX,QUOTEY],K),AffinePlane(K),BLMET,K) -> 
--R
--E 1

)spool
)lisp (bye)

---

--- BlowUpPackage.help ---

====================================================================
BlowUpPackage examples
====================================================================

The following is part of the PAFF package
See Also:
  o )show BlowUpPackage

BlowUpPackage (BLUPPACK)

Exports:
  applyTransform biringToPolyRing newtonPolySlope
  polyRingToBlUpRing quadTransform stepBlowUp

— package BLUPPACK BlowUpPackage —

)abbrev package BLUPPACK BlowUpPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
  The following is part of the PAFF package
BlowUpPackage(K,symb,PolyRing,E, BLMET):Exports == Implementation where
  K : Field
  symb : List Symbol
  PolyRing : FiniteAbelianMonoidRing(K,E)
  E : DirectProductCategory(#symb,NonNegativeInteger)
  BLMET : BlowUpMethodCategory

  NNI ==> NonNegativeInteger
  RFP ==> RootsFindingPackage
  NP ==> NewtonPolygon( K, BlUpRing, E2 , #bls)
  PackPoly ==> PackageForPoly(K,BlUpRing,E2,#bls)
  bls ==> ['X,'Y]
  BlUpRing ==> DistributedMultivariatePolynomial( bls , K )
  E2 ==> DirectProduct(#bls , NNI)
AFP  ==>  AffinePlane(K)

blowUpRec  ==>  Record(recTransStr:BlUpRing,recPoint:AFP,recChart:BLMET, _
  definingExtension:K)
blowUpReturn ==> Record(mult:NonNegativeInteger,subMult: NNI, _
  blUpRec:List(blowUpRec))
recStr    ==>  Record( sM: NNI , blRec:List blowUpRec)

Exports ==>  with

  applyTransform: (PolyRing,BLMET) -> PolyRing
  ++ quadTransform(pol,chart) apply the quadratique transformation to
  ++ pol specified by chart which consist of 3 integers. The last one
  ++ indicates which variables is set to 1, the first one indicates
  ++ which variable remains unchange, and the second one indicates
  ++ which variable on which the transformation is applied.
  ++ For example, [2,3,1] correspond to the following:
  ++ x -> 1, y -> y, z -> yz (here the variable are [x,y,z] in BlUpRing).

  quadTransform: (BlUpRing,NNI,BLMET) -> BlUpRing -- CHH
  ++ quadTransform(pol,n,chart) apply the quadratique transformation
  ++ to pol specified by chart has in quadTransform(pol,chart) and
  ++ extract x**n to it, where x is the variable specified by the
  ++ first integer in chart (blow-up exceptional coordinate).

  stepBlowUp:(BlUpRing,AFP,BLMET,K) -> blowUpReturn -- CHH
  ++ stepBlowUp(pol,pt,n) blow-up the point pt on the curve defined
  ++ by pol in the affine neighbourhood specified by n.

  newtonPolySlope: BlUpRing -> List List(NNI)

  polyRingToBlUpRing: (PolyRing, BLMET) -> BlUpRing

  biringToPolyRing: (BlUpRing, BLMET) -> PolyRing

Implementation ==> add

import BlUpRing
import AFP
import RFP(K)
import PackPoly
import NP

makeAff( l:List(K) , chart: BLMET ):AFP ==
  (excepCoord chart) = 1 => affinePoint( l )$AFP
  affinePoint( reverse l )$AFP

blowExp: (E2, NNI, BLMET ) -> E2

maxOf: (K,K) -> K
getStrTrans: ( BlUpRing , List BlUpRing , BLMET, K ) -> recStr

stepBlowUp(crb:BlUpRing,pt:AFP,chart:BLMET,actualExtension:K) ==
  -- next is with Hamburger-Noether method
  BLMET has HamburgerNoether =>
    nV:Integer:= chartCoord chart
    crbTrans:BlUpRing:=translate(crb, list(pt))$PackPoly
    newtPol:= newtonPolygon( crbTrans, quotValuation chart, _
      ramifMult chart, type chart )$NP
    multPt:= multiplicity(newtPol)$NP
    one?(multPt) =>
      [multPt, 0 , empty() ]$blowUpReturn
    listOfGetTr:List recStr:=
      [ getStrTrans( crbTrans , edge , chart , actualExtension )
        for edge in newtPol ]
    lsubM: List NNI := [ ll.sM for ll in listOfGetTr]
    subM := reduce( "+" , lsubM )
    llistOfRec: List List blowUpRec := [ ll.blRec for ll in listOfGetTr]
    listOfRec:= concat llistOfRec
    [ multPt, subM ,listOfRec]$blowUpReturn
  -- next is with usual quadratic transform.

  BLMET has QuadraticTransform =>
    nV:Integer:= chartCoord chart
    lpt:List(K) := list(pt)$AFP
    crbTrans:=translate(crb,lpt)
    minForm:=minimalForm(crbTrans)
    multPt:=totalDegree( minForm)$PackPoly
    listRec:List(blowUpRec):=empty()
    one?(multPt) => [multPt, 0 , listRec]$blowUpReturn
    -- now pt is singular !!!!
    lstInd:=[i::PositiveInteger for i in 1..2 ]
    -- la ligne suivante fait un choix judicieux pour minimiser le
    -- degre' du transforme' stricte.
    if degree( crbTrans , 2 )$PackPoly < degree( crbTrans , 1 )$PackPoly _
      then lstInd := reverse lstInd
    ptInf:List(K):=[0$K,0$K]
    laCarte:BLMET:=
      ([last(lstInd), first(lstInd),nV] @ List Integer) :: BLMET
    laCarteInf:BLMET:=
      ([first(lstInd),last(lstInd),nV] @ List Integer ) :: BLMET
    transStricte :=quadTransform(crbTrans,multPt,laCarte)
    transStricteInf:=quadTransform(crbTrans,multPt,laCarteInf)
    listPtsSingEcl:List(List(AFP)):=empty()
    transStricteZero:BlUpRing:= replaceVarByOne(minForm,excepCoord laCarte)
    recOfZeros:=
      distinguishedRootsOf(univariate(transStricteZero)$PackPoly ,_
        actualExtension )$RFP(K)
    degExt:=recOfZeros.extDegree


"one?(degExt) =>
    print("You need an extension of degree")::OutputForm
    print(degExt::OutputForm)
    error("Have a nice day")
listPtsSingEcl:=[makeAff([0$K,a]::List(K),laCarte) _
    for a in recOfZeros.zeros]
listRec:=[
    [ transStricte,_
        ptS,laCarte,_
        maxOf(a,actualExtension)]$blowUpRec_
    for ptS in listPtsSingEcl_
        for a in recOfZeros.zeros]
if zero?(constant(transStricteInf))$K then
    listRec:= concat(listRec,[transStricteInf,_
        affinePoint(ptInf)$AFP,_
        laCarteInf,_
        actualExtension]$blowUpRec)
empty?(listRec) =>
    error "Something is very wrong in blowing up!!!!!!"
    [multPt, 0 ,listRec]$blowUpReturn
error "Desingularisation is not implemented for the blowing up method chosen, see BlowingUpMethodCategory."

getStrTrans(crb, inedge, actChart, actualExtension ) ==
edge:= copy inedge
s := slope(edge)$NP
sden:Integer
snum:Integer
i1:Integer
i2:Integer
if s.type case "right" then
    sden:= s.base
    snum:=s.height
    i1:=1
    i2:=2
else -- interchage les roles de X et Y.
    sden:= s.height
    snum:= s.base
    i1:=2
    i2:=1
edge := copy reverse inedge
ee := entries( degree first edge) pretend List Integer
euclq: Integer
if one?(snum) then
    euclq:=1
else
    euclq := s.quotient
-- sMult est la somme des multiplicites des points infiniment
-- voisins par une trans. quadratique
sMult: NNI := ( ( euclq - 1 ) * ee.i2 ) pretend NNI
-- extMult est egal a la plus grande puissance de X que l'on peut
-- extraire de la transformée.

\( \text{extMult} := (\text{ee.i1} + \text{ee.i2} \times \text{euclq}) \) pretend \( \text{NonNegativeInteger} \)

\( \text{ch} : \text{BLMET} \)

\( \text{trStr} : \text{BlUpRing} \)

\( \text{listBlRec} : \text{List blowUpRec} \)

\( ^{\text{zero?}}(s.\text{reste}) => \)

\( \text{ch} := \text{createHN}(i1, i2, \text{chartCoord actChart}, \text{euclq}, s.\text{reste}, \_\endgroup \)

\( \text{false}, \text{s.type}) \$\text{BLMET} \)

\( \text{trStr} := \text{quadTransform}(\text{crb}, \text{extMult}, \text{ch}) \)

\( \text{listBlRec} := [ [\text{trStr}, \text{origin}()]$\text{AFP}, \text{ch}, \text{actualExtension }]$\text{blowUpRec} ] \)

\( [s.\text{Mult}, \text{listBlRec}]$\text{recStr} \)

\( \text{polEdge} := \text{reduce}(\text{"+"}, \text{edge}) \)

\( \text{unipol} := \text{univariate}(\text{replaceVarByOne}(\text{polEdge}, i1)$\text{PackPoly}) \$\text{PackPoly} \)

\( \text{recOfZeros} := \text{distinguishedRootsOf}(\text{unipol}, \text{actualExtension}) \$\text{RFP(K)} \)

\( \text{degExt} := \text{recOfZeros}.\text{extDegree} \)

\( ^{\text{one?}}(\text{degExt}) => \)

\( \text{print}((\text{"You need an extension of degree"})::\text{OutputForm}) \)

\( \text{print}(\text{degExt}::\text{OutputForm}) \)

\( \text{error}((\text{"Have a nice day"}) \)

\( \text{empty? listOfZeros} => \_ \)

\( \text{error} \ " \text{The curve is not absolutely irreducible since the Newton polygon has no sides} \" \)

\( \text{ch} := \_ \)

\( \text{createHN}(i1, i2, \text{chartCoord actChart}, \text{euclq}, 0, \text{false}, \text{s.type}) \$\text{BLMET} \)

\( \text{lsTr} : \text{BlUpRing} := \text{quadTransform}(\text{crb}, \text{extMult}, \text{ch}) \)

\( \text{lAff} : \text{List AFP} := [\text{makeAff}([0$\text{K}, z])::\text{List K} \) for \text{z in listOfZeros} ]

\( \text{listBlRec} := [ [\text{lsTr}, \text{p}, \text{ch}, \text{maxOf} \text{actualExtension}, \text{z}]$\text{blowUpRec} ]

\( \text{for} \text{p in lAff} \) \text{for} \text{z in listOfZeros} ]

\( [s.\text{Mult}, \text{listBlRec}]$\text{recStr} \)

\( \text{blowExp(exp, mult, chart)} == -- \text{CHH} \)

\( \text{zero?}((\text{exceptCoord chart}) \) => \text{exp} \)

\( \text{lexp} : \text{List NNI} := \text{parts}(\text{exp}) \)

\( \text{ch1} : \text{Integer} := \text{exceptCoord chart} \)

\( \text{ch2} : \text{Integer} := \text{transCoord chart} \)

\( \text{e1} : \text{Integer} := \text{lexp}(\text{ch1}) \) pretend \( \text{Integer} \)

\( \text{e2} : \text{Integer} := \text{lexp}(\text{ch2}) \) pretend \( \text{Integer} \)

\( \text{quotVal} : \text{Integer} := \text{quotValuation chart} \)

\( \text{lbexp} := [0, 0] :: \text{List} \text{(NNI)} \)

\( \text{lbexp}(\text{ch1}) := (\text{e1} + \text{quotVal} \times \text{e2} - \text{mult}) \) pretend \( \text{NonNegativeInteger} \)

\( \text{lbexp}(\text{ch2}) := \text{lexp}(\text{ch2}) \)

\( \text{directProduct}(\text{vector}(\text{lbexp})$\text{Vector} \text{(NNI)})$\text{E2} \)

\( \text{quadTransform}(\text{pol}, \text{mult}, \text{chart}) == -- \text{CHH} \)

\( \text{mapExponents}(\text{blowExp}(\#1, \text{mult}, \text{chart}), \text{pol}) \)

\( \text{polyRingToBlUpRing}(\text{pol}, \text{chart}) == \)

\( \text{zero?} \text{pol} => 0 \)

\( \text{lc} := \text{leadingCoefficient} \text{pol} \)

\( \text{d} := \text{entries degree pol} \)
package BOUNDZRO BoundIntegerRoots

-- BoundIntegerRoots.input --

)set break resume
)sys rm -f BoundIntegerRoots.output
)spool BoundIntegerRoots.output
BoundIntegerRoots provides functions to find lower bounds on the integer roots of a polynomial.

See Also:
- )show BoundIntegerRoots
BOUNDZRO

PFECAT

Exports:
integerBound

— package BOUNDZRO BoundIntegerRoots —

)abbrev package BOUNDZRO BoundIntegerRoots
++ Author: Manuel Bronstein
++ Date Created: 11 March 1991
++ Date Last Updated: 18 November 1991
++ Description:
++ \spadtype{BoundIntegerRoots} provides functions to
++ find lower bounds on the integer roots of a polynomial.

BoundIntegerRoots(F, UP):Exports == Implementation where
F : Join(Field, RetractableTo Fraction Integer)
UP : UnivariatePolynomialCategory F
Z ==> Integer
Q ==> Fraction Z
K ==> Kernel F
UPQ ==> SparseUnivariatePolynomial Q
ALGOP ==> "%alg"

Exports == with
integerBound: UP -> Z
  ++ integerBound(p) returns a lower bound on the negative integer
  ++ roots of p, and 0 if p has no negative integer roots.

Implementation == add
import RationalFactorize(UPQ)
import UnivariatePolynomialCategoryFunctions2(F, UP, Q, UPQ)

qboun : (UP, UPQ) -> Z
zroot1 : UP -> Z
qzroot1 : UPQ -> Z
negint : Q -> Z
-- returns 0 if p has no integer root < 0, its negative integer root otherwise
qzroot1 p == negint(- leadingCoefficient(reductum p) / leadingCoefficient p)

-- returns 0 if p has no integer root < 0, its negative integer root otherwise
zroot1 p ==
  z := - leadingCoefficient(reductum p) / leadingCoefficient p
  (r := retractIfCan(z)@Union(Q, "failed")) case Q => negint(r::Q)
  0

-- returns 0 if r is not a negative integer, r otherwise
negint r ==
  ((u := retractIfCan(r)@Union(Z, "failed")) case Z) and (u::Z < 0) => u::Z
  0

if F has ExpressionSpace then
  bringDown: F -> Q
  -- the random substitution used by bringDown is NOT always a ring-homorphism
  -- (because of potential algebraic kernels), but is ALWAYS a Z-linear map.
  -- this guarantees that bringing down the coefficients of (x + n) q(x) for an
  -- integer n yields a polynomial h(x) which is divisible by x + n
  -- the only problem is that evaluating with random numbers can cause a
  -- division by 0. We should really be able to trap this error later and
  -- reevaluate with a new set of random numbers     MB 11/91
  bringDown f ==
    t := tower f
    retract eval(f, t, [random()$Q :: F for k in t])

integerBound p ==
  -- one? degree p => zroot1 p
  (degree p) = 1 => zroot1 p
  q1 := map(bringDown, p)
  q2 := map(bringDown, p)
  qbound(p, gcd(q1, q2))

else
  integerBound p ==
    -- one? degree p => zroot1 p
    (degree p) = 1 => zroot1 p
    qbound(p, map((z1:F):Q +-> retract(z1)@Q, p))

-- we can probably do better here (i.e. without factoring)
qbound(p, q) ==
  bound:Z := 0
  for rec in factors factor q repeat
    -- if one?(degree(rec.factor)) and ((r := qzroot1(rec.factor)) < bound)
    if (degree(rec.factor)) = 1 and ((r := qzroot1(rec.factor)) < bound)
      and zero? p(r::Q::F) then bound := r
package BRILL BrillhartTests

-- BrillhartTests.input --

)set break resume
)sys rm -f BrillhartTests.output
)spool BrillhartTests.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show BrillhartTests
--R
--R BrillhartTests(UP: UnivariatePolynomialCategory(Integer)) is a package constructor
--R Abbreviation for BrillhartTests is BRILL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for BRILL
--R
--R----------------------------------- Operations -----------------------------------
--R noLinearFactor? : UP -> Boolean
--R brillhartIrreducible? : UP -> Boolean
--R brillhartIrreducible? : (UP,Boolean) -> Boolean
--R brillhartTrials : () -> NonNegativeInteger
--R brillhartTrials : NonNegativeInteger -> NonNegativeInteger
--R
--E 1

)spool
)lisp (bye)
BrillhartTests (BRILL)

Exports:
noLinearFactor? brillhartIrreducible? brillhartTrials

—— package BRILL BrillhartTests ——

)abbrev package BRILL BrillhartTests
++ Author: Frederic Lehobey, James H. Davenport
++ Date Created: 28 June 1994
++ Date Last Updated: 11 July 1997
++ References:
++ [1] John Brillhart, Note on Irreducibility Testing,
++ [3] John Brillhart, On the Euler and Bernoulli polynomials,
++ Description:
++ This package has no description

BrillhartTests(UP): Exports == Implementation where
N ==> NonNegativeInteger
Z ==> Integer
UP: UnivariatePolynomialCategory Z
Exports ==> with
  brillhartIrreducible?: UP -> Boolean -- See [1]
  ++ brillhartIrreducible?(p) returns \spad{true} if p can be shown to be
  ++ irreducible by a remark of Brillhart, \spad{false} is inconclusive.
  brillhartIrreducible?: (UP,Boolean) -> Boolean -- See [1]
  ++ brillhartIrreducible?(p,noLinears) returns \spad{true} if p can be
  ++ shown to be irreducible by a remark of Brillhart, \spad{false} else.
  ++ If noLinears is \spad{true}, we are being told p has no linear factors
  ++ \spad{false} does not mean that p is reducible.
  brillhartTrials: () -> N
  ++ brillhartTrials() returns the number of tests in
  ++ \spadfun{brillhartIrreducible?}.
  brillhartTrials: N -> N
  ++ brillhartTrials(n) sets to n the number of tests in
  ++ \spadfun{brillhartIrreducible?} and returns the previous value.
  noLinearFactor?: UP -> Boolean -- See [3] p. 47
  ++ noLinearFactor?(p) returns \spad{true} if p can be shown to have no
  ++ linear factor by a theorem of Lehmer, \spad{false} else. I insist on
  ++ the fact that \spad{false} does not mean that p has a linear factor.

Implementation ==> add

import GaloisGroupFactorizationUtilities(Z,UP,Float)

squaredPolynomial(p:UP):Boolean ==
  d := degree p
  d = 0 => true
  odd? d => false
  squaredPolynomial reductum p

primeEnough?(n:Z,b:Z):Boolean ==
  -- checks if n is prime, with the possible exception of
  -- factors whose product is at most b
  import Float
  bb: Float := b::Float
  for i in 2..b repeat
    while (d:= n exquo i) case Integer repeat
      n:=d::Integer
bb := bb / i::Float
bb < 1$Float => return false
--- we over-divided, so it can't be prime
prime? n

brillharttrials: N := 6
brillhartTrials():N == brillharttrials

brillhartTrials(n:N):N ==
  (brillharttrials,n) := (n,brillharttrials)
n
brillhartIrreducible?(p:UP):Boolean ==
brillhartIrreducible?(p,noLinearFactor? p)
  zero? brillharttrials => false
  origBound := (largeEnough := rootBound(p)+1)
  -- see remarks 2 and 4
  even0 := even? coefficient(p,0)
  even1 := even? p(1)
  polyx2 := squaredPolynomial(p)
  prime? p(largeEnough) => true
  not polyx2 and prime? p(-largeEnough) => true
--
  one? brillharttrials => false
  (brillharttrials = 1) => false
  largeEnough := largeEnough+1
  primeEnough?(p(largeEnough),if noLinears then 4 else 2) => true
  not polyx2 and
  primeEnough?(p(-largeEnough),if noLinears then 4 else 2) => true
  if odd? largeEnough then
    if even0 then largeEnough := largeEnough+1
    else
      if even1 then largeEnough := largeEnough+1
      count :=(if polyx2 then 2 else 1)*(brillharttrials-2)+largeEnough
      for i in (largeEnough+1)..count repeat
        small := if noLinears then (i-origBound)**2 else (i-origBound)**2
        primeEnough?(p(i),small) => return true
        not polyx2 and primeEnough?(p(-i),small) => return true
      false

noLinearFactor?(p:UP):Boolean ==
  (odd? leadingCoefficient p) and (odd? coefficient(p,0)) and (odd? p(1))

— BRILL.dotabb —
"BRILL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=BRILL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"BRILL" -> "PFECAT"
package CARTEN2 CartesianTensorFunctions2

— CartesianTensorFunctions2.input —

)set break resume
)sys rm -f CartesianTensorFunctions2.output
)spool CartesianTensorFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CartesianTensorFunctions2
--R
--R CartesianTensorFunctions2(minix: Integer,dim: NonNegativeInteger,S: CommutativeRing,T$: CommutativeRing)
--R Abbreviation for CartesianTensorFunctions2 is CARTEN2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CARTEN2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((S -> T$),CartesianTensor(minix,dim,S)) -> CartesianTensor(minix,dim,T$)
--R reshape : (List(T$),CartesianTensor(minix,dim,S)) -> CartesianTensor(minix,dim,T$)
--R
--E 1

)spool
)lisp (bye)
This package provides functions to enable conversion of tensors given conversion of the components.

See Also:
- )show CartesianTensorFunctions2

---

CartesianTensorFunctions2 (CARTEN2)

Exports:
map reshape

---

)abbrev package CARTEN2 CartesianTensorFunctions2
++ Author: Stephen M. Watt
++ Date Created: December 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This package provides functions to enable conversion of tensors given conversion of the components.

CartesianTensorFunctions2(minix, dim, S, T): CTPcat == CTPdef where
  minix: Integer
  dim: NonNegativeInteger
  S, T: CommutativeRing
CS ==> CartesianTensor(minix, dim, S)
CT ==> CartesianTensor(minix, dim, T)

CTPcat == with
  reshape: (List T, CS) -> CT
    ++ reshape(lt,ts) organizes the list of components lt into
    ++ a tensor with the same shape as ts.
map: (S->T, CS) -> CT
    ++ map(f,ts) does a componentwise conversion of the tensor ts
    ++ to a tensor with components of type T.

CTPdef == add
  reshape(1, s) == unravel 1
  map(f, s)    == unravel [f e for e in ravel s]

---

— CARTEN2.dotabb —

"CARTEN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CARTEN2"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"CARTEN2" -> "BMODULE"

---

package CHVAR ChangeOfVariable

— ChangeOfVariable.input —

)set break resume
)sys rm -f ChangeOfVariable.output
)spool ChangeOfVariable.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ChangeOfVariable
--R
--R ChangeOfVariable(F: UniqueFactorizationDomain,UP: UnivariatePolynomialCategory(F),UPUP: UnivariatePolynomialCategory(Fraction(UP))) is a package constructor
--R Abbreviation for ChangeOfVariable is CHVAR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CHVAR
--R
--R----------------------------------- Operations -----------------------------------
CHAPTER 4. CHAPTER C

--R goodPoint : (UPUP,UPUP) -> F
--R chvar : (UPUP,UPUP) -> Record(func: UPUP,poly: UPUP,c1: Fraction(UP),c2: Fraction(UP),deg: NonNegativeInteger)
--R eval : (UPUP,Fraction(UP),Fraction(UP)) -> UPUP
--R mkIntegral : UPUP -> Record(coef: Fraction(UP),poly: UPUP)
--R radPoly : UPUP -> Union(Record(radicand: Fraction(UP),deg: NonNegativeInteger),"failed")
--R rootPoly : (Fraction(UP),NonNegativeInteger) -> Record(exponent: NonNegativeInteger,coef: Fraction(UP))
--R 1

E 1

)spool
)lisp (bye)

—— ChangeOfVariable.help ——

====================================================================
ChangeOfVariable examples
====================================================================

Tools to send a point to infinity on an algebraic curve.

See Also:
  o )show ChangeOfVariable

——

ChangeOfVariable (CHVAR)

Exports:
  chvar  eval  goodPoint  mkIntegral
  radPoly  rootPoly

—— package CHVAR ChangeOfVariable ——
++)abbrev package CHVAR ChangeOfVariable
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 22 Feb 1990
++ Description:
++ Tools to send a point to infinity on an algebraic curve.

ChangeOfVariable(F, UP, UPUP): Exports == Implementation where
  F : UniqueFactorizationDomain
  UP : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP

  N ==> NonNegativeInteger
  Z ==> Integer
  Q ==> Fraction Z
  RF ==> Fraction UP

Exports ==> with
  mkIntegral: UPUP -> Record(coef:RF, poly:UPUP)
    ++ mkIntegral(p(x,y)) returns \spad{[c(x), q(x,z)]} such that
    ++ \spad{z = c * y} is integral.
    ++ The algebraic relation between x and y is \spad{p(x, y) = 0}.
    ++ The algebraic relation between x and z is \spad{q(x, z) = 0}.

  radPoly : UPUP -> Union(Record(radicand:RF, deg:N), "failed")
    ++ radPoly(p(x, y)) returns \spad{[c(x), n]} if p is of the form
    ++ \spad{y^n - c(x)}, "failed" otherwise.

  rootPoly : (RF, N) -> Record(exponent: N, coef:RF, radicand:UP)
    ++ rootPoly(g, n) returns \spad{[m, c, P]} such that
    ++ \spad{c * g**(1/n) = P**(1/m)}
    ++ thus if \spad{y^n = g}, then \spad{z^m = P}.
    ++ where \spad{z = c * y}.

  goodPoint : (UPUP,UPUP) -> F
    ++ goodPoint(p, q) returns an integer a such that a is neither
    ++ a pole of \spad{p(x,y)} nor a branch point of \spad{q(x,y) = 0}.

  eval : (UPUP, RF, RF) -> UPUP
    ++ eval(p(x,y), f(x), g(x)) returns \spad{p(f(x), y * g(x))}.

  chvar : (UPUP,UPUP) -> Record(func:UPUP,poly:UPUP,c1:RF,c2:RF,deg:N)
    ++ chvar(f(x,y), p(x,y)) returns
    ++ \spad{[g(z,t), q(z,t), c1(z), c2(z), n]}
    ++ such that under the change of variable
    ++ \spad{x = c1(z)}, \spad{y = t * c2(z)}.
    ++ one gets \spad{f(x,y) = g(z,t)}.
    ++ The algebraic relation between x and y is \spad{p(x, y) = 0}.
    ++ The algebraic relation between z and t is \spad{q(z, t) = 0}.

Implementation ==> add
  import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)
good? : (F, UP, UP) -> Boolean
infIntegral? : (UPUP, UPUP) -> Boolean

eval(p, x, y) == map(s -> s(x), p) monomial(y, 1)
good?(a, p, q) == p(a) ^= 0 and q(a) ^= 0

c := d := squareFreePart a
q := clearDenominator q monomial(inv(d::RF), 1)
while not ground?(a := retract(leadingCoefficient q)@UP) repeat
    c := c * (d := gcd(a, d))
    q := clearDenominator q monomial(inv(d::RF), 1)
RPrim(c, a, q)

RPrim(c, a, q) ==
-- one? a => [c::RF, q]
(a = 1) => [c::RF, q]
[(a * c)::RF, clearDenominator q monomial(inv(a::RF), 1)]

-- always makes the algebraic integral, but does not send a point to infinity
-- if the integrand does not have a pole there (in the case of an nth-root)
chvar(f, modulus) ==
r1 := mkIntegral modulus
f1 := f monomial(r1inv := inv(r1.coef), 1)
infIntegral?(f1, r1.poly) =>
[f1, r1.poly, monomial(1,1)$UP :: RF, r1inv,degree(retract(r1.coef)@UP)]
x := (a:= goodPoint(f1,r1.poly))::UP::RF + inv(monomial(1,1)::RF)
r2c:= retract((r2 := mkIntegral map(s+->s(x), r1.poly)).coef)@UP
t := inv((monomial(1, 1)$UP - a::UP)::RF)
[- inv(monomial(1, 2)$UP :: RF) * eval(f1, x, inv(r2.coef)),
r2.poly, t, r1.coef * r2c t, degree r2c]

-- returns true if y is an n-th root, and it can be guaranteed that p(x,y)dx
-- is integral at infinity
-- expects y to be integral.
infIntegral?(p, modulus) ==
(r := radPoly modulus) case "failed" => false
ninv := inv(r.deg::Q)
degy:Q := degree(retract(r.radicand)@UP) * ninv
degp:Q := 0
while p ^= 0 repeat
    c := leadingCoefficient p
degp := max(degp,
        (2 + degree(numer c)::Z - degree(denom c)::Z)::Q + degree(p) * degy)
p := reductum p
degp <= ninv

mkIntegral p ==
package CPIMA CharacteristicPolynomialInMonogenicalAlgebra

(r := radPoly p) case "failed" => algPoly p
rp := rootPoly(r.radicand, r.deg)
[rp.coef, monomial(1, rp.exponent)$UPUP - rp.radicand::RF::UPUP]

goodPoint(p, modulus) ==
q :=
(r := radPoly modulus) case "failed" =>
retract(resultant(modulus, differentiate modulus))@UP
retract(r.radicand)@UP
d := commonDenominator p
for i in 0.. repeat
good?(a := i::F, q, d) => return a
good?(-a, q, d) => return -a

radPoly p ==
(r := retractIfCan(reductum p)@Union(RF, "failed")) case "failed" => "failed"
[- (r::RF), degree p]

-- we have y**m = g(x) = n(x)/d(x), so if we can write
-- (n(x) * d(x)**(m-1)) ** (1/m) = c(x) * P(x) ** (1/n)
-- then z**q = P(x) where z = (d(x) / c(x)) * y

rootPoly(g, m) ==
zero? g => error "Should not happen"
pr := nthRoot(squareFree((numer g) * (d := denom g) ** (m-1)::N),
[pr.exponent, d / pr.coef, */(pr.radicand)]

package CPIMA CharacteristicPolynomialInMonogenicalAlgebra

--- CharacteristicPolynomialInMonogenicalAlgebra.input ---

)set break resume
--S 1 of 1
)show CharacteristicPolynomialInMonogenicalAlgebra
--R
--R CharacteristicPolynomialInMonogenicalAlgebra(R: CommutativeRing, PolR: UnivariatePolynomialCategory(R))
--R Abbreviation for CharacteristicPolynomialInMonogenicalAlgebra is CPIMA
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CPIMA
--R
--R-------------------------------- Operations --------------------------------
--R characteristicPolynomial : E -> PolR
--R
--E 1

)spool
)lisp (bye)

——— CharacteristicPolynomialInMonogenicalAlgebra.help ———

====================================================================
CharacteristicPolynomialInMonogenicalAlgebra examples
====================================================================

This package implements characteristicPolynomials for monogenic algebras using resultants

See Also:
  o )show CharacteristicPolynomialInMonogenicalAlgebra

———
CharacteristicPolynomialInMonogenousAlgebra (CPIMA)

Exports:
characteristicPolynomial

package CPIMA CharacteristicPolynomialInMonogenousAlgebra

)abbrev package CPIMA CharacteristicPolynomialInMonogenousAlgebra
++ Author: Claude Quitte
++ Date Created: 10/12/93
++ Description:
++ This package implements characteristic polynomials for monogenic algebras
++ using resultants

CharacteristicPolynomialInMonogenousAlgebra(R : CommutativeRing,
   PolR : UnivariatePolynomialCategory(R),
   E : MonogenicAlgebra(R, PolR)): with
characteristicPolynomial : E -> PolR
  ++ characteristicPolynomial(e) returns the characteristic polynomial
  ++ of e using resultants

== add
   Pol ==> SparseUnivariatePolynomial

import UnivariatePolynomialCategoryFunctions2(R, PolR, PolR, Pol(PolR))
XtoY(Q : PolR) : Pol(PolR) == map(x+->monomial(x, 0), Q)

P : Pol(PolR) := XtoY(definingPolynomial()$E)
X : Pol(PolR) := monomial(monomial(1, 1)$PolR, 0)

characteristicPolynomial(x : E) : PolR ==
   Qx : PolR := lift(x)
   -- on utilise le fait que resultant_Y (P(Y), X - Qx(Y))
   return resultant(P, X - XtoY(Qx))
package CHARPOL CharacteristicPolynomialPackage

-- CharacteristicPolynomialPackage.input --

)set break resume
)sys rm -f CharacteristicPolynomialPackage.output
)spool CharacteristicPolynomialPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show CharacteristicPolynomialPackage

-- R
-- R CharacteristicPolynomialPackage(R: CommutativeRing) is a package constructor
-- R Abbreviation for CharacteristicPolynomialPackage is CHARPOL
-- R This constructor is exposed in this frame.
-- R Issue )edit bookvol10.4.pamphlet to see algebra source code for CHARPOL
-- R
-- R ------------------------------- Operations -------------------------------
-- R characteristicPolynomial : (Matrix(R),R) -> R
-- R
-- E 1

)spool
)lisp (bye)

-- CharacteristicPolynomialPackage.help --

====================================================================
CharacteristicPolynomialPackage examples
====================================================================
This package provides a characteristicPolynomial function for any matrix over a commutative ring.

See Also:
o )show CharacteristicPolynomialPackage

CharacteristicPolynomialPackage (CHARPOL)

Exports:
characteristicPolynomial

— package CHARPOL CharacteristicPolynomialPackage —

)abbrev package CHARPOL CharacteristicPolynomialPackage
++ Author: Barry Trager
++ Description:
++ This package provides a characteristicPolynomial function
++ for any matrix over a commutative ring.

CharacteristicPolynomialPackage(R:CommutativeRing):C == T where
PI ==> PositiveInteger
M ==> Matrix R
C == with
  characteristicPolynomial: (M, R) -> R
  ++ characteristicPolynomial(m,r) computes the characteristic
  ++ polynomial of the matrix m evaluated at the point r.
  ++ In particular, if r is the polynomial 'x, then it returns
  ++ the characteristic polynomial expressed as a polynomial in 'x.
T == add
--- characteristic polynomial ---

characteristicPolynomial(A:M,v:R) : R ==
  dimA :PI := (nrows A):PI
  dimA ^= ncols A => error " The matrix is not square"
  B:M:=zero(dimA,dimA)
  for i in 1..dimA repeat
    for j in 1..dimA repeat B(i,j):=A(i,j)
    B(i,i) := B(i,i) - v
  determinant B

---

— CHARPOL.dotabb —

"CHARPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CHARPOL"]
"BMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BMODULE"]
"CHARPOL" -> "BMODULE"

package IBACHIN ChineseRemainderToolsForIntegralBases

— ChineseRemainderToolsForIntegralBases.input —

)set break resume
)sys rm -f ChineseRemainderToolsForIntegralBases.output
)spool ChineseRemainderToolsForIntegralBases.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ChineseRemainderToolsForIntegralBases

--R
--R ChineseRemainderToolsForIntegralBases(K: FiniteFieldCategory,R: UnivariatePolynomialCategory)
--R Abbreviation for ChineseRemainderToolsForIntegralBases is IBACHIN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IBACHIN
--R
--R----------------------------- Operations -----------------------------
--R chineseRemainder : (List(UP),List(Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R)))),
--R factorList : (K,NonNegativeInteger,NonNegativeInteger,NonNegativeInteger) -> List(SparseUniv
--R listConjugateBases : (Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R)),NonNegativeIn
--R
ChineseRemainderToolsForIntegralBases (IBACHIN)

Exports:
chineseRemainder  factorList  listConjugateBases

---

)abbrev package IBACHIN ChineseRemainderToolsForIntegralBases
++ Author: Clifton Williamson
++ Date Created: 9 August 1993
++ Date Last Updated: 3 December 1993
++ Description:
++ This package has no description

ChineseRemainderToolsForIntegralBases(K,R,UP): Exports == Implementation where
  K : FiniteFieldCategory
  R : UnivariatePolynomialCategory K
  UP : UnivariatePolynomialCategory R

  DDFACT ==> DistinctDegreeFactorize
  I   ==> Integer
  L   ==> List
  L2  ==> ListFunctions2
  Mat ==> Matrix R
  NNI ==> NonNegativeInteger
  PI  ==> PositiveInteger
  Q   ==> Fraction R
  SAE ==> SimpleAlgebraicExtension
  SUP ==> SparseUnivariatePolynomial
  SUP2 ==> SparseUnivariatePolynomialFunctions2
  Result ==> Record(basis: Mat, basisDen: R, basisInv: Mat)

Exports == with
  factorList: (K,NNI,NNI,NNI) -> L SUP K
++ factorList(k,n,m,j) undocumented
  listConjugateBases: (Result,NNI,NNI) -> List Result
++ listConjugateBases(bas,q,n) returns the list
++ \spad{[bas,bas^Frob,bas^(Frob^2),...bas^(Frob^(n-1))]}, where
++ \spad{Frob} raises the coefficients of all polynomials
++ appearing in the basis \spad{bas} to the \spad{q}th power.

chineseRemainder: (List UP, List Result, NNI) -> Result
++ chineseRemainder(lu,lr,n) undocumented

Implementation ==> add
  import ModularHermitianRowReduction(R)
  import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

  applyFrobToMatrix: (Matrix R,NNI) -> Matrix R
applyFrobToMatrix(mat,q) ==
  -- raises the coefficients of the polynomial entries of 'mat'
  -- to the qth power
  m := nrows mat; n := ncols mat
  ans : Matrix R := new(m,n,0)
  for i in 1..m repeat for j in 1..n repeat
    qsetelt!(ans,i,j,map((k1:K):K +-> k1 ** q,qelt(mat,i,j)))
  ans

  listConjugateBases(bas,q,n) ==
    outList : List Result := list bas
    b := bas.basis; bInv := bas.basisInv; bDen := bas.basisDen
for i in 1..(n-1) repeat
  b := applyFrobToMatrix(b,q)
  bInv := applyFrobToMatrix(bInv,q)
  bDen := map((k1:K):K +-> k1 ** q,bDen)
  newBasis : Result := [b,bDen,bInv]
  outList := concat(newBasis,outList)
  reverse_! outList

factorList(a,q,n,k) ==
  coef : SUP K := monomial(a,0); xx : SUP K := monomial(1,1)
  outList : L SUP K := list((xx - coef)**k)
  for i in 1..(n-1) repeat
    coef := coef ** q
    outList := concat((xx - coef)**k,outList)
    reverse_! outList

basisInfoToPolys: (Mat,R,R) -> L UP
basisInfoToPolys(mat,lcm,den) ==
  n := nrows(mat) :: I; n1 := n - 1
  outList : L UP := empty()
  for i in 1..n repeat
    pp : UP := 0
    for j in 0..n1 repeat
      pp := pp + monomial((lcm quo den) * qelt(mat,i,j+1),j)
    outList := concat(pp,outList)
  reverse_! outList

basesToPolyLists: (L Result,R) -> L L UP
basesToPolyLists(basisList,lcm) ==
  [basisInfoToPolys(b.basis,lcm,b.basisDen) for b in basisList]

OUT ==> OutputForm

approximateExtendedEuclidean: (UP,UP,R,NNI) -> Record(coef1:UP,coef2:UP)
approximateExtendedEuclidean(f,g,p,n) ==
  -- f and g are monic and relatively prime (mod p)
  -- function returns [coef1,coef2] such that
  -- coef1 * f + coef2 * g = 1 (mod p^n)
  sae := SAE(K,R,p)
  fSUP : SUP R := makeSUP f; gSUP : SUP R := makeSUP g
  fBar : SUP sae := map((r1:R):sae +-> convert(r1)@sae,fSUP)$SUP2(R,sae)
  gBar : SUP sae := map((r1:R):sae +-> convert(r1)@sae,gSUP)$SUP2(R,sae)
  ee := extendedEuclidean(fBar,gBar)
  -- not one?(ee.generator) =>
  not (ee.generator = 1) =>
    error "polynomials aren't relatively prime"
  ss1 := ee.coef1; tt1 := ee.coef2
  s1 : SUP R := map((z1:sae):R +-> convert(z1)@R,ss1)$SUP2(sae,R); s := s1
  t1 : SUP R := map((z1:sae):R +-> convert(z1)@R,tt1)$SUP2(sae,R); t := t1
  pPower := p
for i in 2..n repeat
  num := 1 - s * fSUP - t * gSUP
  rhs := (num exquo pPower) :: SUP R
  sigma := map((r1:R):R +-> r1 rem p, s1*rhs);
  tau := map((r1:R):R +-> r1 rem p, t1*rhs)
  s := s + pPower * sigma; t := t + pPower * tau
  quorem := monicDivide(s, gSUP)
  pPower := pPower * p
  s := map((r1:R):R+->r1 rem pPower, quorem.remainder)
  t := map((r1:R):R+->r1 rem pPower, t + fSUP * (quorem.quotient))
[unmakeSUP s, unmakeSUP t]

--mapChineseToList: (L SUP Q, L SUP Q, I) -> L SUP Q
--mapChineseToList(list, polyList, i) ==
mapChineseToList: (L UP, L UP, I, R) -> L UP
mapChineseToList(list, polyList, i, den) ==
  -- 'polyList' consists of MONIC polynomials
  -- computes a polynomial p such that p = pp (modulo polyList[i])
  -- and p = 0 (modulo polyList[j]) for j ≠ i for each 'pp' in 'list'
  -- create polynomials
  q := UP := 1
  for j in 1..(i-1) repeat
    q := q * first polyList
    polyList := rest polyList
  p := first polyList
  polyList := rest polyList
  for j in (i+1).. while not empty? polyList repeat
    q := q * first polyList
    polyList := rest polyList
  --p := map((numer(#1) rem den)/1, p)
  --q := map((numer(#1) rem den)/1, q)
  -- 'den' is a power of an irreducible polynomial
  --!! make this computation more efficient!!
  factoredDen := factor(den)$DDFACT(K,R)
  prime := nthFactor(factoredDen, 1)
  n := nthExponent(factoredDen, 1) :: NNI
  invPoly := approximateExtendedEuclidean(q, p, prime, n).coef1
  -- monicDivide may be inefficient?
  [monicDivide(pp * invPoly * q, p * q).remainder for pp in list]

polyListToMatrix: (L UP, NNI) -> Mat
polyListToMatrix(polyList, n) ==
  mat : Mat := new(n, n, 0)
  for i in 1..n for poly in polyList repeat
    while not zero? poly repeat
      mat(i, degree(poly) + 1) := leadingCoefficient poly
      poly := reductum poly
  mat

chineseRemainder(factors, factorBases, n) ==
denLCM : R := reduce("lcm", [base.basisDen for base in factorBases])
denLCM = 1 => [scalarMatrix(n,1),1,scalarMatrix(n,1)]

-- compute local basis polynomials with denominators cleared
factorBasisPolyLists := basesToPolyLists(factorBases,denLCM)

-- use Chinese remainder to compute basis polynomials w/o denominators
basisPolyLists : L L UP := empty()
for i in 1.. for pList in factorBasisPolyLists repeat
    polyList := mapChineseToList(pList,factors,i,denLCM)
    basisPolyLists := concat(polyList,basisPolyLists)
basisPolys := concat reverse_! basisPolyLists
mat := squareTop rowEchelon(polyListToMatrix(basisPolys,n),denLCM)
matInv := UpTriBddDenomInv(mat,denLCM)
[mat,denLCM,matInv]

— IBACHIN.dotabb —

"IBACHIN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBACHIN"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"IBACHIN" -> "MONOGEN"

package CVMP CoerceVectorMatrixPackage

— CoerceVectorMatrixPackage.input —

)set break resume
)sys rm -f CoerceVectorMatrixPackage.output
)spool CoerceVectorMatrixPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CoerceVectorMatrixPackage
--R
--R CoerceVectorMatrixPackage(R: CommutativeRing) is a package constructor
--R Abbreviation for CoerceVectorMatrixPackage is CVMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CVMP
--R
--R----------------------- Operations -------------------------------
--R coerce : Vector(Matrix(R)) -> Vector(Matrix(Fraction(Polynomial(R)))))
--R coerceP : Vector(Matrix(R)) -> Vector(Matrix(Polynomial(R)))
--R
--E 1

)spool
)lisp (bye)


---

— CoerceVectorMatrixPackage.help —

====================================================================
CoerceVectorMatrixPackage examples
====================================================================

CoerceVectorMatrixPackage is an unexposed, technical package for data conversions

See Also:
o )show CoerceVectorMatrixPackage

---

CoerceVectorMatrixPackage (CVMP)

Exports:
coerce  coerceP

— package CVMP CoerceVectorMatrixPackage —

)abbrev package CVMP CoerceVectorMatrixPackage
CoerceVectorMatrixPackage(R : CommutativeRing): public == private where
M2FP ==> MatrixCategoryFunctions2(R, Vector R, Vector R, Matrix R, _
  Fraction Polynomial R, Vector Fraction Polynomial R, _
  Vector Fraction Polynomial R, Matrix Fraction Polynomial R)
public ==> with
  coerceP: Vector Matrix R -> Vector Matrix Polynomial R
  coerceP(v) coerces a vector v with entries in \spadtype{Matrix R}
  as vector over \spadtype{Matrix Polynomial R}
  coerce: Vector Matrix R -> Vector Matrix Fraction Polynomial R
  coerce(v) coerces a vector v with entries in \spadtype{Matrix R}
  as vector over \spadtype{Matrix Fraction Polynomial R}
private ==> add
  imbedFP : R -> Fraction Polynomial R
  imbedFP r == (r:: Polynomial R) :: Fraction Polynomial R
  imbedP : R -> Polynomial R
  imbedP r == (r:: Polynomial R)

coerceP(g:Vector Matrix R) : Vector Matrix Polynomial R ==
m2 : Matrix Polynomial R
  lim : List Matrix R := entries g
  l: List Matrix Polynomial R := []
  for m in lim repeat
    m2 := map(imbedP,m)$M2P
    l := cons(m2,l)
  vector reverse l

coerce(g:Vector Matrix R) : Vector Matrix Fraction Polynomial R ==
m3 : Matrix Fraction Polynomial R
  lim : List Matrix R := entries g
  l: List Matrix Fraction Polynomial R := []
  for m in lim repeat
    m3 := map(imbedFP,m)$M2FP
    l := cons(m3,l)
  vector reverse l
--- CVMP.dotabb ---
"CVMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CVMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CVMP" -> "PFECAT"

package COMBF CombinatorialFunction

--- CombinatorialFunction.input ---

)set break resume
)sys rm -f CombinatorialFunction.output
)spool CombinatorialFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 7
f := operator 'f
--R
--R
--R (1) f
--R
--E 1

--S 2 of 7
D(product(f(i,x),i=1..m),x)
--R
--R
--R m m f (i,x)
--R ++-++ --+ 2
--R (2) | | f(i,x)> --------
--R | | --+ f(i,x)
--R i= 1 i= 1
--R
--E 2

--S 3 of 7
)set expose add constructor OutputForm
--R
--I OutputForm is already explicitly exposed in frame frame0
--E 3

--S 4 of 7
pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
--R
--R Type: Void
--E 4

--S 5 of 7
displayRow(n)==output center blankSeparate pascalRow(n)
--R
--R Type: Void
--E 5

--S 6 of 7
for i in 0..7 repeat displayRow i
--R
--R Compiling function pascalRow with type NonNegativeInteger -> List(OutputForm)
--R Compiling function displayRow with type NonNegativeInteger -> Void
--R 1
--R 1 1
--R 1 2 1
--R 1 3 3 1
--R 1 4 6 4 1
--R 1 5 10 10 5 1
--R 1 6 15 20 15 6 1
--R 1 7 21 35 35 21 7 1
--R Type: Void
--E 6

--S 7 of 7
--R CombinatorialFunction(R: Join(OrderedSet,IntegralDomain),F: FunctionSpace(R)) is a package constructor
--R Abbreviation for CombinatorialFunction is COMBF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMBF
--R
--R-------------------------------- Operations -----------------------------
--R ?**? : (F,F) -> F
--R ?belong? : BasicOperator -> Boolean
--R binomial : (F,F) -> F
--R factorial : F -> F
--R factorials : F -> F
--R factorials : (F,Symbol) -> F
--R iidbinom : List(F) -> F
--R iidprod : List(F) -> F
--R iifact : F -> F
--R ipow : List(F) -> F
--R permutation : (F,F) -> F
--R product : (F,Symbol) -> F
--R product : (F,SegmentBinding(F)) -> F
--R summation : (F,Symbol) -> F
--R operator : BasicOperator -> BasicOperator
--R summation : (F,SegmentBinding(F)) -> F
--R
--E 7
--- CombinatorialFunction.help ---

CombinatorialFunction examples

Provides combinatorial functions over an integral domain.

\[ f := \text{operator 'f} \]

\[ D(\text{product}(f(i,x), i=1..m), x) \]

\[ \begin{align*}
   i & \quad f(i,x) \\
   i=1 & \quad f(i,x)
\end{align*} \]

The binomial(n, r) returns the number of subsets of \( r \) objects taken among \( n \) objects, i.e. \( \frac{n!}{r! \cdot (n-r)!} \)

The binomial coefficients are the coefficients of the series expansion of a power of a binomial, that is

\[ \begin{align*}
   n \quad \rightarrow \quad \frac{\prod k}{k=0} x
\end{align*} \]

This leads to the famous pascal triangle. First we expose the OutputForm domain, which is normally hidden, so we can use it to format the lines.

\[ \text{set expose add constructor OutputForm} \]

Next we define a function that will output the list of binomial coefficients right justified with proper spacing:
pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]

and now we format the whole line so that it looks centered:

displayRow(n)==output center blankSeparate pascalRow(n)

and we compute the triangle

    for i in 0..7 repeat displayRow i

giving the pretty result:

    Compiling function pascalRow with type NonNegativeInteger -> List
    OutputForm
    Compiling function displayRow with type NonNegativeInteger -> Void

    1
    1 1
    1 2 1
    1 3 3 1
    1 4 6 4 1
    1 5 10 10 5 1
    1 6 15 20 15 6 1
    1 7 21 35 35 21 7 1

See Also:
o )show CombinatorialFunction
o )d op binomial
o )show OutputForm
o )help set

CombinatorialFunction (COMBF)
**Exports:**
- belong?
- binomial
- factorial
- factorials
- iibinom
- iidprod
- iidsum
- iifact
- iiperm
- iipow
- ipow
- permutation
- product
- summation
- operator
- \( ?^{**} \)

**binomial**

We currently simplify binomial coefficients only for non-negative integral second argument, using the formula

\[
\binom{n}{k} = \frac{1}{k!} \prod_{i=0}^{k-1} (n - i),
\]

except if the second argument is symbolic: in this case binomial(n,n) is simplified to one.

Note that there are at least two different ways to define binomial coefficients for negative integral second argument. One way, particular suitable for combinatorics, is to set the binomial coefficient equal to zero for negative second argument. This is, partially, also the approach taken in combinat.spad, where we find

\[
\text{binomial}(n, m) ==
\begin{align*}
& n < 0 \text{ or } m < 0 \text{ or } m > n \Rightarrow 0 \\
& m = 0 \Rightarrow 1
\end{align*}
\]

Of course, here \( n \) and \( m \) are integers. This definition agrees with the recurrence

\[
\binom{n}{k} + \binom{n}{k+1} = \binom{n+1}{k+1}.
\]

Alternatively, one can use the formula

\[
\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)},
\]

and leave the case where \( k \in \mathbb{Z}, n \in \mathbb{Z} \) and \( k \leq n < 0 \) undefined, since the limit does not exist in this case:

Since we then have that \( n - k + 1 \geq 1 \), \( \Gamma(n - k + 1) \) is finite. So it is sufficient to consider \( \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(k+1)} \). On the one hand, we have

\[
\lim_{n_0 \to n} \lim_{k_0 \to k} \frac{\Gamma(n_0 + 1)}{\Gamma(k_0 + 1)} = 0,
\]

since for any non-integral \( n_0 \), \( \Gamma(n_0 + 1) \) is finite. On the other hand,

\[
\lim_{k_0 \to k} \lim_{n_0 \to n} \frac{\Gamma(n_0 + 1)}{\Gamma(k_0 + 1)}
\]
does not exist, since for non-integral $k_0$, $\Gamma(k_0 + 1)$ is finite while $\Gamma(n_0 + 1)$ is unbounded.

However, since for $k \in \mathbb{Z}$, $n \in \mathbb{Z}$ and $0 < k < n$ both definitions agree, one could also combine them. This is what, for example, Mathematica does. It seems that MuPAD sets $\text{binomial}(n,n)=1$ for all arguments $n$, and returns $\text{binomial}(-2, n)$ unevaluated. Provisos may help here.

**dvsum and dvdsum**

The dvsum and dvdsum operations implement differentiation of sums with and without bounds. Note that the function

$$n \mapsto \sum_{k=1}^{n} f(k, n)$$

is well defined only for integral values of $n$ greater than or equal to zero. There is not even consensus how to define this function for $n < 0$. Thus, it is not differentiable. Therefore, we need to check whether we erroneously are differentiating with respect to the upper bound or the lower bound, where the same reasoning holds.

Differentiating a sum with respect to its indexing variable correctly gives zero. This is due to the introduction of dummy variables in the internal representation of a sum: the operator $\%\text{defsum}$ takes 5 arguments, namely

1. the summands, where each occurrence of the indexing variable is replaced by
2. the dummy variable,
3. the indexing variable,
4. the lower bound, and
5. the upper bound.

**dvprod and dvdprod**

The dvprod and dvdprod operations implement differentiation of products with and without bounds. Note again, that we cannot even properly define products with bounds that are not integral.

To differentiate the product, we use Leibniz rule:

$$\frac{d}{dx} \prod_{i=a}^{b} f(i, x) = \sum_{i=a}^{b} \frac{d}{dx} f(i, x) \prod_{i=a}^{b} f(i, x)$$

There is one situation where this definition might produce wrong results, namely when the product is zero, but axiom failed to recognize it: in this case,

$$\frac{d}{dx} f(i, x)/f(i, x)$$
is undefined for some $i$. However, I was not able to come up with an example. The alternative
definition

\[
\frac{d}{dx} \prod_{i=a}^{b} f(i, x) = \sum_{i=a}^{b} \left( \frac{d}{dx} f(i, x) \right) \prod_{j=a, j \neq i}^{b} f(j, x)
\]

has the slight (display) problem that we would have to come up with a new index variable, which looks very ugly. Furthermore, it seems to me that more simplifications will occur with the first definition.

```lisp
f := operator 'f
D(product(f(i,x),i=1..m),x)
```

dvpow2

The dvpow2 operation implements the differentiation of the power operator $%power$ with respect to its second argument, i.e., the exponent. It uses the formula

\[
\frac{d}{dx} g(y)^x = \frac{d}{dx} e^{x \log g(y)} = \log g(y)g(y)^x.
\]

If $g(y)$ equals zero, this formula is not valid, since the logarithm is not defined there. Although strictly speaking $0^x$ is not differentiable at zero, we return zero for convenience.

— package COMBF CombinatorialFunction —

)abbrev package COMBF CombinatorialFunction
++ Author: Manuel Bronstein, Martin Rubey
++ Date Created: 2 Aug 1988
++ Date Last Updated: 30 October 2005
++ Description:
++ Provides combinatorial functions over an integral domain.

CombinatorialFunction(R, F): Exports == Implementation where
R: Join(OrderedSet, IntegralDomain)
F: FunctionSpace R
OP ==> BasicOperator
K ==> Kernel F
SE ==> Symbol
O ==> OutputForm
SMP ==> SparseMultivariatePolynomial(R, K)
Z ==> Integer
POWER ==> "%power"::Symbol
OPEXP ==> "exp"::Symbol
SPECIALDIFF ==> "%specialDiff"
SPECIALDISP ==> "%specialDisp"
SPECIALEQUAL ==> "%specialEqual"

Exports ==> with

belong? : OP -> Boolean
++ belong?(op) is true if op is a combinatorial operator;
operator : OP -> OP
++ operator(op) returns a copy of op with the domain-dependent
++ properties appropriate for F;
++ error if op is not a combinatorial operator;
"**" : (F, F) -> F
++ a ** b is the formal exponential a**b;
binomial : (F, F) -> F
++ binomial(n, r) returns the number of subsets of r objects
++ taken among n objects, i.e. n!/(r! * (n-r)!);
++
++X [binomial(5,i) for i in 0..5]
permutation: (F, F) -> F
++ permutation(n, r) returns the number of permutations of
++ n objects taken r at a time, i.e. n!/(n-r)!;
factorial : F -> F
++ factorial(n) returns the factorial of n, i.e. n!;
factorials : F -> F
++ factorials(f) rewrites the permutations and binomials in f
++ in terms of factorials;
factorials : (F, SE) -> F
++ factorials(f, x) rewrites the permutations and binomials in f
++ involving x in terms of factorials;
summation : (F, SE) -> F
++ summation(f(n), n) returns the formal sum S(n) which verifies
++ S(n+1) - S(n) = f(n);
summation : (F, SegmentBinding F) -> F
++ summation(f(n), n = a..b) returns f(a) + ... + f(b) as a
++ formal sum;
product : (F, SE) -> F
++ product(f(n), n) returns the formal product P(n) which verifies
++ P(n+1)/P(n) = f(n);
product : (F, SegmentBinding F) -> F
++ product(f(n), n = a..b) returns f(a) * ... * f(b) as a
++ formal product;
iifact : F -> F
++ iifact(x) should be local but conditional;
iibinom : List F -> F
++ iibinom(l) should be local but conditional;
iiperm : List F -> F
++ iiperm(l) should be local but conditional;
iipow : List F -> F
++ iipow(l) should be local but conditional;
iidsum : List F -> F
++ iidsum(l) should be local but conditional;
iidprod : List F -> F
++ iidprod(l) should be local but conditional;

ipow  : List F -> F
++ ipow(l) should be local but conditional;

Implementation ==> add
ifact  : F -> F
iiipow : List F -> F
iperm  : List F -> F
ibinom : List F -> F
isum   : List F -> F
ismum  : List F -> F
iprod  : List F -> F
ispsum : List F -> F
dsum   : List F -> 0
ddsum  : List F -> 0
dprod  : List F -> 0
ddprod : List F -> 0
equalsumprod : (K, K) -> Boolean
equaldsumprod : (K, K) -> Boolean
fourth  : List F -> F
dvpow1  : List F -> F
dvpow2  : List F -> F
summand : List F -> F
dvsum   : (List F, SE) -> F
dvdsum  : (List F, SE) -> F
dvprod  : (List F, SE) -> F
dvdprod : (List F, SE) -> F
facts   : (F, List SE) -> F
K2fact  : (K, List SE) -> F
smpfact : (SMP, List SE) -> F

-- This macro will be used in product and summation, both the 5 and 3
-- argument forms. It is used to introduce a dummy variable in place of the
-- summation index within the summands. This in turn is necessary to keep the
-- indexing variable local, circumventing problems, for example, with
-- differentiation.

dummy == new()$SE :: F

opfact := operator("factorial":::Symbol)$CommonOperators
opperm := operator("permutation":::Symbol)$CommonOperators
opbinom := operator("binomial":::Symbol)$CommonOperators
opssum := operator("summation":::Symbol)$CommonOperators
opdsum := operator("%defsum":::Symbol)$CommonOperators
oppprod := operator("product":::Symbol)$CommonOperators
opdprod := operator("%defprod":::Symbol)$CommonOperators
oppow := operator(POWER::Symbol)$CommonOperators

factorial x       == opfact x
binomial(x, y)    == opbinom [x, y]
permutation(x, y) == opperm [x, y]

import F
import Kernel F

number?(x:F):Boolean ==
if R has RetractableTo(Z) then
  ground?(x) or
  ((retractIfCan(x)@Union(Fraction(Z),"failed")) case Fraction(Z))
else
ground?(x)

x ** y ==
-- Do some basic simplifications
is?(x,POWER) =>
  args : List F := argument first kernels x
  not(#args = 2) => error "Too many arguments to **"
  number?(first args) and number?(y) =>
    oppow [first(args)**y, second args]
    oppow [first args, (second args)* y]

-- Generic case
exp : Union(Record(val:F,exponent:Z),"failed") := isPower x
exp case Record(val:F,exponent:Z) =>
  expr := exp::Record(val:F,exponent:Z)
  oppow [expr.val, (expr.exponent)*y]
    oppow [x, y]

belong? op == has?(op, "comb")

fourth l == third rest l
dvpow1 l == second(l) * first(l) ** (second l - 1)
factorials x == facts(x, variables x)
factorials(x, v) == facts(x, [v])
facts(x, l) == smpfact(numer x, l) / smpfact(denom x, l)
summand l == eval(first l, retract(second l)@K, third l)

product(x:F, i:SE) ==
  dm := dummy
  opprod [eval(x, k := kernel(i)$K, dm), dm, k::F]

summation(x:F, i:SE) ==
  dm := dummy
  opsum [eval(x, k := kernel(i)$K, dm), dm, k::F]

-- These two operations return the product or the sum as unevaluated operators
-- A dummy variable is introduced to make the indexing variable local.
dvsum(l, x) ==
  opsum [differentiate(first l, x), second l, third l]
dvdsum(l, x) ==
x = retract(y := third l)@SE => 0
if member?(x, variables(h := third rest rest l)) or
   member?(x, variables(g := third rest l)) then
   error "a sum cannot be differentiated with respect to a bound"
else
  opdsum [differentiate(first l, x), second l, y, g, h]

dvprod(l, x) ==
dm := retract(dummy)@SE
f := eval(first l, retract(second l)@K, dm::F)
p := product(f, dm)

opsym [differentiate(first l, x)/first l * p, second l, third l]

dvdprod(l, x) ==
x = retract(y := third l)@SE => 0
if member?(x, variables(h := third rest rest l)) or
   member?(x, variables(g := third rest l)) then
   error "a product cannot be differentiated with respect to a bound"
else
  opdsum cons(differentiate(first l, x)/first l, rest l) * opdprod l

-- These four operations handle the conversion of sums and products to
-- OutputForm

dprod l ==
  prod(summand(l)::O, third(l)::O)

ddprod l ==
  prod(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)

dsum l ==
  sum(summand(l)::O, third(l)::O)

ddsum l ==
  sum(summand(l)::O, third(l)::O = fourth(l)::O, fourth(rest l)::O)

-- The two operations handle the testing for equality of sums and products.
-- The corresponding property \verb|%specialEqual| set below is checked in
-- Kernel. Note that we can assume that the operators are equal, since this is
-- checked in Kernel itself.

equalsumprod(s1, s2) ==
  l1 := argument s1
  l2 := argument s2
  (eval(first l1, retract(second l1)@K, second l2) = first l2)

equaldsumprod(s1, s2) ==
  l1 := argument s1
l2 := argument s2
((third rest l1 = third rest l2) and
 (third rest rest l1 = third rest rest l2) and
 (eval(first l1, retract(second l1)@K, second l2) = first l2))

-- These two operations return the product or the sum as unevaluated operators
-- A dummy variable is introduced to make the indexing variable local.

product(x:F, s:SegmentBinding F) ==
k := kernel(variable s)$K
dm := dummy
opdprod [eval(x,k,dm), dm, k::F, lo segment s, hi segment s]

summation(x:F, s:SegmentBinding F) ==
k := kernel(variable s)$K
dm := dummy
opdsum [eval(x,k,dm), dm, k::F, lo segment s, hi segment s]

smpfact(p, l) ==
map(x +-> K2fact(x, l), y+->y::F, p)_
$PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, F)

K2fact(k, l) ==
empty? [y for v in variables(kf := k::F) | member?(v, l)] => kf
empty?(args:List F := [facts(a, l) for a in argument k]) => kf
is?(k, opperm) =>
factorial(n := first args) / factorial(n - second args)
is?(k, opbinom) =>
n := first args
p := second args
factorial(n) / (factorial(p) * factorial(n-p))
(operator k) args

operator op ==
is?(op, "factorial":Symbol) => opfact
is?(op, "permutation":Symbol) => opperm
is?(op, "binomial":Symbol) => opbinom
is?(op, "summation":Symbol) => opsum
is?(op, "%defsum":Symbol) => opdsum
is?(op, "product":Symbol) => opprod
is?(op, "%defprod":Symbol) => opdprod
is?(op, POWER) => oppow
error "Not a combinatorial operator"

iprod l ==
zero? first l => 0
 one? first l => 1
(first l = 1) => 1
kernel(oppred, 1)
isum l ==
zero? first l => 0
kernel(opsum, l)

idprod l ==
member?(retract(second l)@SE, variables first l) =>
kernel(opdprod, l)
first(l) ** (fourth rest l - fourth l + 1)

idsum l ==
member?(retract(second l)@SE, variables first l) =>
kernel(opdsum, l)
first(l) * (fourth rest l - fourth l + 1)

ifact x ==
-- zero? x or one? x => 1
zero? x or (x = 1) => 1
kernel(opfact, x)

ibinom l ==
n := first l
((p := second l) = 0) or (p = n) => 1
-- one? p or (p = n - 1) => n
(p = 1) or (p = n - 1) => n
kernel(opbinom, l)

iperm l ==
zero? second l => 1
kernel(opperm, l)

if R has RetractableTo Z then
iidsum l ==
(r1:=retractIfCan(fourth l)@Union(Z,"failed"))
case "failed" or
(r2:=retractIfCan(fourth rest l)@Union(Z,"failed"))
case "failed" or
(k:=retractIfCan(second l)@Union(K,"failed")) case "failed"
=> idsum l
+/[eval(first l,k::K,i::F) for i in r1::Z .. r2::Z]

iidprod l ==
(r1:=retractIfCan(fourth l)@Union(Z,"failed"))
case "failed" or
(r2:=retractIfCan(fourth rest l)@Union(Z,"failed"))
case "failed" or
(k:=retractIfCan(second l)@Union(K,"failed")) case "failed"
=> idprod l
*/[eval(first l,k::K,i::F) for i in r1::Z .. r2::Z]

iiipow l ==
(u := isExpt(x := first l, OPEXP)) case "failed" => kernel(oppow, l)
rec := u::Record(var: K, exponent: Z)
y := first argument(rec.var)
(r := retractIfCan(y)@Union(Fraction Z, "failed")) case "failed" => kernel(oppow, l)
(operator(rec.var)) (rec.exponent * y * second l)

if F has RadicalCategory then
  ipow 1 ==
  (r := retractIfCan(second l)@Union(Fraction Z, "failed"))
  case "failed" => iiipow l
  first(l) ** (r::Fraction(Z))
else
  ipow 1 ==
  (r := retractIfCan(second l)@Union(Z, "failed"))
  case "failed" => iiipow l
  first(l) ** (r::Z)
else
  ipow 1 ==
  zero?(x := first l) =>
  zero? second l => error "0 ** 0"
  0
  -- one? x or zero?(n := second l) => 1
  (x = 1) or zero?(n: F := second l) => 1
  -- one? n => x
  (n = 1) => x
  (u := isExpt(x, OPEXP)) case "failed" => kernel(oppow, l)
  rec := u::Record(var: K, exponent: Z)
  -- one?(y := first argument(rec.var)) or y = -1 =>
  ((y := first argument(rec.var))=1) or y = -1 =>
  (operator(rec.var)) (rec.exponent * y * n)
  kernel(oppow, l)

if R has CombinatorialFunctionCategory then
  iifact x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => ifact x
  factorial(r::R)::F
iiiperm l ==
  (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
  (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
  => iperm l
  permutation(r1::R, r2::R)::F

if R has RetractableTo(Z) and F has Algebra(Fraction(Z)) then
  iiibinom l ==
  (s:=retractIfCan(second l)@Union(R,"failed")) case R and
  (t:=retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
  ans:=1::F
for i in 0..t-1 repeat
   ans:=ans*(first l - i::R::F)
   (1/factorial t) * ans
(s:=retractIfCan(first l-second l)@Union(R,"failed")) case R and
(t:=retractIfCan(s)@Union(Z,"failed")) case Z and t>0 =>
   ans:=1::F
   for i in 1..t repeat
      ans:=ans*(second l+i::R::F)
   (1/factorial t) * ans
(r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
(r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
=> ibinom l
binomial(r1::R, r2::R)::F

-- ibinom checks those cases in which the binomial coefficient may
-- be evaluated explicitly. Currently, the naive iterative algorithm is
-- used to calculate the coefficient, there is room for improvement here.

else
   ibinom l ==
      (r1 := retractIfCan(first l)@Union(R,"failed")) case "failed" or
      (r2 := retractIfCan(second l)@Union(R,"failed")) case "failed"
      => ibinom l
      binomial(r1::R, r2::R)::F
else
   iifact x == ifact x
   iibinom l == ibinom l
   iiperm l == iperm l

if R has ElementaryFunctionCategory then
   iipow l ==
      (r1:=retractIfCan(first l)@Union(R,"failed")) case "failed" or
      (r2:=retractIfCan(second l)@Union(R,"failed")) case "failed"
      => ipow l
      (r1::R ** r2::R)::F
else
   iiow l == iow l

if F has ElementaryFunctionCategory then
   dvpow2 l == if zero?(first l) then
      0
   else
      log(first l) + first(l) ** second(l)

evaluate(opfact, iifact)$BasicOperatorFunctions1(F)
evaluate(oppow, iipow)
evaluate(opperm, iiperm)
evaluate(opbinom, iibinom)
evaluate(opsum, isum)
evaluate(opdsum, iidsum)
evaluate(opprod, iprod)
evaluate(opdpdprod, iidpdpdprod)
derivative(oppow, [dvpow1, dvpow2])

-- These four properties define special differentiation rules for sums and
-- products.
setProperty(opsum, SPECIALDIFF, dvsum@((List F, SE) -> F) pretend None)
setProperty(opdsum, SPECIALDIFF, dvdsum@((List F, SE)->F) pretend None)
setProperty(opprod, SPECIALDIFF, dvprod@((List F, SE)->F) pretend None)
setProperty(opdpdprod, SPECIALDIFF, dvdpdpdprod@((List F, SE)->F) pretend None)

-- Set the properties for displaying sums and products and testing for
-- equality.
setProperty(opsum, SPECIALDISP, dsum@(List F -> O) pretend None)
setProperty(opdsum, SPECIALDISP, ddsum@(List F -> O) pretend None)
setProperty(opprod, SPECIALDISP, dprod@(List F -> O) pretend None)
setProperty(opdpdprod, SPECIALDISP, ddprod@(List F -> O) pretend None)
setProperty(opsum, SPECIALEQUAL, equalsumprod@((K,K) -> Boolean) pretend None)
setProperty(opdsum, SPECIALEQUAL, equaldsumprod@((K,K) -> Boolean) pretend None)
setProperty(opprod, SPECIALEQUAL, equalsumprod@((K,K) -> Boolean) pretend None)
setProperty(opdpdprod, SPECIALEQUAL, equaldsumprod@((K,K) -> Boolean) pretend None)

— COMBF.dotabb —

"COMBF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMBF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMBF" -> "FS"
CHAPTER C

CommonDenominator examples

CommonDenominator provides functions to compute the common denominator of a finite linear aggregate of elements of the quotient field of an integral domain.

See Also:

- show CommonDenominator
CommonDenominator (CDEN)

Exports:
clearDenominator commonDenominator splitDenominator

— package CDEN CommonDenominator —

)abbrev package CDEN CommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Nov 1989
++ Description:
++ CommonDenominator provides functions to compute the
++ common denominator of a finite linear aggregate of elements of
++ the quotient field of an integral domain.

CommonDenominator(R, Q, A): Exports == Implementation where
  R: IntegralDomain
  Q: QuotientFieldCategory R
  A: FiniteLinearAggregate Q

Exports ==> with
  commonDenominator: A -> R
  ++ commonDenominator([q1,...,qn]) returns a common denominator
  ++ d for q1,...,qn.
  clearDenominator : A -> A
  ++ clearDenominator([q1,...,qn]) returns \spad{[p1,...,pn]} such that
  ++ \spad{qi = pi/d} where d is a common denominator for the qi’s.
  splitDenominator : A -> Record(num: A, den: R)
  ++ splitDenominator([q1,...,qn]) returns
  ++ \spad{[[p1,...,pn], d]} such that
  ++ \spad{qi = pi/d} and d is a common denominator for the qi’s.

Implementation ==> add
  clearDenominator l ==
    d := commonDenominator l
    map(x+-+>numer(d*x)::Q, l)
splitDenominator l ==
d := commonDenominator l
[map(x+->numer(d*x)::Q, l), d]

if R has GcdDomain then
qlcm: (Q, Q) -> Q

qlcm(a, b) == lcm(numer a, numer b)::Q
commonDenominator l == numer reduce(qlcm, map(x+->denom(x)::Q, l), 1)
else
commonDenominator l == numer reduce("*", map(x+->denom(x)::Q, l), 1)

package COMMONOP CommonOperators

--- CommonOperators.input ---

)set break resume
)sys rm -f CommonOperators.output
)spool CommonOperators.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CommonOperators
--R
--R CommonOperators is a package constructor
--R Abbreviation for CommonOperators is COMMONOP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMMONOP
--R
--R----------------------------------- Operations -----------------------------------
--R operator : Symbol -> BasicOperator
This package exports the elementary operators, with some semantics already attached to them. The semantics that is attached here is not dependent on the set in which the operators will be applied.

See Also:
o )show CommonOperators

Exports:
operator

---
CommonOperators(): Exports == Implementation where

\[
\begin{align*}
\text{OP} & \rightarrow \text{BasicOperator} \\
0 & \rightarrow \text{OutputForm} \\
\text{POWER} & \rightarrow ",\text{power}"::\text{Symbol} \\
\text{ALGOP} & \rightarrow ",\text{alg}" \\
\text{EVEN} & \rightarrow \"even\" \\
\text{ODD} & \rightarrow \"odd\" \\
\text{DUMMYVAR} & \rightarrow \"\text{dummyVar}\"
\end{align*}
\]

Exports \Rightarrow with

\[
\text{operator: Symbol} \rightarrow \text{OP}
\]

\[
\begin{align*}
\text{add} & \\
\text{dpi} & : \text{List 0} \rightarrow 0 \\
\text{dgamma} & : \text{List 0} \rightarrow 0 \\
\text{dquote} & : \text{List 0} \rightarrow 0 \\
\text{dexp} & : 0 \rightarrow 0 \\
\text{dfact} & : 0 \rightarrow 0 \\
\text{startUp} & : \text{Boolean} \rightarrow \text{Void} \\
\text{setDummyVar: (OP, NonNegativeInteger) \rightarrow OP} \\
\text{brandNew?: Reference(Boolean)} & := \text{ref true}
\end{align*}
\]

\[
\begin{align*}
\text{opalg} & := \text{operator("rootOf"::Symbol, 2)}\$\text{OP} \\
\text{oproot} & := \text{operator("nthRoot"::Symbol, 2)} \\
\text{oppi} & := \text{operator("pi"::Symbol, 0)} \\
\text{oplog} & := \text{operator("log"::Symbol, 1)} \\
\text{opexp} & := \text{operator("exp"::Symbol, 1)} \\
\text{opabs} & := \text{operator("abs"::Symbol, 1)} \\
\text{opsin} & := \text{operator("sin"::Symbol, 1)} \\
\text{opcos} & := \text{operator("cos"::Symbol, 1)} \\
\text{optan} & := \text{operator("tan"::Symbol, 1)} \\
\text{opcot} & := \text{operator("cot"::Symbol, 1)} \\
\text{opsec} & := \text{operator("sec"::Symbol, 1)} \\
\text{opcsc} & := \text{operator("csc"::Symbol, 1)} \\
\text{opasin} & := \text{operator("asin"::Symbol, 1)} \\
\text{opacos} & := \text{operator("acos"::Symbol, 1)} \\
\text{opatan} & := \text{operator("atan"::Symbol, 1)} \\
\text{opacot} & := \text{operator("acot"::Symbol, 1)} \\
\text{opasec} & := \text{operator("asec"::Symbol, 1)}
\end{align*}
\]
PACKAGE COMMONOP COMMONOPERATORS

opacsc := operator("acsc": Symbol, 1)
opsinh := operator("sinh": Symbol, 1)
opcosh := operator("cosh": Symbol, 1)
optanh := operator("tanh": Symbol, 1)
opcoth := operator("coth": Symbol, 1)
opsech := operator("sech": Symbol, 1)
opcsch := operator("csch": Symbol, 1)
opasinh := operator("asinh": Symbol, 1)
opacosh := operator("acosh": Symbol, 1)
opatanh := operator("atanh": Symbol, 1)
opacoth := operator("acoth": Symbol, 1)
opasech := operator("asech": Symbol, 1)
opacsch := operator("acsch": Symbol, 1)
opbox := operator("%box": Symbol)$OP
oppren := operator("%paren": Symbol)$OP
opquote := operator("applyQuote": Symbol)$OP
opdiff := operator("%diff": Symbol, 3)
opsi := operator("Si": Symbol, 1)
opci := operator("Ci": Symbol, 1)
opei := operator("Ei": Symbol, 1)
opli := operator("li": Symbol, 1)
operf := operator("erf": Symbol, 1)
opl12 := operator("dilog": Symbol, 1)
opfis := operator("fresnelS": Symbol, 1)
opfic := operator("fresnel1C": Symbol, 1)
opGamma := operator("Gamma": Symbol, 1)
opGamma2 := operator("Gamma2": Symbol, 2)
opBeta := operator("Beta": Symbol, 2)
opdigamma := operator("digamma": Symbol, 1)
oppolygamma := operator("polygamma": Symbol, 2)
opBesselJ := operator("besselJ": Symbol, 2)
opBesselY := operator("besselY": Symbol, 2)
opBesselI := operator("besselI": Symbol, 2)
opBesselK := operator("besselK": Symbol, 2)
opAiryAi := operator("airyAi": Symbol, 1)
opAiryBi := operator("airyBi": Symbol, 1)
opint := operator("integral": Symbol, 3)
opdint := operator("%defint": Symbol, 5)
opfact := operator("factorial": Symbol, 1)
opperm := operator("permutation": Symbol, 2)
opbinom := operator("binomial": Symbol, 2)
opow := operator("POWER", 2)
opsum := operator("summation": Symbol, 3)
opdsum := operator("%defsum": Symbol, 5)
opprom := operator("product": Symbol, 3)
opdprod := operator("%defprod": Symbol, 5)

algop := [oproot, opalg]$List(OP)
rtrigop := [opsin, opcos, optan, opcot, opsec, opcsc,
opasin, opacos, optan, opcot, opasec, opacsc]
htrigop := [opsinh, opcosh, optanh, opcoth, opsech, opcsch, opasinh, opacosh, opatanh, opacoth, opasech, opacsch]
trigop := concat(rtrigop, htrigop)
elemop := concat(trigop, [oppi, oplog, opexp])
primop := [opei, opli, opsi, opci, operf, opli2, opint, opdint, opfis, opfic]
combop := [opfact, opperm, opbinom, oppow, opsum, opdsum, opprod, opdprod]
specop := [opGamma, opGamma2, opBeta, opdigamma, oppolygamma, opabs, opBesselJ, opBesselY, opBesselI, opBesselK, opAiryAi, opAiryBi]
anyop := [oppren, opdiff, opbox, opquote]
allop := concat(concat(concat(concat(allop, elemop), primop), combop), specop), anyop)

-- odd and even operators, must be maintained current!
evenop := [opcos, opcsc, opcosh, opsech, opabs]
oddop := [opsin, opcsc, optan, opcot, opasin, opacsc, opatan, opsinh, opcsch, optanh, opcoth, opasinh, opacsch, opatanh, opacoth, opsi, operf]

-- operators whose second argument is a dummy variable
dummyvarop1 := [opdiff, opalg, opint, opsum, opprod]

-- operators whose second and third arguments are dummy variables
dummyvarop2 := [opdint, opdsum, opdprod]

operator s ==
  if (deref brandNew?) then startUp false
  for op in allop repeat
    is?(op, s) => return copy op
  operator(s)$OP

dpi l == "%pi"::Symbol::0
dfact x == postfix("!":Symbol::0, (ATOM(x)$Lisp => x; paren x))
dquote l == prefix(quote(first(l)::O), rest l)
dgamma l == prefix(hconcat("|":Symbol::O, overbar(" ":Symbol::0)), l)
setDummyVar(op, n) == setProperty(op, DUMMYVAR, n pretend None)
dexp x ==
e := "%e":Symbol::0
x = 1::0 => e
  e ** x

fsupersub(x:List 0):0 == supersub("A":Symbol::0, x)
fbinomial(x:List 0):0 == binomial(first x, second x)
fpower(x:List 0):0 == first(x) ** second(x)
fsum(x:List 0):0 == sum(first x, second x, third x)
fprod(x:List 0):0 == prod(first x, second x, third x)
finite(x:List 0):0 ==
int(first x * hconcat("d":Symbol::0, second x),empty(), third x)

fpren(x:List InputForm):InputForm ==
  convert concat(convert("("::Symbol)@InputForm,
  concat(x, convert(")"::Symbol)@InputForm))

fpow(x:List InputForm):InputForm ==
  convert concat(convert("**"::Symbol)@InputForm, x)

froot(x:List InputForm):InputForm ==
  convert [convert("**"::Symbol)@InputForm, first x, 1 / second x]

startUp b ==
  brandNew?() := b
  display(oppren, paren)
  display(opbox, commaSeparate)
  display(oppi, dpi)
  display(opexp, dexp)
  display(opGamma, dgamma)
  display(opGamma2, dgamma)
  display(opfact, dfact)
  display(opquote, dquote)
  display(opperm, fsupersub)
  display(opbinom, fbinomial)
  display(oppov, fpower)
  display(opsum, fsum)
  display(opprod, fprod)
  display(opint, fint)
  input(oppren, fpren)
  input(oppow, fpow)
  input(oproot, froot)
  for op in algop repeat assert(op, ALGOP)
  for op in rtrigop repeat assert(op, "rtrig")
  for op in htrigop repeat assert(op, "htrig")
  for op in trigop repeat assert(op, "trig")
  for op in elemop repeat assert(op, "elem")
  for op in primop repeat assert(op, "prim")
  for op in combop repeat assert(op, "comb")
  for op in specop repeat assert(op, "special")
  for op in anyop repeat assert(op, "any")
  for op in evenop repeat assert(op, "EVEN")
  for op in oddop repeat assert(op, "ODD")
  for op in dummyvarop1 repeat setDummyVar(op, 1)
  for op in dummyvarop2 repeat setDummyVar(op, 2)
  assert(oppren, "linear")
  void
"COMMONOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMMONOP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"COMMONOP" -> "ALIST"

package COMMUPC CommuteUnivariatePolynomialCategory

— CommuteUnivariatePolynomialCategory.input —

)set break resume
)sys rm -f CommuteUnivariatePolynomialCategory.output
)spool CommuteUnivariatePolynomialCategory.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CommuteUnivariatePolynomialCategory
--R
--R CommuteUnivariatePolynomialCategory(R: Ring,UP: UnivariatePolynomialCategory(R),UPUP: UnivariatePolynomialCategory(UP)) is a package constructor
--R Abbreviation for CommuteUnivariatePolynomialCategory is COMMUPC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMMUPC
--R
--R--------------------------------- Operations ---------------------------------
--R swap : UPUP -> UPUP
--R
--E 1

)spool
)lisp (bye)

— CommuteUnivariatePolynomialCategory.help —

====================================================================
CommuteUnivariatePolynomialCategory examples
====================================================================
A package for swapping the order of two variables in a tower of two \texttt{UnivariatePolynomialCategory} extensions.

See Also:
\begin{itemize}
\item \texttt{)}show \texttt{CommuteUnivariatePolynomialCategory}
\end{itemize}

---

\textbf{CommuteUnivariatePolynomialCategory (COMMUPC)}

\begin{verbatim}
COMMUPC
PFECAT
\end{verbatim}

Exports:
\begin{itemize}
\item \texttt{swap}
\end{itemize}

---

\texttt{)}abbrev package COMMUPC CommuteUnivariatePolynomialCategory
++ Author: Manuel Bronstein
++ Description:
++ A package for swapping the order of two variables in a tower of two
++ \texttt{UnivariatePolynomialCategory} extensions.

\texttt{CommuteUnivariatePolynomialCategory(R, UP, UPUP): Exports == Impl where}
\begin{itemize}
\item \texttt{R} : \texttt{Ring}
\item \texttt{UP} : \texttt{UnivariatePolynomialCategory R}
\item \texttt{UPUP} : \texttt{UnivariatePolynomialCategory UP}
\end{itemize}

\texttt{N ==> NonNegativeInteger}

Exports ==> with
\begin{itemize}
\item \texttt{swap: UPUP -> UPUP}
\end{itemize}
++ \texttt{swap(p(x,y))} returns \texttt{p(y,x)}. 

++
Impl ==> add

\[
\text{makePoly: (UP, N) \to UPUP}
\]

-- converts \(P(x,y)\) to \(P(y,x)\)

\[
\text{swap poly ==}
\]

\[
\text{ans:UPUP := 0}
\]

\[
\text{while poly \neq 0 repeat}
\]

\[
\text{ans := ans + makePoly(leadingCoefficient poly, degree poly)}
\]

\[
\text{poly := reductum poly}
\]

\[
\text{ans}
\]

\[
\text{makePoly(poly, d) ==}
\]

\[
\text{ans:UPUP := 0}
\]

\[
\text{while poly \neq 0 repeat}
\]

\[
\text{ans := ans + monomial(monomial(leadingCoefficient poly, d), degree poly)}
\]

\[
\text{poly := reductum poly}
\]

\[
\text{ans}
\]

package COMPFACT ComplexFactorization

— ComplexFactorization.input —

)set break resume
)sys rm -f ComplexFactorization.output
)spool ComplexFactorization.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexFactorization
--R
--R ComplexFactorization(RR: EuclideanDomain,PR: UnivariatePolynomialCategory(Complex(RR))) is
--R Abbreviation for ComplexFactorization is COMPFACT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMPFACT
--R
--R----------------------------------- Operations -------------------------------
--R factor : PR -> Factored(PR)
--R

)spool
)lisp (bye)

__________________________________________________________

— ComplexFactorization.help —

====================================================================
ComplexFactorization examples
====================================================================

This package has no description

See Also:
o )show ComplexFactorization

ComplexFactorization (COMPFACT)

Exports:
factor

— package COMPFACT ComplexFactorization —
)abbrev package COMPFAC ComplexFactorization
++ Description:
++ This package has no description

ComplexFactorization(RR,PR) : C == T where
  RR : EuclideanDomain -- R is Z or Q
  PR : UnivariatePolynomialCategory Complex RR
  R ==> Complex RR
  I ==> Integer
  RN ==> Fraction I
  GI ==> Complex I
  GRN ==> Complex RN

C == with

  factor : PR -> Factored PR
  ++ factor(p) factorizes the polynomial p with complex coefficients.

T == add
  SUP ==> SparseUnivariatePolynomial
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FF ==> Record(flg:fUnion, fctr:PR, xpnt:Integer)
  SAEF := SimpleAlgebraicExtensionAlgFactor(SUP RN,GRN,SUP GRN)
  UPCF2 := UnivariatePolynomialCategoryFunctions2(R,PR,GRN,SUP GRN)
  UPCFB := UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,R,PR)

myMap(r:R) : GRN ==
  R is GI =>
    cr :GI := r pretend GI
    complex((real cr)::RN,(imag cr)::RN)
  R is GRN => r pretend GRN

compND(cc:GRN):Record(cnum:GI,cden:Integer) ==
  ccr:=real cc
  cci:=imag cc
  dccr:=denom ccr
  dccci:=denom cci
  ccd:=lcm(dccr,dcci)
  [complex(((ccd exquo dccr)::Integer)*numer ccr,
    ((ccd exquo dccci)::Integer)*numer cci),ccd]

conv(f:SUP GRN) :Record(convP:SUP GI, convD:RN) ==
  pris:SUP GI :=0
  dris:Integer:=1
  dris1:Integer:=1
  pdris:Integer:=1
  for i in 0..(degree f) repeat
    (cf:= coefficient(f,i)) = 0 => "next i"
    cdf:=compND cf
dris := \text{lcm}(\text{cdf.cden}, \text{dris1})
pris := ((\text{dris} \text{exquo} \text{dris1})::\text{Integer}) \times \text{pris} + 
((\text{dris} \text{exquo} \text{cdf.cden})::\text{Integer})*
\text{monomial}(\text{cdf.cnum}, i)$(\text{SUP GI})
\text{dris1} := \text{dris}
[\text{pris,dris}::\text{RN}]

\text{backConv}(f\text{fr}:\text{Factored SUP GRN}) : \text{Factored PR} ==
R \text{ is GRN} =>
\text{makeFR}((\text{unit ffr}) \text{pretend PR},[[f.\text{flg},(f.\text{fctr}) \text{pretend PR},f.\text{xpnt}]
\text{for f in factorList ffr})
R \text{ is GI} =>
\text{const} := \text{unit ffr}
\text{ris} : \text{List FF} ::=[
\text{for ff in factorList ffr repeat}
\text{fact} := \text{primitivePart}(\text{conv}(f\text{fr}.\text{fctr}).\text{convP})
\text{expf} := ff.\text{xpnt}
\text{ris} := \text{cons}([[ff.\text{flg}, \text{fact pretend PR}, \text{expf}], \text{ris}])
\text{lc} : \text{GRN} := \text{myMap leadingCoefficient}(\text{fact pretend PR})
\text{const} := \text{const} * (\text{leadingCoefficient}(ff.\text{fctr})/\text{lc})**\text{expf}
\text{uconst} : \text{GI} := \text{compND(coefficient(const,0)).cnum}
\text{makeFR}((\text{uconst pretend R})::\text{PR}, \text{ris})

\text{factor}(\text{pol} : \text{PR}) : \text{Factored PR} ==
\text{ratPol} : \text{SUP GRN} := 0
\text{ratPol} := \text{map}(\text{myMap}, \text{pol}$\text{UPCF2}$
\text{ffr} := \text{factor} \text{ratPol}
\text{backConv ffr}

——

— COMPFACT.dotabb —

"COMPFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMPFACT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"COMPFACT" \rightarrow "COMPCAT"

——

package COMPLEX2 ComplexFunctions2

— ComplexFunctions2.input —

)set break resume
CHAPTER 4. CHAPTER C

---S 1 of 1
)show ComplexFunctions2
--R
--R ComplexFunctions2(R: CommutativeRing,S: CommutativeRing) is a package constructor
--R Abbreviation for ComplexFunctions2 is COMPLEX2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMPLEX2
--R
--R------------------------------ Operations -------------------------------
--R map : ((R -> S),Complex(R)) -> Complex(S)
--R
--E 1

)spool
)lisp (bye)

---

--- ComplexFunctions2.help ---

====================================================================
ComplexFunctions2 examples
====================================================================

This package extends maps from underlying rings to maps between
complex over those rings.

See Also:
  o )show ComplexFunctions2

---
ComplexFunctions2 (COMPLEX2)

Exports:
map

package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation
--- ComplexIntegerSolveLinearPolynomialEquation.input ---

)set break resume
)sys rm -f ComplexIntegerSolveLinearPolynomialEquation.output
)spool ComplexIntegerSolveLinearPolynomialEquation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexIntegerSolveLinearPolynomialEquation
--R ComplexIntegerSolveLinearPolynomialEquation(R: IntegerNumberSystem,Cr: ComplexCategory(R))
--R Abbreviation for ComplexIntegerSolveLinearPolynomialEquation is CINTSLPE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CINTSLPE
--R
--R----------------------------------- Operations -----------------------------------
--R solveLinearPolynomialEquation : (List(SparseUnivariatePolynomial(CR)),SparseUnivariatePolynomial(CR)) -> Union(List(SparseUnivariatePolynomial(CR)),"failed")
--R
--E 1

)spool
)lisp (bye)

---

--- ComplexIntegerSolveLinearPolynomialEquation.help ---

====================================================================
ComplexIntegerSolveLinearPolynomialEquation examples
====================================================================

This package provides the generalized euclidean algorithm which is needed as the basic step for factoring polynomials.

See Also:
  o )show ComplexIntegerSolveLinearPolynomialEquation

---
ComplexIntegerSolveLinearPolynomialEquation (CINTSLPE)

Exports:
solveLinearPolynomialEquation

--- package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation ---

)abbrev package CINTSLPE ComplexIntegerSolveLinearPolynomialEquation++ Author: James Davenport++ Date Created: 1990++ Description:+ This package provides the generalized euclidean algorithm which is++ needed as the basic step for factoring polynomials.

ComplexIntegerSolveLinearPolynomialEquation(R,CR): C == T
where
CP ==> SparseUnivariatePolynomial CR
R:IntegerNumberSystem
CR:ComplexCategory(R)
C == with
solveLinearPolynomialEquation: (List CP,CP) -> Union(List CP,"failed")++ solveLinearPolynomialEquation([f1, ..., fn], g)++ where (fi relatively prime to each other)++ returns a list of ai such that++ g = sum ai prod fj (j \neq i) or++ equivalently g/prod fj = sum (ai/fi)++ or returns "failed" if no such list exists

T == add
oldlp:List CP := []
slpePrime:R:=(2::R)
oldtable:Vector List CP := empty()
solveLinearPolynomialEquation(lp,p) ==
if (oldlp ^= lp) then
   -- we have to generate a new table
   deg:= \+/[degree u for u in lp]
   ans:=Union(Vector List CP,"failed"):="failed"
slpePrime:=67108859::R -- 2**26 -5 : a prime
-- a good test case for this package is
-- (good question?)
while (ans case "failed") repeat
  ans:=tablePow(deg,complex(slpePrime,0),lp)$GenExEuclid(CR,CP)
  if (ans case "failed") then
    slpePrime:= slpePrime-4::R
  while not prime?(slpePrime)$IntegerPrimesPackage(R) repeat
    slpePrime:= slpePrime-4::R
  oldtable:=(ans:: Vector List CP)
  answer:=solveid(p,complex(slpePrime,0),oldtable)
  answer

---------

--- CINTSLPE.dotabb ---

"CINTSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CINTSLPE"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CINTSLPE" -> "COMPCAT"

---------

package COMPLPAT ComplexPattern

--- ComplexPattern.input ---

)set break resume
)sys rm -f ComplexPattern.output
)spool ComplexPattern.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexPattern
--R
--R ComplexPattern(R: SetCategory,S: Join(ConvertibleTo(Pattern(R)),CommutativeRing),CS: ComplexCategory(R)) is a package constructor
--R Abbreviation for ComplexPattern is COMPLPAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMPLPAT
--R
--R----------------------------- Operations -----------------------------
--R convert : CS -> Pattern(R)
ComplexPattern (COMPLPAT)

Exports:
convert

---

ComplexPattern examples

This package supports converting complex expressions to patterns

See Also:
  o )show ComplexPattern

---

EXTERNALS

ComplexPattern examples

This package supports converting complex expressions to patterns

See Also:
  o )show ComplexPattern

---

Exports:
convert
++ This package supports converting complex expressions to patterns

ComplexPattern(R, S, CS) : C == T where
  R: SetCategory
  S: Join(ConvertibleTo Pattern R, CommutativeRing)
  CS: ComplexCategory S
  C == with
    convert: CS -> Pattern R
    ++ convert(cs) converts the complex expression cs to a pattern

T == add

  ipat : Pattern R := patternVariable("%i":Symbol, true, false, false)

  convert(cs) ==
    zero? imag cs => convert real cs
    convert real cs + ipat * convert imag cs

— COMPLPAT.dotabb —

"COMPLPAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMPLPAT"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"COMPLPAT" -> "COMPCAT"

package CPMATCH ComplexPatternMatch

— ComplexPatternMatch.input —

)set break resume
)sys rm -f ComplexPatternMatch.output
)spool ComplexPatternMatch.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexPatternMatch
--R
--R ComplexPatternMatch(R: SetCategory,S: Join(PatternMatchable(R),CommutativeRing),CS: ComplexCategory) is a package constructor
--R Abbreviation for ComplexPatternMatch is CPMATCH
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CPMATCH
--R
--R-------------------------------- Operations ---------------------------------
--R patternMatch : (CS,Pattern(R),PatternMatchResult(R,CS)) -> PatternMatchResult(R,CS) if Polynomial(S) has
--R
--E 1

)spool
)lisp (bye)

ComplexPatternMatch.help —

==================================================================== ComplexPatternMatch examples
====================================================================

This package supports matching patterns involving complex expressions

See Also:
o )show ComplexPatternMatch

ComplexPatternMatch (CPMATCH)

Exports:
patternMatch

— package CPMATCH ComplexPatternMatch —
)abbrev package CPMATCH ComplexPatternMatch
++ Author: Barry Trager
++ Date Created: 30 Nov 1995
++ Description:
++ This package supports matching patterns involving complex expressions

ComplexPatternMatch(R, S, CS) : C == T where
R: SetCategory
S: Join(PatternMatchable R, CommutativeRing)
CS: ComplexCategory S
PMRS ==> PatternMatchResult(R, CS)
PS ==> Polynomial S
C == with
  if PS has PatternMatchable(R) then
    patternMatch: (CS, Pattern R, PMRS) -> PMRS
      ++ patternMatch(cexpr, pat, res) matches the pattern pat to the
      ++ complex expression cexpr. res contains the variables of pat
      ++ which are already matched and their matches.

T == add
import PatternMatchPushDown(R, S, CS)
import PatternMatchResultFunctions2(R, PS, CS)
import PatternMatchResultFunctions2(R, CS, PS)

ivar : PS := "%i"::Symbol::PS

makeComplex(p:PS):CS ==
  up := univariate p
  degree up > 1 => error "not linear in %i"
  icoef:=leadingCoefficient(up)
  rcoef:=leadingCoefficient(reductum p)
  complex(rcoef,icoef)

makePoly(cs:CS):PS == real(cs)*ivar + imag(cs)::PS

if PS has PatternMatchable(R) then
  patternMatch(cs, pat, result) ==
    zero? imag cs =>
      patternMatch(real cs, pat, result)
    map(makeComplex, patternMatch(makePoly cs, pat, map(makePoly, result)))
package CRFP ComplexRootFindingPackage

-- ComplexRootFindingPackage.input --

)set break resume
)sys rm -f ComplexRootFindingPackage.output
)spool ComplexRootFindingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexRootFindingPackage

-- R
-- R ComplexRootFindingPackage(R: Join(Field,OrderedRing),UP: UnivariatePolynomialCategory(Complex(R))) is a
-- R Abbreviation for ComplexRootFindingPackage is CRFP
-- R This constructor is not exposed in this frame.
-- R Issue )edit bookvol10.4.pamphlet to see algebra source code for CRFP
-- R
-- R----------------------------------- Operations -----------------------------------
-- R factor : (UP,R) -> Factored(UP) factor : UP -> Factored(UP)
-- R graeffe : UP -> UP norm : UP -> R
-- R reciprocalPolynomial : UP -> UP rootRadius : (UP,R) -> R
-- R rootRadius : UP -> R schwerpunkt : UP -> Complex(R)
-- R setErrorBound : R -> R
-- R complexZeros : UP -> List(Complex(R))
-- R complexZeros : (UP,R) -> List(Complex(R))
-- R divisorCascade : (UP,UP,Boolean) -> List(Record(factors: List(UP),error: R))
-- R divisorCascade : (UP,UP) -> List(Record(factors: List(UP),error: R))
-- R factor : (UP,R,Boolean) -> Factored(UP)
-- R pleskenSplit : (UP,R,Boolean) -> Factored(UP)
-- R pleskenSplit : (UP,R) -> Factored(UP)
-- R startPolynomial : UP -> Record(start: UP,factors: Factored(UP))

-- R
-- E 1

)spool
)lisp (bye)
ComplexRootFindingPackage provides functions to find all roots of a polynomial p over the complex number by using Plesken’s idea to calculate in the polynomial ring modulo f and employing the Chinese Remainder Theorem.

In this first version, the precision (see digits) is not increased when this is necessary to avoid rounding errors. Hence it is the user’s responsibility to increase the precision if necessary.

Note also, if this package is called with e.g. Fraction Integer, the precise calculations could require a lot of time.

Also note that evaluating the zeros is not necessarily a good check whether the result is correct: already evaluation can cause rounding errors.

See Also:
- )show ComplexRootFindingPackage

Exports:
complexZeros  divisorCascade  factor  graeffe  norm
pleskenSplit  reciprocalPolynomial  rootRadius  schwerpunkt  setErrorBound
startPolynomial

--- package CRFP ComplexRootFindingPackage ---

)abbrev package CRFP ComplexRootFindingPackage
++ Author: J. Grabmeier
++ Date Created: 31 January 1991
++ Date Last Updated: 12 April 1991
++ References: J. Grabmeier: On Plesken's root finding algorithm,
++ in preparation
++ A. Schoenhage: The fundamental theorem of algebra in terms of computational
++ complexity, preliminary report, Univ. Tuebingen, 1982
++ Description:
++ \spadtype{ComplexRootFindingPackage} provides functions to
++ find all roots of a polynomial \spad{p} over the complex number by
++ using Plesken's idea to calculate in the polynomial ring
++ modulo \spad{f} and employing the Chinese Remainder Theorem.
++ In this first version, the precision (see digits)
++ is not increased when this is necessary to
++ avoid rounding errors. Hence it is the user's responsibility to
++ increase the precision if necessary.
++ Note also, if this package is called with e.g. \spad{Fraction Integer},
++ the precise calculations could require a lot of time.
++ Also note that evaluating the zeros is not necessarily a good check
++ whether the result is correct: already evaluation can cause
++ rounding errors.

ComplexRootFindingPackage\( (\mathcal{R}, \mathcal{P}) \) public == private where
-- \mathcal{R} : Join(Field, OrderedRing, CharacteristicZero)
-- Float not in CharacteristicZero !
\mathcal{R} : Join(Field, OrderedRing)
\mathcal{P} : UnivariatePolynomialCategory Complex \mathcal{R}

\mathcal{C} \implies Complex \mathcal{R}
\mathcal{FR} \implies Factored
\mathcal{I} \implies Integer
\mathcal{L} \implies List
\mathcal{FAE} \implies Record(factors : \mathcal{L} \mathcal{P}, error : \mathcal{R})
\mathcal{NNI} \implies NonNegativeInteger
\mathcal{OF} \implies OutputForm
\mathcal{ICF} \implies IntegerCombinatoricFunctions(I)

public == with
complexZeros : \mathcal{P} \to \mathcal{L} \mathcal{C}
++ complexZeros(\mathcal{p}) tries to determine all complex zeros
++ of the polynomial \mathcal{p} with accuracy given by the package
++ constant \texttt{globalEps} which you may change by \texttt{setErrorBound}.
complexZeros : (\mathcal{P}, \mathcal{R}) \to \mathcal{L} \mathcal{C}
complexZeros(p, eps) tries to determine all complex zeros of the polynomial p with accuracy given by eps.

divisorCascade : (UP,UP, Boolean) -> L FAE
++ divisorCascade(p,tp) assumes that degree of polynomial tp is smaller than degree of polynomial p, both monic.
++ A sequence of divisions are calculated using the remainder, made monic, as divisor
++ for the the next division. The result contains also the error of the ++ factorizations, i.e. the norm of the remainder polynomial.
++ If info is true, then information messages are issued.

divisorCascade : (UP,UP) -> L FAE
++ divisorCascade(p,tp) assumes that degree of polynomial tp is smaller than degree of polynomial p, both monic.
++ A sequence of divisions is calculated using the remainder, made monic, as divisor
++ for the the next division. The result contains also the error of the ++ factorizations, i.e. the norm of the remainder polynomial.

defactor: (UP,R,Boolean) -> FR UP
++ defactor(p, eps, info) tries to factor p into linear factors with error atmost eps. An overall error bound
++ eps0 is determined and iterated tree-like calls to pleskenSplit are used to get the factorization.
++ If info is true, then information messages are given.

defactor: (UP,R) -> FR UP
++ defactor(p, eps) tries to factor p into linear factors with error atmost eps. An overall error bound
++ eps0 is determined and iterated tree-like calls to pleskenSplit are used to get the factorization.

defactor: UP -> FR UP
++ defactor(p) tries to factor p into linear factors with error atmost globalEps, the internal error bound,
++ which can be set by setErrorBound. An overall error bound
++ eps0 is determined and iterated tree-like calls to pleskenSplit are used to get the factorization.

graeffe : UP -> UP
++ graeffe p determines q such that \spad{q(-z**2) = p(z)*p(-z)}.
++ Note that the roots of q are the squares of the roots of p.

norm : UP -> R
++ norm(p) determines sum of absolute values of coefficients
++ Note that this function depends on abs.

pleskenSplit: (UP, R, Boolean) -> FR UP
++ pleskenSplit(poly,eps,info) determines a start polynomial start by using "startPolynomial then it increases the exponent
++ n of start ** n mod poly to get an approximate factor of poly, in general of degree "degree poly -1". Then a divisor
++ cascade is calculated and the best splitting is chosen, as soon ++ as the error is small enough.
--++ In a later version we plan
--++ to use the whole information to get a split into more than 2 --++ factors.
++ If info is true, then information messages are issued.

pleskenSplit: (UP, R) -> FR UP
++ pleskenSplit(poly, eps) determines a start polynomial start
++ by using "startPolynomial then it increases the exponent
++ n of start ** n mod poly to get an approximate factor of
++ poly, in general of degree "degree poly -1". Then a divisor
++ cascade is calculated and the best splitting is chosen, as soon
++ as the error is small enough.
--++ In a later version we plan
--++ to use the whole information to get a split into more than 2
--++ factors.

reciprocalPolynomial: UP -> UP
++ reciprocalPolynomial(p) calculates a polynomial which has exactly
++ the inverses of the non-zero roots of p as roots, and the same
++ number of 0-roots.

rootRadius: (UP,R) -> R
++ rootRadius(p, errQuot) calculates the root radius of p with a
++ maximal error quotient of errQuot.

rootRadius: UP -> R
++ rootRadius(p) calculates the root radius of p with a
++ maximal error quotient of 1+globalEps, where
++ globalEps is the internal error bound, which can be
++ set by setErrorBound.

schwerpunkt: UP -> C
++ schwerpunkt(p) determines the 'Schwerpunkt' of the roots of the
++ polynomial p of degree n, i.e. the center of gravity, which is
++ coefficient of \spad{x**(n-1)} divided by
++ n times coefficient of \spad{x**n}.

setErrorBound : R -> R
++ setErrorBound(eps) changes the internal error bound,
-- by default being 10 ** (-20) to eps, if R is
++ by default being 10 ** (-3) to eps, if R is
++ a member in the category \spadtype{QuotientFieldCategory Integer}.
++ The internal globalDigits is set to
++ \em ceiling(1/r)**2*10 being 10**7 by default.

startPolynomial: UP -> Record(start: UP, factors: FR UP)
++ startPolynomial(p) uses the ideas of Schoenhage’s
++ variant of Graeffe’s method to construct circles which separate
++ roots to get a good start polynomial, i.e. one whose
++ image under the Chinese Remainder Isomorphism has both entries
++ of norm smaller and greater or equal to 1. In case the
++ roots are found during internal calculations.
++ The corresponding factors
++ are in factors which are otherwise 1.

private ==> add

Rep := ModMonic(C, UP)
-- constants
c : C
r : R
--globalDigits : I := 10 ** 41
globalDigits : I := 10 ** 7
globalEps : R :=
  --a : R := (100000000000000000000000 :: I) :: R
  a : R := (1000 :: I) :: R
  1/a
emptyLine : OF := " "
dashes : OF := center "-----------------------------------------------"
dots : OF := center "............................................."
one : R := 1$R
two : R := 2 * one
ten : R := 10 * one
eleven : R := 11 * one
weakEps := eleven/ten
--invLog2 : R := 1/log10 (2*one)

-- signatures of local functions

absC : C -> R

absR : R -> R

--
calculateScale : UP -> R
--
makeMonic : UP -> UP
  -- 'makeMonic p' divides 'p' by the leading coefficient,
  -- to guarantee new leading coefficient to be 1$R we cannot
  -- simply divide the leading monomial by the leading coefficient
  -- because of possible rounding errors
min: (FAE, FAE) -> FAE
-- takes factorization with smaller error
nthRoot : (R, NNI) -> R
  -- nthRoot(r,n) determines an approximation to the n-th
  -- root of r, if \spadtype{R} has ?**?: (R,Fraction Integer)->R
  -- we use this, otherwise we use approxNthRoot via
  -- \spadtype{Integer}
shift: (UP,C) -> UP
  -- shift(p,c) changes p(x) into p(x+c), thereby modifying the
  -- roots u_j of p to the roots (u_j - c) of shift(p,c)
scale: (UP,C) -> UP
  -- scale(p,c) changes p(x) into p(cx), thereby modifying the
  -- roots u_j of p to the roots ((1/c) u_j) of scale(p,c)

-- implementation of exported functions
complexZeros(p, eps) ==
  -- r1 : R := rootRadius(p, weakEps)
  -- eps0 : R = r1 * nthRoot(eps, degree p)
  -- right now we are content with
  eps0 : R := eps/(ten ** degree p)
  facs : FR UP := factor(p, eps0)
  [-coefficient(linfac.factor, 0) for linfac in factors facs]

complexZeros p == complexZeros(p, globalEps)
setErrorBound r ==
  r <= 0 => error "setErrorBound: need error bound greater 0"
  globalEps := r
  if R has QuotientFieldCategory Integer then
    rd : Integer := ceiling(1/r)
    globalDigits := rd * rd * 10
  lof : List OF := _
  ["setErrorBound: internal digits set to", globalDigits::OF]
  print hconcat lof
  messagePrint "setErrorBound: internal error bound set to"
  globalEps

pleskenSplit(poly, eps, info) ==
  p := makeMonic poly
  fp : FR UP
  if not zero? (md := minimumDegree p) then
    fp : FR UP := irreducibleFactor(monomial(1,1)$UP, md)$(FR UP)
  p := p quo monomial(1, md)$UP
  sP : Record(start: UP, factors: FR UP) := startPolynomial p
  fp : FR UP := sP.factors
  if not one? fp then
    if not (fp = 1) then
      qr : Record(quotient: UP, remainder: UP) := divide(p, makeMonic expand fp)
      p := qr.quotient
      st := sP.start
      zero? degree st => fp
      -- we calculate in ModMonic(C, UP),
      -- next line defines the polynomial, which is used for reducing
      setPoly p
      nm : R := eps
      split : FAE
      sR : Rep := st :: Rep
      psR : Rep := sR ** (degree poly)
      notFoundSplit : Boolean := true
      while notFoundSplit repeat
        if info then
          lof : L OF := ["not successful, new exponent:", nn::OF]
          print hconcat lof
          psR := psR * psR * sR -- exponent (2*d +1)
          -- be careful, too large exponent results in rounding errors
-- tp is the first approximation of a divisor of poly:
> tp : UP := lift psR
> zero? degree tp =>
> if info then print "we leave as we got constant factor"
> nilFactor(poly,1)$(FR UP)
-- this was the case where we don't find a non-trivial factorization
-- we refine tp by repeated polynomial division and hope that
-- the norm of the remainder gets small from time to time
> splits : L FAE := divisorCascade(p, makeMonic tp, info)
> split := reduce(min,splits)
> notFoundSplit := (eps <= split.error)

for fac in split.factors repeat
  fp :=
  -- one? degree fac => fp * nilFactor(fac,1)$(FR UP)
  (degree fac = 1) => fp * nilFactor(fac,1)$(FR UP)
  fp * irreducibleFactor(fac,1)$(FR UP)
fp

startPolynomial p == -- assume minimumDegree is 0
  --print (p :: OF)
  fp : FR UP := 1
  -- one? degree p =>
  (degree p = 1) =>
  p := makeMonic p
  [p,irreducibleFactor(p,1)]
startPoly : UP := monomial(1,1)$UP
eps : R := weakEps -- 10 per cent errors allowed
r1 : R := rootRadius(p, eps)
rd : R := 1/rootRadius(reciprocalPolynomial p, eps)
(r1 > (2::R)) and (rd < 1/(2::R)) => [startPoly,fp] -- unit circle splitting!
-- otherwise the norms of the roots are too closed so we
-- take the center of gravity as new origin:
> u : C := schwerpunkt p
> startPoly := startPoly-monomial(u,0)
> p := shift(p,-u)
-- determine new rootRadius:
> r1 : R := rootRadius(p, eps)
> startPoly := startPoly/(r1::C)
-- use one of the 4 points ri*zeta, where zeta is a 4th root of unity
-- as new origin, this could be changed to an arbitrary list
-- of elements of norm 1.
> listOfCenters : L C := [complex(r1,0), complex(0,r1),
  complex(-r1,0), complex(0,-r1)]
> lp : L UP := [shift(p,v) for v in listOfCenters]
-- next we check if one of these centers is a root
> centerIsRoot : Boolean := false
for i in 1..maxIndex lp repeat
  if (mD := minimumDegree lp.i) > 0 then
    pp : UP := monomial(1,1)-monomial(listOfCenters.i-u,0)
centerIsRoot := true
fp := fp * irreducibleFactor(pp,mD)
centerIsRoot =>
p := shift(p,u) quo expand fp
-- print (p::OF)
zero? degree p => [p,fp]
sp:= startPolynomial(p)
[sP.start,fp]

-- choose the best one w.r.t. maximal quotient of norm of largest
-- root and norm of smallest root
lpr1 : L R := [rootRadius(q,eps) for q in lp]
lprd : L R := [1/rootRadius(reciprocalPolynomial q,eps) for q in lp]
-- later we should check here of an rd is smaller than globalEps
lq : L R := []
for i in 1..maxIndex lpr1 repeat
  lq := cons(lpr1.i/lprd.i, lq)
--lq := reverse lq
po := position(reduce(max,lq),lq)
--p := lp.po
--lrr : L R := [rootRadius(p,i,1+eps) for i in 2..(degree(p)-1)]
--lrr := concat(concat(lpr1.po,lrr),lprd.po)
--lu : L R := [(lrr.i + lrr.(i+1))/2 for i in 1..(maxIndex(lrr)-1)]
[startPoly - monomial(listOfCenters.po,0),fp]

norm p ==
-- reduce(_+$R,map(absC,coefficients p))
nm : R := 0
for c in coefficients p repeat
  nm := nm + absC c
nm

pleskenSplit(poly,eps) == pleskenSplit(poly,eps,false)

graeffe p ==
-- If p = ao x++n + a1 x**(n-1) + ... + a<n-1> x + an
-- and q = bo x++n + b1 x**(n-1) + ... + b<n-1> x + bn
-- are such that q(-x**2) = p(x)p(-x), then
-- bk := ak**2 + 2 * ((-1) * a<k-1>*a<k+1> + ... +
-- (-1)**l * a<l>*a<l>) where l = min(k, n-k).
-- graeffe(p) constructs q using these identities.
n : NNI := degree p
aForth : L C := []
for k in 0..n repeat -- aForth = [a0, a1, ..., a<n-1>, an]
aForth := cons(coefficient(p, k::NNI), aForth)
aBack : L C := [] -- after k steps
  -- aBack = [ak, a<k-1>, ..., a1, a0]
gp := 0$UP
for k in 0..n repeat
  ak : C := first aForth
aForth := rest aForth
aForthCopy : L C := aForth -- we iterate over aForth and
aBackCopy : L C := aBack -- aBack but do not want to
  -- destroy them
sum    : C := 0
const  : I := -1 -- after i steps const = (-1)**i
for aminus in aBack for aplus in aForth repeat
  -- after i steps aminus = a<k-i> and aplus = a<k+i>
  sum := sum + const * aminus * aplus
aForthCopy := rest aForthCopy
aBackCopy := rest aBackCopy
const := -const
gp := gp + monomial(ak*ak + 2 * sum, (n-k)::NNI)
aBack := cons(ak, aBack)
gp

rootRadius(p, errorQuotient) ==
  errorQuotient <= 1$R =>
    error "rootRadius: second Parameter must be greater than 1"
  pp : UP := p
  rho : R := calculateScale makeMonic pp
  rR : R := rho
  pp := makeMonic scale(pp,complex(rho,0$R))
  expo : NNI := 1
d  : NNI := degree p
  currentError : R := nthRoot(2::R, 2)
currentError := d*20*currentError
while nthRoot(currentError, expo) >= errorQuotient repeat
  -- if info then print (expo :: OF)
  pp := graeffe pp
  rho := calculateScale pp
  expo := 2 * expo
  rR := nthRoot(rho, expo) * rR
  pp := makeMonic scale(pp,complex(rho,0$R))
rR

rootRadius(p) == rootRadius(p, 1+globalEps)

schwerpunkt p ==
  zero? p => 0$C
  zero? (d := degree p) => error 
  "schwerpunkt: non-zero const. polynomial has no roots and no schwerpunkt"
  -- coefficient of x**d and x**(d-1)
  lC : C := coefficient(p,d) -- ^= 0
  nC : C := coefficient(p,(d-1) pretend NNI)
  (denom := recip ((d::I::C)*lC)) case "failed" => error "schwerpunkt: 

  degree * leadingCoefficient not invertible in ring of coefficients"
  -- (nC*(denom::C))
reciprocalPolynomial p ==
  zero? p => 0
  d : NNI := degree p
  md : NNI := d+minimumDegree p
  lm : L UP := [monomial(coefficient(p,i),(md-i) :: NNI) for i in 0..d]
  sol := reduce(_,+, lm)

divisorCascade(p, tp, info) ==
  lfae : L FAE := nil()
  for i in 1..degree tp while (degree tp > 0) repeat
    -- USE monicDivide !!!
    qr : Record(quotient: UP, remainder: UP) := divide(p,tp)
    factor1 : UP := qr.quotient
    factor2 : UP := makeMonic factor1
    -- refinement of tp:
    tp := qr.remainder
    nm : R := norm tp
    listOfFactors : L UP := cons(factor2,nil()$(L UP))
    listOfFactors := cons(factor1,listOfFactors)
    lfae := cons( [listOfFactors,nm], lfae)
  if info then
    --lof : L OF := [i :: OF,"-th division:"::OF]
    --print center box hconcat lof
    print emptyLine
    lof : L OF := ["error polynomial has degree " ::OF,_
                  (degree tp)::OF, " and norm " :: OF, nm :: OF]
    print center hconcat lof
    lof : L OF := ["degrees of factors:" ::OF,_
                  (degree factor1)::OF," ", (degree factor2)::OF]
    print center hconcat lof
    if info then print emptyLine
    reverse lfae
  divisorCascade(p, tp) == divisorCascade(p, tp, false)

factor(poly,eps) == factor(poly,eps,false)
factor(p) == factor(p, globalEps)

factor(poly,eps,info) ==
  result : FR UP := coerce monomial(leadingCoefficient poly,0)
  d : NNI := degree poly
  --should be
  --den : R := (d::I)::R * two**(d::Integer) * norm poly
  --eps0 : R := eps / den
  -- for now only
  eps0 : R := eps / (ten*ten)
  --
  one? d => irreducibleFactor(poly,1)$(FR UP)
  (d = 1) => irreducibleFactor(poly,1)$(FR UP)
  listOfFactors : L Record(factor: UP,exponent: I) :=_
list [makeMonic poly,1]
if info then
  lof : L OF := [dashes,dots,"list of Factors:",dots,listOfFactors::OF, _
    dashes, "list of Linear Factors:", dots, result::OF, _
    dots,dashes]
  print vconcat lof
while not null listOfFactors repeat
  p : UP := (first listOfFactors).factor
  exponentOfp : I := (first listOfFactors).exponent
  listOfFactors := rest listOfFactors
  if info then
    lof : L OF := ["just now we try to split the polynomial:",p::OF]
    print vconcat lof
  split : FR UP := pleskenSplit(p, eps0, info)
  -- one? numberOfFactors split =>
  (numberOfFactors split = 1) =>
    -- in a later version we will change error bound and
    -- accuracy here to deal this case as well
    lof : L OF := ["factor: couldn't split factor",_
      center(p :: OF), "with required error bound"]
    print vconcat lof
    result := result * nilFactor(p, exponentOfp)
    -- now we got 2 good factors of p, we drop p and continue
    -- with the factors, if they are not linear, or put a
    -- linear factor to the result
    for rec in factors(split)$(FR UP) repeat
      newFactor : UP := rec.factor
      expOfFactor := exponentOfp * rec.exponent
      -- one? degree newFactor =>
      (degree newFactor = 1) =>
        result := result * nilFactor(newFactor,expOfFactor)
        listOfFactors:=cons([newFactor,expOfFactor],_
          listOfFactors)
      result

-- implementation of local functions

absC c == nthRoot(norm(c)$C,2)
absR r ==
  r < 0 => -r
  r
min(fae1,fae2) ==
  fae2.error < fae1.error => fae2
  fae1
calculateScale p ==
  d := degree p
  maxi : R := 0
  for j in 1..d for cof in rest coefficients p repeat
    -- here we need abs: R -> R
    rc : R := absR real cof
ic : R := absR imag cof
locmax: R := max(rc,ic)
maxi := max( nthRoot( locmax/(binomial(d,j)$ICF::R), j), maxi)
-- Maybe I should use some type of logarithm for the following:
maxi = 0$R => error("Internal Error: scale cannot be 0")
rho := one
rho < maxi =>
while rho < maxi repeat rho := ten * rho
rho / ten
while maxi < rho repeat rho := rho / ten
rho = 0 => one
rho
makeMonic p ==
p = 0 => p
monomial(1,degree p)$UP + (reductum p)/(leadingCoefficient p)
scale(p, c) ==
-- eval(p,cx) is missing !!
eq : Equation UP := equation(monomial(1,1), monomial(c,1))
eval(p,eq)
-- improvement?: direct calculation of the new coefficients
shift(p,c) ==
rhs : UP := monomial(1,1) + monomial(c,0)
eq : Equation UP := equation(monomial(1,1), rhs)
eval(p,eq)
-- improvement?: direct calculation of the new coefficients
nthRoot(r,n) ==
R has RealNumberSystem => r ** (1/n)
R has QuotientFieldCategory Integer =>
den : I := approxNthRoot(globalDigits * denom r ,n)$IntegerRoots(I)
um : I := approxNthRoot(globalDigits * numer r ,n)$IntegerRoots(I)
num/den
-- the following doesn’t compile
--R has coerce: % -> Fraction Integer =>
-- q : Fraction Integer := coerce(r)@Fraction(Integer)
-- den : I := approxNthRoot(globalDigits * denom q ,n)$IntegerRoots(I)
-- num : I := approxNthRoot(globalDigits * numer q ,n)$IntegerRoots(I)
-- num/den
r -- this is nonsense, perhaps a Newton iteration for x**n-r here
)

-- for late use:
graeffe2 p ==
-- substitute x by -x :
eq : Equation UP := equation(monomial(1,1), monomial(-1$C,1))
pp : UP := p*eval(p,eq)
gp : UP := 0$UP
while \( pp \) ^= 0 repeat
\[
\text{i:NNI := (degree } pp \text{ quo } (2::\text{NNI})
\]
\[
\text{coef:C:=}
\]
\[
even? \text{ i } \Rightarrow \text{ leadingCoefficient } pp
\]
\[
- \text{ leadingCoefficient } pp
\]
\[
\text{gp := gp + monomial(coef,i)}
\]
\[
\text{pp := reductum } pp
\]
\[
\text{gp}
\]

\text{shift2}(p,c) ==
\[
\text{d := degree } p
\]
\[
\text{cc : C := 1}
\]
\[
\text{coef := List C := [cc := c * cc for i in 1..d]}
\]
\[
\text{coef := cons(1,coef)}
\]
\[
\text{coef := [coefficient(p,i)*coef.(i+1) for i in 0..d]}
\]
\[
\text{res : UP := 0}
\]
\[
\text{for } j \text{ in 0..d repeat}
\]
\[
\text{cc := 0}
\]
\[
\text{for } i \text{ in } j..d \text{ repeat}
\]
\[
\text{cc := cc + coef.i * (binomial(i,j)$ICF :: R)}
\]
\[
\text{res := res + monomial(cc,j)$UP}
\]
\[
\text{res}
\]

\text{scale2}(p,c) ==
\[
\text{d := degree } p
\]
\[
\text{cc : C := 1}
\]
\[
\text{coef := List C := [cc := c * cc for i in 1..d]}
\]
\[
\text{coef := cons(1,coef)}
\]
\[
\text{coef := [coefficient(p,i)*coef.(i+1) for i in 0..d]}
\]
\[
\text{res : UP := 0}
\]
\[
\text{for } i \text{ in 0..d repeat res := res + monomial(coef.(i+1),i)$UP}
\]
\[
\text{res}
\]

\text{scale2: (UP,C) -> UP}
\text{shift2: (UP,C) -> UP}
\text{graeffe2 : UP -> UP}
\[
\text{++ graeffe2 } p \text{ determines } q \text{ such that } \spad{q(-z**2) = p(z)*p(-z)}.\n\]
\[
\text{++ Note that the roots of } q \text{ are the squares of the roots of } p.
\]

— CRFP.dotabb —
package CMPLXRT ComplexRootPackage

--- ComplexRootPackage.input ---

)set break resume
)sys rm -f ComplexRootPackage.output
)spool ComplexRootPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ComplexRootPackage

--R
--R ComplexRootPackage(UP: UnivariatePolynomialCategory(Complex(Integer)),Par: Join(Field,OrderedRing)) is
--R Abbreviation for ComplexRootPackage is CMPLXRT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CMPLXRT
--R
--R------------------------------- Operations --------------------------------
--R complexZeros : (UP,Par) -> List(Complex(Par))
--R
--E 1

)spool
)lisp (bye)


--- ComplexRootPackage.help ---

====================================================================
ComplexRootPackage examples
====================================================================

This package provides functions complexZeros for finding the complex zeros of univariate polynomials with complex rational number coefficients. The results are to any user specified precision and are returned as either complex rational number or complex floating point numbers depending on the type of the second argument which specifies the precision.

See Also:
- )show ComplexRootPackage

--------
ComplexRootPackage (CMPLXRT)

Exports:
complexZeros

— package CMPLXRT ComplexRootPackage —

)abbrev package CMPLXRT ComplexRootPackage
++ Author: P. Gianni
++ Description:
++ This package provides functions complexZeros
++ for finding the complex zeros
++ of univariate polynomials with complex rational number coefficients.
++ The results are to any user specified precision and are returned
++ as either complex rational number or complex floating point numbers
++ depending on the type of the second argument which specifies the
++ precision.

ComplexRootPackage(UP,Par) : T == C where
RN ==> Fraction Integer
I ==> Integer
NF ==> Float
UP : UnivariatePolynomialCategory Complex Integer
Par : Join(Field, OrderedRing) -- will be Float or RN
CP ==> Complex Par
PCI ==> Polynomial Complex Integer

T == with
  complexZeros:(UP,Par) -> List CP
  ++ complexZeros(poly, eps) finds the complex zeros of the
  ++ univariate polynomial poly to precision eps with
  ++ solutions returned as complex floats or rationals
  ++ depending on the type of eps.

C == add
complexZeros(p:UP,eps:Par):List CP ==
package CTRIGMNP ComplexTrigonometricManipulations

--- ComplexTrigonometricManipulations.input ---

)set break resume
)sys rm -f ComplexTrigonometricManipulations.output
)spool ComplexTrigonometricManipulations.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show ComplexTrigonometricManipulations
--R
--R ComplexTrigonometricManipulations(R: Join(IntegralDomain,OrderedSet,RetractableTo(Integer)),F: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(Complex(R)))) is a package constructor
--R Abbreviation for ComplexTrigonometricManipulations is CTRIGMNP
ComplexTrigonometricManipulations (CTRIGMNP)

ComplexTrigonometricManipulations provides function that compute the real and imaginary parts of complex functions.

See Also:
- )show ComplexTrigonometricManipulations
Exports:
complexElementary complexForm complexNormalize imag real trigs

— package CTRIGMNP ComplexTrigonometricManipulations —

)abbrev package CTRIGMNP ComplexTrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 11 June 1993
++ Date Last Updated: 14 June 1993
++ Description:
++ \spadtype{ComplexTrigonometricManipulations} provides function that
++ compute the real and imaginary parts of complex functions.

ComplexTrigonometricManipulations(R, F): Exports == Implementation where
R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory, 
    FunctionSpace Complex R)

SY ==> Symbol
FR ==> Expression R
K ==> Kernel F

Exports ==> with
    complexNormalize: F -> F
    ++ complexNormalize(f) rewrites \spad{f} using the least possible number
    ++ of complex independent kernels.
    complexNormalize: (F, SY) -> F
    ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
    ++ number of complex independent kernels involving \spad{x}.
    complexElementary: F -> F
    ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
    ++ complex transcendental elementary functions: \spad{log, exp}.
    complexElementary: (F, SY) -> F
    ++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
    ++ \spad{x} in terms of the 2 fundamental complex
    ++ transcendental elementary functions: \spad{log, exp}.
    real : F -> FR
    ++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
    ++ function.
    imag : F -> FR
    ++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
    ++ is a complex function.
    real? : F -> Boolean
    ++ real?(f) returns \spad{true} if \spad{f = real f}.
    trigs : F -> F
    ++ trigs(f) rewrites all the complex logs and exponentials
    ++ appearing in \spad{f} in terms of trigonometric functions.
complexForm: F \to \text{Complex FR} \\
\quad \text{++ complexForm}(f) \text{ returns } \spad{[\text{real } f, \text{imag } f]}.

Implementation ==> add
\quad \text{import InnerTrigonometricManipulations}(R, FR, F)
\quad \text{import ElementaryFunctionStructurePackage(Complex R, F)}

rreal?: \text{Complex R} \to \text{Boolean}

kreal?: \text{Kernel F} \to \text{Boolean}

localexplogs : (F, F, List SY) \to F

real f == \text{real complexForm } f

imag f == \text{imag complexForm } f

rreal? r == \text{zero? imag } r

kreal? k == \text{every?(rreal?, argument } k)\text{List(}F\text{)}

\text{complexForm } f == \text{explogs2trigs } f

trigs f ==
\quad \text{GF2FG } \text{explogs2trigs } f

real? f ==
\quad \text{every?(rreal?, coefficients numer } f\text{) and every?(kreal?, kernels } f\text{)}

localexplogs(f, g, lx) ==
\quad \text{trigs2explogs}(g, [k \text{ for } k \text{ in tower } f
\quad \quad \text{| is?(k, "tan":SY) or is?(k, "cot":SY)], } \text{lx})

\text{complexElementary } f ==
\quad \text{any?(x } \mapsto \text{has?(x, "rtrig"),}
\quad \quad \text{operators}(g := \text{realElementary } f)\text{List(BasicOperator) } \to
\quad \quad \text{localexplogs}(f, g, \text{variables } g)
\quad g

\text{complexElementary}(f, x) ==
\quad \text{any?(y } \mapsto \text{has?(operator } y, \text{ "rtrig"),}
\quad \quad [k \text{ for } k \text{ in tower}(g := \text{realElementary}(f, x))
\quad \quad \text{| member?(x, variables}(k::F))\text{List(K)})\text{List(K) } \to
\quad \quad \text{localexplogs}(f, g, [x])
\quad g

\text{complexNormalize}(f, x) ==
\quad \text{any?(y } \mapsto \text{has?(operator } y, \text{ "rtrig"),}
\quad \quad [k \text{ for } k \text{ in tower}(g := \text{realElementary}(f, x))
\quad \quad \text{| member?(x, variables}(k::F))\text{List(K)})\text{List(K) } \Rightarrow
\quad \quad \text{rischNormalize}(\text{localexplogs}(f, g, [x]), x).\text{func}
\quad \text{rischNormalize}(g, x).\text{func}

\text{complexNormalize } f ==
\quad l := \text{variables}(g := \text{realElementary } f)
any?(y +-> has?(y, "rtrig"), operators g)$List(BasicOperator) =>
h := localexplogs(f, g, l)
for x in l repeat h := rischNormalize(h, x).func
h
for x in l repeat g := rischNormalize(g, x).func
g

———

CTRIGMNP.dotabb ——

"CTRIGMNP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CTRIGMNP"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"CTRIGMNP" -> "ACF"
"CTRIGMNP" -> "FS"
"CTRIGMNP" -> "COMPCAT"

———

package ODECONST ConstantLODE

—— ConstantLODE.input ——

)set break resume
)sys rm -f ConstantLODE.output
)spool ConstantLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ConstantLODE
--R
--R ConstantLODE(R: Join(OrderedSet,EuclideanDomain,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)) =>
--R Abbreviation for ConstantLODE is ODECONST
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODECONST
--R
--R----------------------------------- Operations -----------------------------------
--R constDsolve : (L,F,Symbol) -> Record(particular: F,basis: List(F))
--R
--E 1
ConstantLODE examples

Solution of linear ordinary differential equations, constant coefficient case.

See Also:
- )show ConstantLODE

Exports:
- constDsolve

— package ODECONST ConstantLODE —

)abbrev package ODECONST ConstantLODE
++ Author: Manuel Bronstein
++ Date Created: 18 March 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ Solution of linear ordinary differential equations,
++ constant coefficient case.

ConstantLODE(R, F, L): Exports == Implementation where
  R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
          LinearlyExplicitRingOver Integer, CharacteristicZero)
  F: Join(AlgebraicallyClosedFunctionSpace R,
          TranscendentalFunctionCategory, PrimitiveFunctionCategory)
  L: LinearOrdinaryDifferentialOperatorCategory F

Z ==> Integer
SY ==> Symbol
K ==> Kernel F
V ==> Vector F
M ==> Matrix F
SUP ==> SparseUnivariatePolynomial F

Exports ==> with
  constDsolve: (L, F, SY) -> Record(particular:F, basis:List F)
  constDsolve(op, g, x) returns \spad{[f, [y1,...,ym]]}
  where \( f \) is a particular solution of the equation \( \spad{op y = g} \),
  and the \spad{yi}'s form a basis for the solutions of \( \spad{op y = 0} \).

Implementation ==> add
  import ODETools(F, L)
  import ODEIntegration(R, F)
  import ElementaryFunctionSign(R, F)
  import AlgebraicManipulations(R, F)
  import FunctionSpaceIntegration(R, F)
  import FunctionSpaceUnivariatePolynomialFactor(R, F, SUP)

homoBasis: (L, F) -> List F
quadSol : (SUP, F) -> List F
basisSqfr: (SUP, F) -> List F
basisSol : (SUP, Z, F) -> List F

constDsolve(op, g, x) ==
  b := homoBasis(op, x::F)
  [particularSolution(op, g, b, (f1:F):F -> int(f1, x))::F, b]

homoBasis(op, x) ==
  p:SUP := 0
  while op ^= 0 repeat
    p := p + monomial(leadingCoefficient op, degree op)
    op := reductum op
  b:List(F) := empty()
  for ff in factors ffactor p repeat
    b := concat_!(b, basisSol(ff.factor, dec(ff.exponent), x))
  b

basisSol(p, n, x) ==
l := basisSqfr(p, x)
zero? n => l
ll := copy l
xn := x::F
for i in 1..n repeat
  l := concat_!(l, [xn * f for f in ll])
  xn := x * xn
l

basisSqfr(p, x) ==
  -- one?(d := degree p) =>
  ((d := degree p) = 1) =>
    [exp(- coefficient(p, 0) * x / leadingCoefficient p)]
  d = 2 => quadSol(p, x)
  [exp(a * x) for a in rootsOf p]

quadSol(p, x) ==
  (u := sign(delta := (b := coefficient(p, 1))**2 - 4 *
             (a := leadingCoefficient p) * (c := coefficient(p, 0))))
  case Z and negative?(u::Z) =>
    y := x / (2 * a)
    r := - b * y
    i := rootSimp(sqrt(-delta)) * y
    [exp(r) * cos(i), exp(r) * sin(i)]
  [exp(a * x) for a in zerosOf p]

— ODECONST.dotabb —
"ODECONST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODECONST"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ODECONST" -> "ACFS"

package COORDSYS CoordinateSystems

— CoordinateSystems.input —

)set break resume
)sys rm -f CoordinateSystems.output
)spool CoordinateSystems.output
)set message test on
PACKAGE COORDSYS COORDINATESYSTEMS

)set message auto off
)clear all

--S 1 of 1
)show CoordinateSystems
--R
--R CoordinateSystems(R: Join(Field,TranscendentalFunctionCategory,RadicalCategory)) is a package constructor
--R Abbreviation for CoordinateSystems is COORDSYS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COORDSYS
--R
--R------------------------------- Operations --------------------------------
--R cartesian : Point(R) -> Point(R) cylindrical : Point(R) -> Point(R)
--R parabolic : Point(R) -> Point(R) paraboloidal : Point(R) -> Point(R)
--R polar : Point(R) -> Point(R) spherical : Point(R) -> Point(R)
--R bipolar : R -> (Point(R) -> Point(R)) bipolarCylindrical : R -> (Point(R) -> Point(R))
--R conical : (R,R) -> (Point(R) -> Point(R))
--R elliptic : R -> (Point(R) -> Point(R)) ellipticCylindrical : R -> (Point(R) -> Point(R))
--R oblateSpheroidal : R -> (Point(R) -> Point(R))
--R parabolicCylindrical : Point(R) -> Point(R) prolateSpheroidal : R -> (Point(R) -> Point(R))
--R toroidal : R -> (Point(R) -> Point(R))
--R
--E 1

)spool
)lisp (bye)

-------------

CoordinateSystems.help

====================================================================
CoordinateSystems examples
====================================================================

CoordinateSystems provides coordinate transformation functions for plotting. Functions in this package return conversion functions which take points expressed in other coordinate systems and return points with the corresponding Cartesian coordinates.

See Also:
o )show CoordinateSystems

-------------
### CoordinateSystems (COORDSYS)

**Exports:**
- bipolar
- bipolarCylindrical
- cartesian
- conical
- cylindrical
- elliptic
- ellipticCylindrical
- oblateSpheroidal
- parabolic
- parabolicCylindrical
- paraboloidal
- polar
- prolateSpheroidal
- spherical
- toroidal

---

```spad
define COORDSYS(PTCAT)
++ Author: Jim Wen
++ Date Created: 12 March 1990
++ Date Last Updated: 19 June 1990, Clifton J. Williamson
++ Description:
++ CoordinateSystems provides coordinate transformation functions
++ for plotting. Functions in this package return conversion functions
++ which take points expressed in other coordinate systems and return points
++ with the corresponding Cartesian coordinates.

coordinateSystems(R): Exports == Implementation where
R : Join(Field,TranscendentalFunctionCategory,RadicalCategory)
Pt == Point R

Exports ==

- cartesian : Pt -> Pt
  + cartesian(pt) returns the Cartesian coordinates of point pt.
- polar : Pt -> Pt
  + polar(pt) transforms pt from polar coordinates to Cartesian
  + coordinates: the function produced will map the point \( r, \theta \) to \( x = r \cos(\theta), y = r \sin(\theta) \).
- cylindrical : Pt -> Pt
  + cylindrical(pt) transforms pt from polar coordinates to Cartesian
  + coordinates: the function produced will map the point \( r, \theta, z \) to \( x = r \cos(\theta), y = r \sin(\theta), z \).
- spherical : Pt -> Pt
  + spherical(pt) transforms pt from polar coordinates to Cartesian
  + coordinates: the function produced will map the point \( r, \theta, \phi \) to \( x = r \sin(\theta) \cos(\phi), y = r \sin(\theta) \sin(\phi), z = r \cos(\theta) \).
```

---
++ spherical(pt) transforms pt from spherical coordinates to Cartesian
++ coordinates: the function produced will map the point \spad{(r,theta,phi)}
++ to \spad{x = r*sin(phi)*cos(theta)}, \spad{y = r*sin(phi)*sin(theta)},
++ \spad{z = r*cos(phi)}.
parabolic: Pt \to Pt
++ parabolic(pt) transforms pt from parabolic coordinates to Cartesian
++ coordinates: the function produced will map the point \spad{(u,v)} to
++ \spad{x = 1/2*(u**2 - v**2)}, \spad{y = u*v}.
parabolicCylindrical: Pt \to Pt
++ parabolicCylindrical(pt) transforms pt from parabolic cylindrical
++ coordinates to Cartesian coordinates: the function produced will
++ map the point \spad{(u,v,z)} to \spad{x = 1/2*(u**2 - v**2)},
++ \spad{y = u*v}, \spad{z}.
paraboloidal: Pt \to Pt
++ paraboloidal(pt) transforms pt from paraboloidal coordinates to
++ Cartesian coordinates: the function produced will map the
++ point \spad{(u,v,phi)} to \spad{x = u*v*cos(phi)}, \spad{y = u*v*sin(phi)},
++ \spad{z = 1/2 * (u**2 - v**2)}.

elliptic: R \to (Pt \to Pt)
++ elliptic(a) transforms from elliptic coordinates to Cartesian
++ coordinates: \spad{elliptic(a)} is a function which will map the
++ point \spad{(u,v)} to \spad{x = a*cosh(u)*cos(v)}, \spad{y = a*sinh(u)*sin(v)}.

ellipticCylindrical: R \to (Pt \to Pt)
++ ellipticCylindrical(a) transforms from elliptic cylindrical coordinates
++ to Cartesian coordinates: \spad{ellipticCylindrical(a)} is a function
++ which will map the point \spad{(u,v,z)} to \spad{x = a*cosh(u)*cos(v)},
++ \spad{y = a*sinh(u)*sin(v)}, \spad{z}.

prolateSpheroidal: R \to (Pt \to Pt)
++ prolateSpheroidal(a) transforms from prolate spheroidal coordinates to
++ Cartesian coordinates: \spad{prolateSpheroidal(a)} is a function
++ which will map the point \spad{(xi,eta,phi)} to
++ \spad{x = a*sinh(xi)*sin(eta)*cos(phi)}, \spad{y = a*sinh(xi)*sin(eta)*sin(phi)},
++ \spad{z = a*cosh(xi)*cos(eta)}.

oblateSpheroidal: R \to (Pt \to Pt)
++ oblateSpheroidal(a) transforms from oblate spheroidal coordinates to
++ Cartesian coordinates: \spad{oblateSpheroidal(a)} is a function which
++ will map the point \spad{(xi,eta,phi)} to \spad{x = a*sinh(xi)*sin(eta)*cos(phi)},
++ \spad{y = a*sinh(xi)*sin(eta)*sin(phi)}, \spad{z = a*cosh(xi)*cos(eta)}.

bipolar: R \to (Pt \to Pt)
++ bipolar(a) transforms from bipolar coordinates to Cartesian coordinates:
++ \spad{bipolar(a)} is a function which will map the point \spad{(u,v)} to
++ \spad{x = a*sinh(v)/(cosh(v)-cos(u))}, \spad{y = a*sin(u)/(cosh(v)-cos(u))}.

bipolarCylindrical: R \to (Pt \to Pt)
++ bipolarCylindrical(a) transforms from bipolar cylindrical coordinates
++ to Cartesian coordinates: \spad{bipolarCylindrical(a)} is a function which
++ will map the point \spad{(u,v,z)} to \spad{x = a*sinh(v)/(cosh(v)-cos(u))},
++ \spad{y = a*sin(u)/(cosh(v)-cos(u))}, \spad{z}.

toroidal: R \to (Pt \to Pt)
++ toroidal(a) transforms from toroidal coordinates to Cartesian
++ coordinates: \spad{toroidal(a)} is a function which will map the point
++ \spad{(u,v,phi)} to \spad{x = a*sinh(v)*cos(phi)/(cosh(v)-cos(u))},
++ \spad{y = a*sinh(v)*sin(phi)/(cosh(v)-cos(u))}, \spad{z = a*sin(u)/(cosh(v)-cos(u))}.

\spad{conical: (R,R) \to (Pt \to Pt)}
++ \spad{conical(a,b)} transforms from conical coordinates to Cartesian coordinates:
++ \spad{conical(a,b)} is a function which will map the point \spad{(lambda,mu,nu)} to
++ \spad{x = lambda*mu*nu/(a*b)},
++ \spad{y = lambda/a*sqrt((mu**2-a**2)*(nu**2-a**2)/(a**2-b**2))},
++ \spad{z = lambda/b*sqrt((mu**2-b**2)*(nu**2-b**2)/(b**2-a**2))}.

Implementation ==> add
cartesian pt ==
-- we just want to interpret the cartesian coordinates
-- from the first \(\mathbb{N}\) elements of the point - so the
-- identity function will do
pt

polar pt0 ==
pt := copy pt0
r := elt(pt0,1); theta := elt(pt0,2)
pt.1 := r * cos(theta); pt.2 := r * sin(theta)
pt

cylindrical pt0 == polar pt0
-- apply polar transformation to first 2 coordinates

spherical pt0 ==
pt := copy pt0
r := elt(pt0,1); theta := elt(pt0,2); phi := elt(pt0,3)
pt.1 := r * sin(phi) * cos(theta); pt.2 := r * sin(phi) * sin(theta)
pt.3 := r * cos(phi)
pt

parabolic pt0 ==
pt := copy pt0
u := elt(pt0,1); v := elt(pt0,2)
pt.1 := (u*u - v*v)/(2::R); pt.2 := u*v
pt

parabolicCylindrical pt0 == parabolic pt0
-- apply parabolic transformation to first 2 coordinates

paraboloidal pt0 ==
pt := copy pt0
u := elt(pt0,1); v := elt(pt0,2); phi := elt(pt0,3)
pt.1 := u*v*cos(phi); pt.2 := u*v*sin(phi); pt.3 := (u*u - v*v)/(2::R)
pt
elliptic a ==
\( x \rightarrow \)
\[ \text{pt} := \text{copy}(x) \]
\[ u := \text{elt}(x,1); v := \text{elt}(x,2) \]
\[ \text{pt.1} := a \cdot \cosh(u) \cdot \cos(v); \text{pt.2} := a \cdot \sinh(u) \cdot \sin(v) \]
\[ \text{pt} \]

\( \text{ellipticCylindrical } a \Rightarrow \text{elliptic } a \)
-- apply elliptic transformation to first 2 coordinates

\( \text{prolateSpheroidal } a \Rightarrow \)
\[ x \rightarrow \]
\[ \text{pt} := \text{copy}(x) \]
\[ xi := \text{elt}(x,1); \text{eta} := \text{elt}(x,2); \text{phi} := \text{elt}(x,3) \]
\[ \text{pt.1} := a \cdot \sinh(xi) \cdot \sin(\eta) \cdot \cos(\phi) \]
\[ \text{pt.2} := a \cdot \sinh(xi) \cdot \sin(\eta) \cdot \sin(\phi) \]
\[ \text{pt.3} := a \cdot \cosh(xi) \cdot \cos(\eta) \]
\[ \text{pt} \]

\( \text{oblateSpheroidal } a \Rightarrow \)
\[ x \rightarrow \]
\[ \text{pt} := \text{copy}(x) \]
\[ xi := \text{elt}(x,1); \text{eta} := \text{elt}(x,2); \text{phi} := \text{elt}(x,3) \]
\[ \text{pt.1} := a \cdot \sinh(xi) \cdot \sin(\eta) \cdot \cos(\phi) \]
\[ \text{pt.2} := a \cdot \cosh(xi) \cdot \cos(\eta) \cdot \sin(\phi) \]
\[ \text{pt.3} := a \cdot \sinh(xi) \cdot \sin(\eta) \]
\[ \text{pt} \]

\( \text{bipolar } a \Rightarrow \)
\[ x \rightarrow \]
\[ \text{pt} := \text{copy}(x) \]
\[ u := \text{elt}(x,1); v := \text{elt}(x,2) \]
\[ \text{pt.1} := a \cdot \sinh(v)/(\cosh(v)-\cos(u)) \]
\[ \text{pt.2} := a \cdot \sin(u)/(\cosh(v)-\cos(u)) \]
\[ \text{pt} \]

\( \text{bipolarCylindrical } a \Rightarrow \text{bipolar } a \)
-- apply bipolar transformation to first 2 coordinates

\( \text{toroidal } a \Rightarrow \)
\[ x \rightarrow \]
\[ \text{pt} := \text{copy}(x) \]
\[ u := \text{elt}(x,1); v := \text{elt}(x,2); \text{phi} := \text{elt}(x,3) \]
\[ \text{pt.1} := a \cdot \sinh(v)/(\cosh(v)-\cos(u)) \]
\[ \text{pt.2} := a \cdot \sin(u)/(\cosh(v)-\cos(u)) \]
\[ \text{pt.3} := a \cdot \sinh(v)/(\cosh(v)-\cos(u)) \]
\[ \text{pt} \]

\( \text{conical}(a,b) \Rightarrow \)
\[ x \rightarrow \]
\[ \text{pt} := \text{copy}(x) \]
lambda := elt(x,1); mu := elt(x,2); nu := elt(x,3)
pt.1 := lambda*mu*nu/(a*b)
pt.2 := lambda/a*sqrt((mu**2-a**2)*(nu**2-a**2)/(a**2-b**2))
pt.3 := lambda/b*sqrt((mu**2-b**2)*(nu**2-b**2)/(b**2-a**2))

package CRAPACK CRApackage

)--test--
--R--

)--S 1 of 1
)--R

)--R CRApackage(R: EuclideanDomain) is a package constructor
)--R Abbreviation for CRApackage is CRAPACK
)--R This constructor is exposed in this frame.
)--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CRAPACK

)--R

)--R---------------------------------- Operations ----------------------------------

--)modTree : (R,List(R)) -> List(R)
--)chineseRemainder : (List(R),List(R)) -> R
--)chineseRemainder : (List(List(R)),List(R)) -> List(R)
--)multiEuclideanTree : (List(R),R) -> List(R)

)--E 1

)--S 1 of 1

)--R

)--R (package CRAPACK)

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

)--R

--)
This package has no documentation

See Also:
  o )show CRApackage

---

CRAPACK

Exports:
  modTree  chineseRemainder  multiEuclideanTree

---

)abbrev package CRAPACK CRApackage
++ Description:
++ This package has no documentation
CRAPACK(R:EuclideanDomain): Exports == Implementation where
Exports == with
  modTree: (R,List R) -> List R
    ++ modTree(r,l) \undocumented{}
  chineseRemainder: (List R, List R) -> R
    ++ chineseRemainder(lv,lm) returns a value \axiom{v} such that, if
    ++ \axiom{x} is \axiom{lv.i} modulo \axiom{lm.i} for all \axiom{i}, then
+++ x is $v \mod (l_1 \cdot l_2 \cdot \ldots \cdot l_n)$.

\textbf{chineseRemainder}: \text{(List List \texttt{R}, List \texttt{R}) -> List \texttt{R}}

++ \text{chineseRemainder}([llv],\texttt{lm}) returns a list of values, each of which
++ \text{corresponds to the Chinese remainder of the associated element of}
++ \text{\texttt{llv}} and \text{\texttt{lm}}. \text{This is more efficient than applying}
++ \text{chineseRemainder several times.}

\textbf{multiEuclideanTree}: \text{(List \texttt{R}, \texttt{R}) -> List \texttt{R}}

++ \text{multiEuclideanTree}(l,r) \text{ undocumented}{}

\text{Implementation == add}

\texttt{BB:=BalancedBinaryTree}(\texttt{R})

\texttt{x:BB}

-- Definition for modular reduction mapping with several moduli
\texttt{modTree}(a,\texttt{lm}) ==
\hspace{1cm} t := \text{balancedBinaryTree}(\#\texttt{lm}, 0$\texttt{R}$)
\hspace{1cm} \text{setleaves}_!(t,\texttt{lm})
\hspace{1cm} \text{mapUp}_!(t,"*")
\hspace{1cm} \text{leaves mapDown}_!(t, a, "rem")

\texttt{chineseRemainder}(\texttt{llv}:\text{List \texttt{R}}, \texttt{lm}:\text{List \texttt{R}}):\text{\texttt{R}} ==
\hspace{1cm} \#\texttt{lm} ^= \#\texttt{lv} \Rightarrow \text{error "lists of moduli and values not of same length"}
\hspace{1cm} x := \text{balancedBinaryTree}(\#\texttt{lm}, 0$\texttt{R}$)
\hspace{1cm} x := \text{setleaves}_!(x, \texttt{lm})
\hspace{1cm} \text{mapUp}_!(x,"*")
\hspace{1cm} y := \text{balancedBinaryTree}(\#\texttt{lm}, 1$\texttt{R}$)
\hspace{1cm} y := \text{mapUp}_!(\\text{copy y},x,(a,b,c,d)\mapsto a*d + b*c)
\hspace{1cm} (u := \text{extendedEuclidean}(\\text{value y}, \\text{value x},1)) \text{ case "failed"} \Rightarrow
\hspace{1cm} \text{error "moduli not relatively prime"}
\hspace{1cm} \text{inv} := u . \text{coef1}
\hspace{1cm} \text{linv} := \text{modTree}(\text{inv}, \text{\texttt{lm}})
\hspace{1cm} l := [(u*v) \mod m \text{ for } v \text{ in } \texttt{lv} \text{ for } u \text{ in } \texttt{linv} \text{ for } m \text{ in } \texttt{lm}]
\hspace{1cm} y := \text{setleaves}_!(y, l)
\hspace{1cm} \text{value}(\text{mapUp}_!(y, x, (a,b,c,d)\mapsto a*d + b*c)) \mod \text{value(x)}

\texttt{chineseRemainder}([llv:List \texttt{R}, \texttt{lm}:List \texttt{R}]):List \texttt{R} ==
\hspace{1cm} x := \text{balancedBinaryTree}(\#\texttt{lm}, 0$\texttt{R}$)
\hspace{1cm} x := \text{setleaves}_!(x, \texttt{lm})
\hspace{1cm} \text{mapUp}_!(x,"*")
\hspace{1cm} y := \text{balancedBinaryTree}(\#\texttt{lm}, 1$\texttt{R}$)
\hspace{1cm} y := \text{mapUp}_!(\\text{copy y},x,(a,b,c,d)\mapsto a*d + b*c)
\hspace{1cm} (u := \text{extendedEuclidean}(\\text{value y}, \\text{value x},1)) \text{ case "failed"} \Rightarrow
\hspace{1cm} \text{error "moduli not relatively prime"}
\hspace{1cm} \text{inv} := u . \text{coef1}
\hspace{1cm} \text{linv} := \text{modTree}(\text{inv}, \text{\texttt{lm}})
\hspace{1cm} \text{retVal:List \texttt{R} := []}
\hspace{1cm} \text{for lv in llv repeat}
\hspace{1cm} l := [(u3*v) \mod m \text{ for } v \text{ in } \texttt{lv} \text{ for } u3 \text{ in } \texttt{linv} \text{ for } m \text{ in } \texttt{lm}]
\hspace{1cm} y := \text{setleaves}!(y, l)
\hspace{1cm} \text{retVal :=}
cons(value(mapUp!(y, x, (a,b,c,d)->a*d+b*c)) rem value(x),retVal)
reverse retVal

extEuclidean: (R, R, R) -> List R
extEuclidean(a, b, c) ==
  u := extendedEuclidean(a, b, c)
  u case "failed" => error [c, " not spanned by ", a, " and ",b]
  [u.coef2, u.coef1]

multiEuclideanTree(fl, rhs) ==
  x := balancedBinaryTree(#fl, rhs)
  x := setleaves_!(x, fl)
  mapUp_!(x,"*")
  leaves mapDown_!(x, rhs, extEuclidean)

———

— CRAPACK.dotabb —

"CRAPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CRAPACK"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"CRAPACK" -> "FLAGG"

———

package CYCLES CycleIndicators

—— CycleIndicators.input ——

)set break resume
)sys rm -f CycleIndicators.output
)spool CycleIndicators.output
)set message test on
)set message auto off
)clear all

--S 1 of 48
complete 1
--R
--R
--R (1) (1)
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 1
276

--S 2 of 48
complete 2
--R
--R
--R 1 1 2
--R (2) - (2) + - (1 )
--R 2 2
--R

Type: SymmetricPolynomial(Fraction(Integer))
--E 2

--S 3 of 48
complete 3
--R
--R
--R 1 1 1 3
--R (3) - (3) + - (2 1) + - (1 )
--R 3 2 6
--R

Type: SymmetricPolynomial(Fraction(Integer))
--E 3

--S 4 of 48
complete 7
--R
--R
--R

Type: SymmetricPolynomial(Fraction(Integer))
--E 4

--S 5 of 48
elementary 7
--R
--R
--R

Type: SymmetricPolynomial(Fraction(Integer))
--E 5
--R 6 5 4 6 8 12 120
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 9

--S 10 of 48
cap(complete 2**2, complete 2*complete 1**2)
--R
--R (10) 4
--R Type: Fraction(Integer)
--E 10

--S 11 of 48
cap(elementary 2**2, complete 2*complete 1**2)
--R
--R (11) 2
--R Type: Fraction(Integer)
--E 11

--S 12 of 48
cap(complete 3*complete 2*complete 1, complete 2**2*complete 1**2)
--R
--R (12) 24
--R Type: Fraction(Integer)
--E 12

--S 13 of 48
cap(elementary 3*elementary 2*elementary 1, complete 2**2*complete 1**2)
--R
--R (13) 8
--R Type: Fraction(Integer)
--E 13

--S 14 of 48
cap(complete 3*complete 2*complete 1, elementary 2**2*elementary 1**2)
--R
--R (14) 8
--R Type: Fraction(Integer)
--E 14

--S 15 of 48
eval(cup(complete 3*complete 2*complete 1, _
    cup(complete 2**2*complete 1**2, complete 2**3)))
--R
--R
--R (15) 1500
--R Type: Fraction(Integer)
--E 15

--S 16 of 48
square:=dihedral 4
--R
--R
--R 1 3 2 1 2 1 4
--R (16) - (4) + - (2 ) + - (2 1 ) + - (1 )
--R 4 8 4 8
--R Type: SymmetricPolynomial(Fraction(Integer))
--E 16

--S 17 of 48
cap(complete 2**2,square)
--R
--R
--R (17) 2
--R Type: Fraction(Integer)
--E 17

--S 18 of 48
cap(complete 3*complete 2**2,dihedral 7)
--R
--R
--R (18) 18
--R Type: Fraction(Integer)
--E 18

--S 19 of 48
cap(graphs 5,complete 7*complete 3)
--R
--R
--R (19) 4
--R Type: Fraction(Integer)
--E 19

--S 20 of 48
s(x) == powerSum(x)
--R
--R
--E 20

--S 21 of 48
cube:=(1/24)*(s 1**8+9*s 2**4 + 8*s 3**2*s 1**2+6*s 4**2)
--R Compiling function s with type PositiveInteger ->
--R SymmetricPolynomial(Fraction(Integer))
--R
--R 1 2 1 2 2 3 4 1 8
(21) - (4) + -(3 1) + -(2) + -(1) 
4 3 8 24 
Type: SymmetricPolynomial(Fraction(Integer)) 

S 22 of 48
cap(complete 4**2, cube) 
(22) 7 
Type: Fraction(Integer) 

S 23 of 48
cap(complete 2**3*complete 1**2, wreath(elementary 4, elementary 2)) 
(23) 7 
Type: Fraction(Integer) 

S 24 of 48
cap(complete 2**3*complete 1**2, wreath(elementary 4, complete 2)) 
(24) 17 
Type: Fraction(Integer) 

S 25 of 48
cap(complete 2**3*complete 1**2, wreath(complete 4, elementary 2)) 
(25) 10 
Type: Fraction(Integer) 

S 26 of 48
cap(complete 2**3*complete 1**2, wreath(complete 4, complete 2)) 
(26) 23 
Type: Fraction(Integer) 

S 27 of 48
x: ULS(FRAC INT,'x,0) := 'x 
(27) x
ZeroOrOne: INT -> ULS(FRAC INT, 'x, 0)

Integers: INT -> ULS(FRAC INT, 'x, 0)

ZeroOrOne n == 1+x**n

Integers n == 1/(1-x**n)
\)expose EVALCYC
--R
--I EvaluateCycleIndicators is now explicitly exposed in frame frame0
--E 34

--S 35 of 48
eval(ZeroOrOne, graphs 5)
--R
--R
--R 2 3 4 5 6 7 8 9 10 11
--R (34) 1 + x + 2x + 4x + 6x + 6x + 4x + 2x + x + x + O(x )
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 35

--S 36 of 48
eval(ZeroOrOne, dihedral 8)
--R
--R
--R 2 3 4 5 6 7 8
--R (35) 1 + x + 4x + 5x + 8x + 5x + 4x + x + x
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 36

--S 37 of 48
eval(Integers, complete 4)
--R
--R
--R (36)
--R 2 3 4 5 6 7 8 9 10 11
--R 1 + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x + O(x )
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 37

--S 38 of 48
eval(Integers, elementary 4)
--R
--R
--R (37)
--R 6 7 8 9 10 11 12 13 14 15 16
--R x + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x
--R +
--R 17
--R O(x )
--R Type: UnivariateLaurentSeries(Fraction(Integer),x,0)
--E 38

--S 39 of 48
eval(ZeroOrOne, cube)
--R
--R
\[
\begin{align*}
\text{--R} & \quad 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{--R} & (38) & 1 + x + 3x + 3x + 7x + 3x + 3x + x + x \\
\text{--R} & \quad \text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)} \\
\text{--E} & 39 \\
\text{--S} & 40 \text{ of } 48 \\
\text{eval(Integers,cube)} & \quad \\
\text{--R} & \quad \\
\text{--R} & (39) \\
\text{--R} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{--R} & 1 + x + 4x + 7x + 21x + 37x + 85x + 151x + 292x + 490x + 848x \\
\text{--R} + \\
\text{--R} & 11 \\
\text{--R} & O(x) \\
\text{--R} & \quad \text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)} \\
\text{--E} & 40 \\
\text{--S} & 41 \text{ of } 48 \\
\text{eval(Integers,graphs 5)} & \quad \\
\text{--R} & \quad \\
\text{--R} & (40) \\
\text{--R} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{--R} & 1 + x + 3x + 7x + 17x + 35x + 76x + 149x + 291x + 539x + 974x \\
\text{--R} + \\
\text{--R} & 11 \\
\text{--R} & O(x) \\
\text{--R} & \quad \text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)} \\
\text{--E} & 41 \\
\text{--S} & 42 \text{ of } 48 \\
\text{eval(ZeroGrüne ,graphs 15)} & \quad \\
\text{--R} & \quad \\
\text{--R} & (41) \\
\text{--R} & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
\text{--R} & 1 + x + 2x + 5x + 11x + 26x + 68x + 177x + 496x + 1471x + 4583x \\
\text{--R} + \\
\text{--R} & 11 \\
\text{--R} & O(x) \\
\text{--R} & \quad \text{Type: UnivariateLaurentSeries(Fraction(Integer),x,0)} \\
\text{--E} & 42 \\
\text{--S} & 43 \text{ of } 48 \\
cap(dihedral 30,complete 7*complete 8*complete 5*complete 10) & \quad \\
\text{--R} & \quad \\
\text{--R} & (42) & 49958972383320 \\
\text{--R} & \quad \text{Type: Fraction(Integer)}
\end{align*}
\]
sf3221 := SFunction [3,2,2,1]

--R

(43)

1 1 2 1 2 1 1 4 1 2

--R

(6 2) - (6 1) - (4) + (4 3 1) + (4 1) - (3 2)

12 12 16 12 24 36

--R

+ 12 2 1 2 1 3 1 5 1 4 1 3 2

(3 1) - (3 2 1) - (3 2 1) - (3 1) - (2) + (2 1)

36 24 36 72 192 48

--R

+ 12 4 1 6 1 8

(2 1) - (2 1) + (1)

96 144 576

Type: SymmetricPolynomial(Fraction(Integer))

--E 44

--S 45 of 48

cap (sf3221, complete 2**4)

(44) 3

Type: Fraction(Integer)

--E 45

--S 46 of 48

cap (sf3221, powerSum 1**8)

(45) 70

Type: Fraction(Integer)

--E 46

--S 47 of 48

eval (Integers, sf3221)

(46)

9 10 11 12 13 14 15 16 17 18

x + 3x + 7x + 14x + 27x + 47x + 79x + 126x + 196x + 294x

+ 19 20

432x + O(x)

Type: UnivariateLaurentSeries(Fraction(Integer), x, 0)

--E 47
CycleIndicators examples

Polya-Redfield enumeration by cycle indices.

This section is based upon the paper J. H. Redfield, ‘‘The Theory of Group-Reduced Distributions’’, American J. Math., 49 (1927) 433-455, and is an application of group theory to enumeration problems. It is a development of the work by P. A. MacMahon on the application of symmetric functions and Hammond operators to combinatorial theory.

The theory is based upon the power sum symmetric functions $s(i)$ which are the sum of the $i$-th powers of the variables. The cycle index of a permutation is an expression that specifies the sizes of the cycles of a permutation, and may be represented as a partition. A partition of a non-negative integer $n$ is a collection of positive integers called
its parts whose sum is \( n \). For example, the partition \((3^2 2 1^2)\) will
be used to represent \( s^2_3 s_2 s^2_1 \) and will indicate that the
permutation has two cycles of length 3, one of length 2 and two of
length 1. The cycle index of a permutation group is the sum of the
cycle indices of its permutations divided by the number of
permutations. The cycle indices of certain groups are provided.

The operation \texttt{complete} returns the cycle index of the symmetric group
of order \( n \) for argument \( n \). Alternatively, it is the \( n \)-th complete
homogeneous symmetric function expressed in terms of power sum
symmetric functions.

\[
\text{complete 1} \\
(1) \\
\text{Type: SymmetricPolynomial Fraction Integer}
\]

\[
\begin{align*}
\text{complete 2} \\
1 & \quad 1 & \quad 2 \\
- (2) & + & - (1) \\
2 & & 2 \\
\text{Type: SymmetricPolynomial Fraction Integer}
\end{align*}
\]

\[
\begin{align*}
\text{complete 3} \\
1 & \quad 1 & \quad 1 & \quad 3 \\
- (3) & + & - (2 1) & + & - (1) \\
3 & & 2 & & 6 \\
\text{Type: SymmetricPolynomial Fraction Integer}
\end{align*}
\]

\[
\begin{align*}
\text{complete 7} \\
1 & \quad 1 & \quad 1 & \quad 2 & \quad 1 & \quad 1 & \quad 1 & \quad 3 \\
- (7) & - & - (6 1) & + & -- (5 2) & + & -- (5 1) & + & -- (4 3) & + & - (4 2 1) & + & -- (4 1) \\
7 & & 6 & & 10 & & 10 & & 12 & & 8 & & 24 \\
+ & & & & & & & & & & & & \\
1 & & 2 & & 1 & & 2 & & 1 & & 2 & & 1 & & 4 & & 1 & & 3 & & 1 & & 2 & & 3 \\
-- (3 1) & + & -- (3 2) & + & -- (3 2 1) & + & -- (3 1) & + & -- (2 1) & + & -- (2 1) \\
18 & & 24 & & 12 & & 72 & & 48 & & 48 \\
+ & & & & & & & & & & & & \\
1 & & 5 & & 1 & & 7 \\
--- (2 1) & + & ---- (1) \\
240 & & 5040 \\
\text{Type: SymmetricPolynomial Fraction Integer}
\end{align*}
\]

The operation \texttt{elementary} computes the \( n \)-th elementary symmetric
function for argument \( n \).

\[
\begin{align*}
\text{elementary 7} \\
1 & \quad 1 & \quad 1 & \quad 1 & \quad 2 & \quad 1 & \quad 1 & \quad 1 & \quad 3 \\
- (7) & - & - (6 1) & - & - (5 2) & + & -- (5 1) & - & -- (4 3) & + & - (4 2 1) & - & -- (4 1) \\
7 & & 6 & & 10 & & 10 & & 12 & & 8 & & 24 \\
+ & & & & & & & & & & & & \\
\end{align*}
\]
The operation alternating returns the cycle index of the alternating group having an even number of even parts in each cycle partition.

\[
\text{alternating 7} = \frac{1}{7!} (1 \ 5 \ 1 \ 7) + \frac{1}{240} (1) = \frac{1}{240} (1) + \frac{1}{5040} (1)
\]

Type: SymmetricPolynomial Fraction Integer

The operation cyclic returns the cycle index of the cyclic group.

\[
\text{cyclic 7} = \frac{1}{7} (7) + \frac{1}{7} (1) = \frac{1}{7} (1)
\]

Type: SymmetricPolynomial Fraction Integer

The operation dihedral is the cycle index of the dihedral group.

\[
\text{dihedral 7} = \frac{1}{7} (7) + \frac{1}{2} (2 \ 1) + \frac{1}{14} (1) = \frac{1}{2} (2 \ 1) + \frac{1}{14} (1)
\]

Type: SymmetricPolynomial Fraction Integer

The operation graphs for argument n returns the cycle index of the group of permutations on the edges of the complete graph with n nodes induced by applying the symmetric group to the nodes.

\[
\text{graphs 5} = \frac{1}{6!} (6 \ 3 \ 1) + \frac{1}{5} (5) + \frac{1}{4} (4 \ 2) + \frac{1}{3} (3 \ 1) + \frac{1}{2} (2 \ 1) + \frac{1}{10} (1)
\]

Type: SymmetricPolynomial Fraction Integer

The cycle index of a direct product of two groups is the product of the cycle indices of the groups. Redfield provided two operations on
two cycle indices which will be called "cup" and "cap" here. The cup of two cycle indices is a kind of scalar product that combines monomials for permutations with the same cycles. The cap operation provides the sum of the coefficients of the result of the cup operation which will be an integer that enumerates what Redfield called group-reduced distributions.

We can, for example, represent complete 2 * complete 2 as the set of objects a a b b and complete 2 * complete 1 * complete 1 as c c d e.

This integer is the number of different sets of four pairs.

\[
cap(\text{complete 2}^2, \text{complete 2} \ast \text{complete 1}^2) = 4
\]

Type: Fraction Integer

For example,

\[
\text{a a b b a a b b a a b b a a b b c c d e c c d e c c d e c c d e c c d e c c d e c c d e c c}
\]

This integer is the number of different sets of four pairs no two pairs being equal.

\[
cap(\text{elementary 2}^2, \text{complete 2} \ast \text{complete 1}^2) = 2
\]

Type: Fraction Integer

For example,

\[
\text{a a b b a a b b c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d c e c c d e e f g.
\]

\[
cap(\text{complete 3} \ast \text{complete 2} \ast \text{complete 1}, \text{complete 2}^2 \ast \text{complete 1}^2) = 24
\]

Type: Fraction Integer

Here it is again, but with no equal pairs.

\[
cap(\text{elementary 3} \ast \text{elementary 2} \ast \text{elementary 1}, \text{complete 2}^2 \ast \text{complete 1}^2) = 8
\]

Type: Fraction Integer

\[
cap(\text{complete 3} \ast \text{complete 2} \ast \text{complete 1}, \text{elementary 2}^2 \ast \text{elementary 1}^2)
\]
The number of 6-triples, first from a a a b b c, second from d d e e f g, third from h h i i j j.

\[
eval(\text{cup(\text{complete } 3*\text{complete } 2*\text{complete } 1, \text{cup(\text{complete } 2*2*\text{complete } 1*2,\text{complete } 2*3))})}
\]
\[
1500
\]

The cycle index of vertices of a square is dihedral 4.

\[
square:=\text{dihedral } 4
\]
\[
1 \quad 3 \quad 2 \quad 1 \quad 2 \quad 1 \quad 4
\]
\[
- (4) \quad + \quad - (2) \quad + \quad - (2 \ 1) \quad + \quad - (1)
\]
\[
4 \quad 8 \quad 4 \quad 8
\]

The number of different squares with 2 red vertices and 2 blue vertices.

\[
\text{cap(\text{complete } 2*2, square)}
\]
\[
2
\]

The number of necklaces with 3 red beads, 2 blue beads and 2 green beads.

\[
\text{cap(\text{complete } 3*\text{complete } 2*2, \text{dihedral } 7)}
\]
\[
18
\]

The number of graphs with 5 nodes and 7 edges.

\[
\text{cap(\text{graphs } 5, \text{complete } 7*\text{complete } 3)}
\]
\[
4
\]

The cycle index of rotations of vertices of a cube.

\[
s(x) \quad = \quad \text{powerSum}(x)
\]

\[
\text{cube:=(1/24)*(s \text{**} 8+9*s \text{**} 4 + 8*s 3\text{**}2*s 1\text{**}2+6*s 4\text{**}2)}
\]
\[
1 \quad 2 \quad 1 \quad 2 \quad 2 \quad 3 \quad 4 \quad 1 \quad 8
\]
\[
- (4) \quad + \quad - (3 \ 1) \quad + \quad - (2) \quad + \quad - (1)
\]
\[
4 \quad 3 \quad 8 \quad 24
\]

The number of cubes with 4 red vertices and 4 blue vertices.

\[
\text{cap(\text{complete } 4*2, \text{cube})}
\]
The number of labeled graphs with degree sequence 2 2 2 1 1 with no loops or multiple edges.

\[ \text{cap(complete } 2^3 \otimes \text{complete } 1^2, \text{wreath(elementary } 4, \text{elementary } 2) \]  
\[ 7 \]  
Type: Fraction Integer

Again, but with loops allowed but not multiple edges.

\[ \text{cap(complete } 2^3 \otimes \text{complete } 1^2, \text{wreath(elementary } 4, \text{complete } 2) \]  
\[ 17 \]  
Type: Fraction Integer

Again, but with multiple edges allowed, but not loops.

\[ \text{cap(complete } 2^3 \otimes \text{complete } 1^2, \text{wreath(complete } 4, \text{elementary } 2) \]  
\[ 10 \]  
Type: Fraction Integer

Again, but with both multiple edges and loops allowed.

\[ \text{cap(complete } 2^3 \otimes \text{complete } 1^2, \text{wreath(complete } 4, \text{complete } 2) \]  
\[ 23 \]  
Type: Fraction Integer

Having constructed a cycle index for a configuration we are at liberty to evaluate the \(s_i\) components any way we please. For example we can produce enumerating generating functions. This is done by providing a function \(f\) on an integer \(i\) to the value required of \(s_i\), and then evaluating \(\text{eval}(f, \text{cycleindex})\).

\[ \text{x: ULS(FRAC INT,'x,0) := 'x} \]  
\[ \text{x} \]  
Type: UnivariateLaurentSeries(Fraction Integer,x,0)

\[ \text{ZeroOrOne: INT -> ULS(FRAC INT, 'x, 0)} \]  
Type: Void

\[ \text{Integers: INT -> ULS(FRAC INT, 'x, 0)} \]  
Type: Void

For the integers 0 and 1, or two colors.

\[ \text{ZeroOrOne n == 1+x**n} \]  
Type: Void

\[ \text{ZeroOrOne 5} \]
For the integers 0, 1, 2, ... we have this.

Integers \( n = 1/(1-x^n) \)

Integers 5

\[ 1 + x + x + O(x) \]

The coefficient of \( x^n \) is the number of graphs with 5 nodes and \( n \) edges.

Note that there is an eval function that takes two arguments. It has the signature:

\[ ((\text{Integer} \to \text{D1}), \text{SymmetricPolynomial Fraction Integer}) \to \text{D1} \]

This function is not normally exposed (it will not normally be considered in the list of eval functions) as it is only useful for this particular domain. To use it we ask that it be considered thus:

\( )\text{expose EVALCYC} \)

and now we can use it:

\[ \text{eval(ZeroOrOne, graphs 5)} \]

\[ 1 + x + 2x + 4x + 6x + 6x + 4x + 2x + x + x + O(x) \]

The coefficient of \( x^n \) is the number of necklaces with \( n \) red beads and \( n-8 \) green beads.

\[ \text{eval(ZeroOrOne, dihedral 8)} \]

\[ 1 + x + 4x + 5x + 8x + 5x + 4x + x + x \]

The coefficient of \( x^n \) is the number of partitions of \( n \) into 4 or fewer parts.

\[ \text{eval(\text{Integers, complete 4})} \]

\[ 1 + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x + O(x) \]
The coefficient of \( x^n \) is the number of partitions of \( n \) into 4 boxes containing ordered distinct parts.

\[
eval(\text{Integers}, \text{elementary 4})
\]

\[
\begin{align*}
&6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 + 14 + 15 + 16 \\
&x + x + 2x + 3x + 5x + 6x + 9x + 11x + 15x + 18x + 23x \\
&+ 17 \\
&O(x)
\end{align*}
\]

\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}

The coefficient of \( x^n \) is the number of different cubes with \( n \) red vertices and \( 8-n \) green ones.

\[
eval(\text{ZeroOrOne}, \text{cube})
\]

\[
\begin{align*}
&2 + 3 + 4 + 5 + 6 + 7 + 8 \\
&1 + x + 3x + 3x + 7x + 3x + 3x + x + x
\end{align*}
\]

\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}

The coefficient of \( x^n \) is the number of different cubes with integers on the vertices whose sum is \( n \).

\[
eval(\text{Integers}, \text{cube})
\]

\[
\begin{align*}
&2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 \\
&1 + x + 4x + 7x + 21x + 37x + 85x + 151x + 292x + 490x + 848x \\
&+ 11 \\
&O(x)
\end{align*}
\]

\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}

The coefficient of \( x^n \) is the number of graphs with 5 nodes and with integers on the edges whose sum is \( n \). In other words, the enumeration is of multigraphs with 5 nodes and \( n \) edges.

\[
eval(\text{Integers}, \text{graphs 5})
\]

\[
\begin{align*}
&2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 \\
&1 + x + 3x + 7x + 17x + 35x + 76x + 149x + 291x + 539x + 974x \\
&+ 11 \\
&O(x)
\end{align*}
\]

\text{Type: UnivariateLaurentSeries(Fraction Integer,x,0)}

Graphs with 15 nodes enumerated with respect to number of edges.

\[
eval(\text{ZeroOrOne}, \text{graphs 15})
\]

\[
\begin{align*}
&2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 \\
&1 + x + 2x + 5x + 11x + 26x + 68x + 177x + 496x + 1471x + 4583x \\
&+ 11 \\
&O(x)
\end{align*}
\]
Necklaces with 7 green beads, 8 white beads, 5 yellow beads and 10 red beads.

cap(dihedral 30, complete 7*complete 8*complete 5*complete 10)
49958972383320

Type: Fraction Integer

The operation SFunction is the S-function or Schur function of a partition written as a descending list of integers expressed in terms of power sum symmetric functions.

In this case the argument partition represents a tableau shape. For example 3,2,2,1 represents a tableau with three boxes in the first row, two boxes in the second and third rows, and one box in the fourth row. SFunction [3,2,2,1] counts the number of different tableaux of shape 3, 2, 2, 1 filled with objects with an ascending order in the columns and a non-descending order in the rows.

\[
\text{sf3221} := \text{SFunction} \{3,2,2,1\}
\]
\[
\begin{array}{llllllllllll}
1 & 1 & 2 & 1 & 2 & 1 & 1 & 4 & 1 & 2 \\
-- (6 2) & -- (6 1) & -- (4) & + & -- (4 3 1) & + & -- (4 1) & -- (3 2) \\
12 & 12 & 16 & 12 & 24 & 36
\end{array}
\]
\[
+ \begin{array}{llllllllllll}
1 & 2 & 2 & 1 & 2 & 1 & 3 & 1 & 5 & 1 & 4 & 1 & 3 & 2 \\
-- (3 1) & -- (3 2 1) & -- (3 2 1) & -- (3 1) & -- (2) & + & -- (2 1) \\
36 & 24 & 36 & 72 & 192 & 48
\end{array}
\]
\[
+ \begin{array}{llllllllllll}
1 & 2 & 4 & 1 & 6 & 1 & 8 \\
-- (2 1) & -- (2 1) & + & -- (1) \\
96 & 144 & 576
\end{array}
\]

This is the number filled with a a b b c c d d.

cap(sf3221, complete 2**4)
3

Type: Fraction Integer

The configurations enumerated above are:

a a b a a c a a d 
b c b b 
c d c d c c 
d d d

This is the number of tableaux filled with 1..8.

cap(sf3221, powerSum 1**8)
The coefficient of $x^n$ is the number of column strict reverse plane partitions of $n$ of shape 3 2 2 1.

\[
\begin{align*}
\text{eval(Integers, sf3221)} \\
9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\
x & + 3x & + 7x & + 14x & + 27x & + 47x & + 79x & + 126x & + 196x \\
+ \\
18 & 19 & 20 \\
294x & + 432x & + 0(x) \\
\end{align*}
\]

Type: UnivariateLaurentSeries(Fraction Integer, x, 0)

The smallest is

0 0 0
1 1
2 2
3

See Also:

\( )\)show CycleIndicators

CycleIndicators (CYCLES)

Exports:

alternating cap complete cup cyclic
dihedral elementary eval graphs powerSum
SFunction skewSFunction wreath

— package CYCLES CycleIndicators —
)abbrev package CYCLES CycleIndicators

++ Author: William H. Burge
++ Date Created: 1986
++ Date Last Updated: 11 Feb 1992
++ References: J.H.Redfield, 'The Theory of Group-Reduced Distributions',
++ G.Polya, 'Kombinatorische Anzahlbestimmungen fur Gruppen,
++ Graphen und chemische Verbindungen', Acta Math. 68
++ (1937) 145-254.
++ Description:
++ Polya-Redfield enumeration by cycle indices.

CycleIndicators: Exports == Implementation where
I ==> Integer
L ==> List
B ==> Boolean
SPOL ==> SymmetricPolynomial
PTN ==> Partition
RN ==> Fraction Integer
FR ==> Factored Integer
h ==> complete
s ==> powerSum
--a ==> elementary
alt ==> alternating
cyc ==> cyclic
dih ==> dihedral
ev == eval
Exports ==> with

complete: I -> SPOL RN
++\spad{complete n} is the \spad{n} th complete homogeneous
++ symmetric function expressed in terms of power sums.
++ Alternatively it is the cycle index of the symmetric
++ group of degree \spad{n}.

powerSum: I -> SPOL RN
++\spad{powerSum n} is the \spad{n} th power sum symmetric
++ function.

elementary: I -> SPOL RN
++\spad{elementary n} is the \spad{n} th elementary symmetric
++ function expressed in terms of power sums.

-- s2h: I -> SPOL RN--s to h

alternating: I -> SPOL RN
++\spad{alternating n} is the cycle index of the
++ alternating group of degree \spad{n}.

cyclic: I -> SPOL RN --cyclic group
\( \text{cyclic } n \) is the cycle index of the cyclic group of degree \( n \).

\( \text{dihedral: } I \rightarrow \text{SPOL RN} \quad \text{--dihedral group} \)
\( \text{dihedral } n \) is the cycle index of the dihedral group of degree \( n \).

\( \text{graphs: } I \rightarrow \text{SPOL RN} \)
\( \text{graphs } n \) is the cycle index of the group induced on the edges of a graph by applying the symmetric function to the \( n \) nodes.

\( \text{cap: } (\text{SPOL RN}, \text{SPOL RN}) \rightarrow \text{RN} \)
\( \text{cap}(s1, s2) \), introduced by Redfield, is the scalar product of two cycle indices.

\( \text{cup: } (\text{SPOL RN}, \text{SPOL RN}) \rightarrow \text{SPOL RN} \)
\( \text{cup}(s1, s2) \), introduced by Redfield, is the scalar product of two cycle indices, in which the power sums are retained to produce a cycle index.

\( \text{eval: } \text{SPOL RN} \rightarrow \text{RN} \)
\( \text{eval } s \) is the sum of the coefficients of a cycle index.

\( \text{wreath: } (\text{SPOL RN}, \text{SPOL RN}) \rightarrow \text{SPOL RN} \)
\( \text{wreath}(s1, s2) \) is the cycle index of the wreath product of the two groups whose cycle indices are \( s1 \) and \( s2 \).

\( \text{SFunction: } L I \rightarrow \text{SPOL RN} \)
\( \text{SFunction}(li) \) is the S-function of the partition \( li \) expressed in terms of power sum symmetric functions.

\( \text{skewSFunction: } (L I, L I) \rightarrow \text{SPOL RN} \)
\( \text{skewSFunction}(li1, li2) \) is the S-function of the partition difference \( li1 - li2 \) expressed in terms of power sum symmetric functions.

Implementation \( \Rightarrow \) add
import PartitionsAndPermutations
import IntegerNumberTheoryFunctions

\( \text{trm: } \text{PTN} \rightarrow \text{SPOL RN} \)
\( \text{trm pt} \equiv \text{monomial}(\text{inv(pdct(pt) :: RN)}, \text{pt}) \)

\( \text{list: } \text{Stream } L I \rightarrow \text{L L I} \)
\( \text{list st} \equiv \text{entries complete st} \)

\( \text{complete } i \equiv \)
\( \text{if } i=0 \)
then 1
else if i<0
    then 0
    else
        \text{+}/[\text{trm}(\text{partition } pt) \text{ for } pt \text{ in } \text{list}(\text{partitions } i)]

\text{even?}: \mathbb{Z} \rightarrow \mathbb{B}
\text{even? } li \equiv \text{even?}(\#([i \text{ for } i \text{ in } li \mid \text{even? } i]))

\text{alt } i \equiv
    2 \ast \text{+}/[\text{trm}(\text{partition } li) \text{ for } li \text{ in } \text{list}(\text{partitions } i) \mid \text{even? } li]

\text{elementary } i \equiv
    \text{if } i=0
    \text{then } 1
    \text{else if } i<0
    \text{then } 0
    \text{else}
    \text{+}/([\text{spol } \leftarrow \text{trm}(\text{partition } pt) \mid \text{even? } pt \Rightarrow \text{spol}; -\text{spol})
    \text{for } pt \text{ in } \text{list}(\text{partitions } i)]

\text{divisors}: \mathbb{Z} \rightarrow \mathbb{L} \mathbb{Z}
\text{divisors } n \equiv
    b := \text{factors}(n :: \mathbb{F}R)
    c := \text{concat}(1,\text{"append"}/
        [a.\text{factor}**j \text{ for } j \text{ in } 1..a.\text{exponent} \text{ for } a \text{ in } b]);
    \text{if } \#(b) = 1 \text{ then } c \text{ else } \text{concat}(n,c)

\text{ss}: (\mathbb{Z},\mathbb{Z}) \rightarrow \text{SPOL RN}
\text{ss}(n,m) \equiv
    li : \mathbb{L} \mathbb{Z} := [n \text{ for } j \text{ in } 1..m]
    \text{monomial}(1,\text{partition } li)
    s n \equiv ss(n,1)

\text{cyc } n \equiv
    n = 1 \Rightarrow s 1
    \text{+}/[(\text{eulerPhi}(i) / n) \ast ss(i,\text{numer}(n/i)) \text{ for } i \text{ in } \text{divisors } n]

\text{dih } n \equiv
    k := n \text{ quo } 2
    \text{odd? } n \Rightarrow (1/2) \ast cyc n + (1/2) \ast ss(2,k) \ast s 1
    (1/2) \ast cyc n + (1/4) \ast ss(2,k) + (1/4) \ast ss(2,k-1) \ast ss(1,2)

\text{trm2}: \mathbb{L} \mathbb{Z} \rightarrow \text{SPOL RN}
\text{trm2 } li \equiv
    lli := \text{powers}(li)$\text{PTN}$
    xx := 1/(\text{pdct } \text{partition } li)
    \text{prod } : \text{SPOL RN} := 1
    \text{for } ll \text{ in } lli \text{ repeat}
ln0 := first ll; ln1 := second ll
k := ln0 quo 2
c :=
  odd? ln0 => ss(ln0,ln1 * k)
  ss(k,ln1) * ss(ln0,ln1 * (k - 1))
c := c * ss(ln0,ln0 * ((ln1*(ln1 - 1)) quo 2))
prod2 : SPOL RN := 1
for r in lli | first(r) < ln0 repeat
  r0 := first r; r1 := second r
  prod2 := ss(lcm(r0,ln0),gcd(r0,ln0) * r1 * ln1) * prod2
  prod := c * prod2 * prod
xx * prod

graphs n == _+/[trm2 li for li in list(partitions n)]

cupp: (PTN,SPOL RN) -> SPOL RN
cupp(pt,spol) ==
  zero? spol => 0
  (dg := degree spol) < pt => 0
  dg = pt => (pdct pt) * monomial(leadingCoefficient spol,dg)
  cupp(pt,reduce spol)

cup(spoll1,spol2) ==
  zero? spoll1 => 0
  p := leadingCoefficient(spol1) * cupp(degree spol1,spol2)
  p + cup(reductum spol1,spol2)

ev spol ==
  zero? spol => 0
  leadingCoefficient(spol) + ev(reductum spol)

cap(spoll1,spol2) == ev cup(spoll1,spol2)

mtpol: (I,SPOL RN) -> SPOL RN
mtpol(n,spol) ==
  zero? spol => 0
  deg := partition [n*k for k in (degree spol)::L(I)]
  monomial(leadingCoefficient spol,deg) + mtpol(n,reductum spol)

fn2: I -> SPOL RN
evspol: ((I -> SPOL RN),SPOL RN) -> SPOL RN
evspol(fn2,spol) ==
  zero? spol => 0
  lc := leadingCoefficient spol
  prod := _+/[fn2 i for i in (degree spol)::L(I)]
  lc * prod + evspol(fn2,reductum spol)

wreath(spoll1,spol2) == evspol(x+->mtpol(x,spol2),spoll1)

hh: I -> SPOL RN  --symmetric group
hh n == if n=0 then 1 else if n<0 then 0 else h n
SFUn\text{ction } li==
a:\text{Matrix } SPOL RN:=\text{matrix }[[hh(k-j+i) \text{ for } k \text{ in } li \text{ for } j \text{ in } 1..\#li]
\text{ for } i \text{ in } 1..\#li]
determinant a

roundup: (L I, L I) \rightarrow L I
roundup(li1,li2)==
\#li1 > \#li2 \Rightarrow \text{roundup(li1,concat(li2,0))}
li2

skewSFunction(li1,li2)==
\#li1 < \#li2 =>
\text{error } "\text{skewSFunction: partition1 does not include partition2}"
li2:=\text{roundup}(li1,li2)
a:\text{Matrix } SPOL RN:=\text{matrix }[[hh(k-li2.i-j+i) 
\text{ for } k \text{ in } li1 \text{ for } j \text{ in } 1..\#li1 \text{ for } i \text{ in } 1..\#li1]
determinant a

——

— CYCLES.dotabb —

"CYCLES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CYCLES"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CYCLES" -> "PFECAT"

——

package CSTTOOLS CyclicStreamTools

— CyclicStreamTools.input —

)set break resume
)sys rm -f CyclicStreamTools.output
)spool CyclicStreamTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CyclicStreamTools
--R
--R CyclicStreamTools(S: Type,ST: LazyStreamAggregate(S)) is a package constructor
--R Abbreviation for CyclicStreamTools is CSTTOOLS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CSTTOOLS
--R
--R----------------------------- Operations ---------------------------------
--R computeCycleEntry : (ST,ST) -> ST  cycleElt : ST -> Union(ST,"failed")
--R computeCycleLength : ST -> NonNegativeInteger
--R
--E 1
)
spool
)lisp (bye)

---

— CyclicStreamTools.help —

====================================================================
CyclicStreamTools examples
====================================================================

This package provides tools for working with cyclic streams.

See Also:
  o )show CyclicStreamTools

---

CyclicStreamTools (CSTTOOLS)

Exports:
computeCycleEntry  computeCycleLength  cycleElt
— package CSTTOOLS CyclicStreamTools —

)abbrev package CSTTOOLS CyclicStreamTools
++ Author: Clifton J. Williamson
++ Date Created: 5 December 1989
++ Date Last Updated: 5 December 1989
++ Description:
++ This package provides tools for working with cyclic streams.

CyclicStreamTools(S,ST): Exports == Implementation where
  S : Type
  ST : LazyStreamAggregate S

Exports ==> with

cycleElt: ST -> Union(ST,"failed")
  ++ cycleElt(s) returns a pointer to a node in the cycle if the stream
  ++ s is cyclic and returns "failed" if s is not cyclic
  ++
  ++X p:=repeating([1,2,3])
  ++X q:=cons(4,p)
  ++X cycleElt q
  ++X r:=[1,2,3]::Stream(Integer)
  ++X cycleElt r

cycleElt x ==
y := x
for i in 0.. repeat
  (explicitlyEmpty? y) or (lazy? y) => return "failed"

computeCycleLength: ST -> NonNegativeInteger
  ++ computeCycleLength(s) returns the length of the cycle of a
  ++ cyclic stream t, where s is a pointer to a node in the
  ++ cyclic part of t.
  ++
  ++X p:=repeating([1,2,3])
  ++X q:=cons(4,p)
  ++X computeCycleLength(cycleElt(q))

computeCycleEntry: (ST,ST) -> ST
  ++ computeCycleEntry(x,cycElt), where cycElt is a pointer to a
  ++ node in the cyclic part of the cyclic stream x, returns a
  ++ pointer to the first node in the cycle
  ++
  ++X p:=repeating([1,2,3])
  ++X q:=cons(4,p)
  ++X computeCycleEntry(q,cycleElt(q))

Implementation ==> add

  cycleElt x ==
  y := x
  for i in 0.. repeat
    (explicitlyEmpty? y) or (lazy? y) => return "failed"

y := rst y
if odd? i then x := rst x
eq?(x,y) => return y

computeCycleLength cycElt ==
i : NonNegativeInteger
y := cycElt
for i in 1.. repeat
  y := rst y
  eq?(y,cycElt) => return i

computeCycleEntry(x,cycElt) ==
y := rest(x, computeCycleLength cycElt)
repeat
  eq?(x,y) => return x
  x := rst x ; y := rst y

——

—— CSTTOOLS.dotabb ——
"CSTTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CSTTOOLS"]
"LZSTAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LZSTAGG"]
"CSTTOOLS" -> "LZSTAGG"

——

package CYCLOTOM CyclotomicPolynomialPackage

—— CyclotomicPolynomialPackage.input ——

)set break resume
)sys rm -f CyclotomicPolynomialPackage.output
)spool CyclotomicPolynomialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CyclotomicPolynomialPackage
--R
--R CyclotomicPolynomialPackage is a package constructor
--R Abbreviation for CyclotomicPolynomialPackage is CYCLOTOM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CYCLOTOM
--R
--R-------------------------------- Operations --------------------------------
--R cyclotomic : Integer -> SparseUnivariatePolynomial(Integer)
--R cyclotomicDecomposition : Integer -> List(SparseUnivariatePolynomial(Integer))
--R cyclotomicFactorization : Integer -> Factored(SparseUnivariatePolynomial(Integer))
--R
--E 1

)spool
)lisp (bye)

---

— CyclotomicPolynomialPackage.help —

====================================================================
CyclotomicPolynomialPackage examples
====================================================================

This package has no description

See Also:
o )show CyclotomicPolynomialPackage

---

CyclotomicPolynomialPackage (CYCLOTOM)

Exports:
cyclotomic  cyclotomicDecomposition  cyclotomicFactorization

— package CYCLOTOM CyclotomicPolynomialPackage —
)abbrev package CYCLOTOM CyclotomicPolynomialPackage
++ Description:
++ This package has no description

CyclotomicPolynomialPackage: public == private where
SUP ==> SparseUnivariatePolynomial(Integer)
LSUP ==> List(SUP)
NNI ==> NonNegativeInteger
FR ==> Factored SUP
IFP ==> IntegerFactorizationPackage Integer

public == with
cyclotomicDecomposition: Integer -> LSUP
  ++ cyclotomicDecomposition(n) \undocumented{}
cyclotomic: Integer -> SUP
  ++ cyclotomic(n) \undocumented{}
cyclotomicFactorization: Integer -> FR
  ++ cyclotomicFactorization(n) \undocumented{}

private == add
cyclotomic(n:Integer): SUP ==
x,y,z,l: SUP
g := factors factor(n)$IFP
  --Now, for each prime in the factorization apply recursion
l := monomial(1,1) - monomial(1,0)
for u in g repeat
  l := (monicDivide(multiplyExponents(l,u.factor::NNI),l)).quotient
  if u.exponent>1 then
    l := multiplyExponents(l,((u.factor)**((u.exponent-1)::NNI))::NNI)
l
  cyclotomicDecomposition(n:Integer):LSUP ==
x,y,z: SUP
l,ll,m: LSUP
rr: Integer
g := factors factor(n)$IFP
l := [monomial(1,1) - monomial(1,0)]
  --Now, for each prime in the factorization apply recursion
for u in g repeat
  m := [monicDivide(multiplyExponents(z,u.factor::NNI),z)).quotient for z in l]
  for rr in 1..(u.exponent-1) repeat
    l := append(l,m)
  m := [multiplyExponents(z,u.factor::NNI) for z in m]
l := append(l,m)
l
  cyclotomicFactorization(n:Integer):FR ==
f: SUP
fr : FR := 1$FR
for f in cyclotomicDecomposition(n) repeat
    fr := fr * primeFactor(f,1$Integer)
fr

---

— CYCLOTOM.dotabb —

"CYCLOTOM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CYCLOTOM"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CYCLOTOM" -> "PFECAT"

---

package CAD CylindricalAlgebraicDecompositionPackage

--- CylindricalAlgebraicDecompositionPackage.input —

)set break resume
)sys rm -f CylindricalAlgebraicDecompositionPackage.output
)spool CylindricalAlgebraicDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show CylindricalAlgebraicDecompositionPackage
--R
--R CylindricalAlgebraicDecompositionPackage(TheField: RealClosedField) is a package constructor
--R Abbreviation for CylindricalAlgebraicDecompositionPackage is CAD
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CAD
--R
--R---------------------------------------------------------- Operations -------------------------------
--R coefficientSet : SparseUnivariatePolynomial(Polynomial(TheField)) -> List(Polynomial(TheField))
--R cylindricalDecomposition : List(Polynomial(TheField)) -> List(Cell(TheField))
--R cylindricalDecomposition : (List(Polynomial(TheField)),List(Symbol)) -> List(Cell(TheField))
--R discriminantSet : List(SparseUnivariatePolynomial(Polynomial(TheField))) -> List(Polynomial(TheField))
--R principalSubResultantSet : (SparseUnivariatePolynomial(Polynomial(TheField)),SparseUnivariatePolynomial(Polynomial(TheField))) -> List(Polynomial(TheField))
--R projectionSet : List(SparseUnivariatePolynomial(Polynomial(TheField))) -> List(Polynomial(TheField))
--R resultantSet : List(SparseUnivariatePolynomial(Polynomial(TheField)))) -> List(Polynomial(TheField))
--R specialise : (List(Polynomial(TheField)),Cell(TheField)) -> List(SparseUnivariatePolynomial(TheField))
--R
--E 1
CylindricalAlgebraicDecompositionPackage (CAD)

Exports:
  coefficientSet  cylindricalDecomposition  cylindricalDecomposition
  discriminantSet  principalSubResultantSet  projectionSet
  resultantSet  specialise

— package CAD CylindricalAlgebraicDecompositionPackage —

)abbrev package CAD CylindricalAlgebraicDecompositionPackage

CylindricalAlgebraicDecompositionPackage(TheField) : PUB == PRIV where

  TheField : RealClosedField

  ThePols ==> Polynomial(TheField)
P ==> ThePols
BUP ==> SparseUnivariatePolynomial(TheField)
RUP ==> SparseUnivariatePolynomial(ThePols)

Z ==> Integer
N ==> NonNegativeInteger

CELL ==> Cell(TheField)
SCELL ==> SimpleCell(TheField,BUP)

PUB == with

cylindricalDecomposition: List P -> List CELL

cylindricalDecomposition: (List(P),List(Symbol)) -> List CELL

projectionSet: (List RUP) -> List P

coefficientSet: RUP -> List P

discriminantSet : List RUP -> List(P)

resultantSet : List RUP -> List P

principalSubResultantSet : (RUP,RUP) -> List P

specialise : (List(ThePols),CELL) -> List(BUP)

PRIV == add

cylindricalDecomposition(lpols) ==
  lv : List(Symbol) := []
  for pol in lpols repeat
    ground?(pol) => "next pol"
    lv := removeDuplicates(append(variables(pol),lv))
    lv := reverse(sort(lv))
  cylindricalDecomposition(lpols,lv)

cylindricalDecomposition(lpols,lvars) ==
  lvars = [] => error("CAD: cylindricalDecomposition: empty list of vars")
  mv := first(lvars)
  lv := rest(lvars)
  lv = [] =>
    lp1 := [ univariate(pol) for pol in lpols ]
    scells := allSimpleCells(lp1,mv)$SCELL
    [ makeCell([scell]) for scell in scells ]
  lpols1 := projectionSet [univariate(pol,mv) for pol in lpols]
  previousCad := cylindricalDecomposition(lpols1,lv)
  res : List(CELL) := []
  for cell in previousCad repeat
lspec := specialise(lpols,cell)  
scells := allSimpleCells(lspec,mv)  
res := append(res,[makeCell(scell,cell) for scell in scells])  
res

PACK1 ==> CylindricalAlgebraicDecompositionUtilities(ThePols,RUP)  
PACK2 ==> CylindricalAlgebraicDecompositionUtilities(TheField,BUP)

specialise(lpols,cell) ==  
lpols = [] => error("CAD: specialise: empty list of pols")  
sp := samplePoint(cell)  
vl := variablesOf(cell)  
res : List(BUP) := []  
for pol in lpols repeat  
p1 := univariate(eval(pol,vl,sp))  
if degree(p1) = 0 then "next pol"  
res := cons(p1,res)

coefficientSet(pol) ==  
res : List(ThePols) := []  
for c in coefficients(pol) repeat  
if ground?(c) then return(res)  
res := cons(c,res)

SUBRES ==> SubResultantPackage(ThePols,RUP)  
discriminantSet(lpols) ==  
res : List(ThePols) := []  
for p in lpols repeat  
v := subresultantVector(p,differentiate(p))$SUBRES  
if not(zero?(degree(v.0))) then return(error "Bad discriminant")  
d := leadingCoefficient(v.0)  
if not(zero?(d)) then res := cons(d,res)

principalSubResultantSet(p,q) ==  
if degree(p) < degree(q) then (p,q) := (q,p)  
if degree(p) = degree(q) then (p,q) := (q,pseudoRemainder(p,q))  
v := subresultantVector(p,q)$SUBRES  
[coefficient(v.i,i) for i in 0..((#v)-2)::N]]

resultantSet(lpols) ==  
res : List(ThePols) := []  
laux := lpols  
for p in lpols repeat  
laux := rest(laux)
for q in laux repeat
    r : ThePols := first(principalSubResultantSet(p,q))
    zero?(r) => return(error "Non relatively prime polynomials")
    if not(ground? r) then res := cons(r,res)
res

projectionSet(lpols) ==
res : List(ThePols) := []
for p in lpols repeat
    c := content(p)
    ground?(c) => "next p"
    res := cons(c,res)
lp1 := [primitivePart p for p in lpols]
f : ((RUP,RUP) -> Boolean) := (degree(#1) <= degree(#2))
lp1 := sort(f,lp1)
lsqfrb := squareFreeBasis(lp1)$PACK1
lsqfrb := sort(f,lsqfrb)
for p in lp1 repeat
    res := append(res,coefficientSet(p))
    res := append(res,discriminantSet(lsqfrb))
    append(res,resultantSet(lsqfrb))

package CADU CylindricalAlgebraicDecompositionUtil-
ities

Test are some standard tools which are needed to compute with univariate polynomials.
A gcd basis for a set of polynomials is a set of pairwise relatively prime polynomials which
all divide the original set and whose product is the same than the product of the original
set.
A square free basis for a list of polynomials is just a list of square free polynomials which
are pairwise relatively primes and have the same roots than the original polynomials.

— CylindricalAlgebraicDecompositionUtilities.input —
CylindricalAlgebraicDecompositionUtilities

Test are some standard tools which are needed to compute with univariate polynomials.

A gcd basis for a set of polynomials is a set of pairwise relatively prime polynomials which all divide the original set and whose product is the same than the product of the original set.

A square free basis for a list of polynomials is just a list of square free polynomials which are pairwise relatively primes and have the same roots than the original polynomials.

See Also:

- )show CylindricalAlgebraicDecompositionUtilities
CylindricalAlgebraicDecompositionUtilities (CADU)

Exports:
gcdBasis gcdBasisAdd squareFreeBasis

— package CADU CylindricalAlgebraicDecompositionUtilities —

)abbrev package CADU CylindricalAlgebraicDecompositionUtilities
CylindricalAlgebraicDecompositionUtilities(R,P) : PUB == PRIV where

R : GcdDomain
P : UnivariatePolynomialCategory(R)

PUB == with
  squareFreeBasis : List(P) -> List(P)
  ++
  gcdBasis : List(P) -> List(P)
  ++ decompose a list of polynomials into pairwise relatively prime polynomials
  ++
  gcdBasisAdd : (P,List(P)) -> List(P)
  ++ add one polynomial to list of pairwise relatively prime polynomials

PRIV == add

squareFreeBasis(lpolis) ==
lpolis = [] => []
pol := first(lpolis)
sqpol := unitCanonical(squareFreePart(pol))
gcdBasis(squareFreeBasis(rest(lpolis)))

gcdBasisAdd(p,lpolis) ==
(degree(p) = 0) => lpolis
null lpolis => [unitCanonical p]
p1 := first(lpolis)
g := gcd(p,p1)
(degree(g) = 0) => cons(p1,gcdBasisAdd(p,rest lpolis))
p := (p exquo g)::P

\begin{verbatim}
p1 := (p1 exquo g)::P
basis := gcdBasisAdd(p, rest(lpols))
if degree(p1) > 0 then basis := cons(p1, basis)
gcdBasisAdd(g, basis)

gcdBasis(lpols) ==
  (#lpols <= 1) => lpols
  basis := gcdBasis(rest lpols)
  gcdBasisAdd(first(lpols), basis)
\end{verbatim}

\textbf{— CADU.dotabb —}

"CADU" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CADU"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"CADU" -> "PFECAT"
Chapter 5

Chapter D

package DFINTTLS DefiniteIntegrationTools

| DefiniteIntegrationTools.input |

)set break resume
)sys rm -f DefiniteIntegrationTools.output
)spool DefiniteIntegrationTools.output
)set message test on
)set message auto off
)clear all

--) 1 of 1
)show DefiniteIntegrationTools
)--
)-- DefiniteIntegrationTools(R: Join(GcdDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),F: Join(TranscendentalFunctionCategory,AlgebraicallyClosedFunctionSpace(R))) is a package constructor
)-- Abbreviation for DefiniteIntegrationTools is DFINTTLS
)-- This constructor is not exposed in this frame.
)-- Issue )edit bookvol10.4.pamphlet to see algebra source code for DFINTTLS
)--
)--------------------------------- Operations --------------------------------
)-- ignore? : String -> Boolean
)-- checkForZero : (Polynomial(R),Symbol,OrderedCompletion(F),OrderedCompletion(F),Boolean) -> Union(Boolean,"failed")
)-- checkForZero : (SparseUnivariatePolynomial(F),OrderedCompletion(F),OrderedCompletion(F),Boolean) -> Union(Boolean,"failed")
)-- computeInt : (Kernel(F),F,OrderedCompletion(F),OrderedCompletion(F),Boolean) -> Union(OrderedCompletion(F),"failed")
)--
)--E 1

)spool
)lisp (bye)

-------------

313
DefiniteIntegrationTools (DFINNTLS)

Exports:
ignore? checkForZero computeInt

— package DFINNTLS DefiniteIntegrationTools —

)abbrev package DFINNTLS DefiniteIntegrationTools
++ Author: Manuel Bronstein
++ Date Created: 15 April 1992
++ Date Last Updated: 24 February 1993
++ Description:
++ \spadtype{DefiniteIntegrationTools} provides common tools used
++ by the definite integration of both rational and elementary functions.

DefiniteIntegrationTools(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, RetractableTo Integer,
           LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory, 
    AlgebraicallyClosedFunctionSpace R)

B ==> Boolean
Z ==> Integer
Q ==> Fraction Z
SE ==> Symbol
P ==> Polynomial R
RF ==> Fraction P
UP ==> SparseUnivariatePolynomial F
K ==> Kernel F
OFE ==> OrderedCompletion F
UPZ ==> SparseUnivariatePolynomial Z
UPQ ==> SparseUnivariatePolynomial Q
REC ==> Record(left:Q, right:Q)
REC2 ==> Record(endpoint:Q, dir:Z)
U ==> Union(fin:REC, halfinf:REC2, all:"all", failed:"failed")
IGNOR ==> "noPole"

Exports ==> with
    ignore?: String -> B
    ++ ignore?(s) is true if s is the string that tells the integrator
    ++ to assume that the function has no pole in the integration interval.
    computeInt: (K, F, OFE, OFE, B) -> Union(OFE, "failed")
    ++ computeInt(x, g, a, b, eval?) returns the integral of \spad{f} for x
    ++ between a and b, assuming that g is an indefinite integral of
    ++ \spad{f} and \spad{f} has no pole between a and b.
    ++ If \spad{eval?} is true, then \spad{g} can be evaluated safely
    ++ at \spad{a} and \spad{b}, provided that they are finite values.
    ++ Otherwise, limits must be computed.
    checkForZero: (P, SE, OFE, OFE, B) -> Union(B, "failed")
    ++ checkForZero(p, x, a, b, incl?) is true if p has a zero for x between
    ++ a and b, false otherwise, "failed" if this cannot be determined.
    ++ Check for a and b inclusive if incl? is true, exclusive otherwise.
    checkForZero: (UP, OFE, OFE, B) -> Union(B, "failed")
    ++ checkForZero(p, a, b, incl?) is true if p has a zero between
    ++ a and b, false otherwise, "failed" if this cannot be determined.
    ++ Check for a and b inclusive if incl? is true, exclusive otherwise.

Implementation ==> add
    import RealZeroPackage UPZ
    import InnerPolySign(F, UP)
    import ElementaryFunctionSign(R, F)
    import PowerSeriesLimitPackage(R, F)
    import UnivariatePolynomialCommonDenominator(Z, Q, UPQ)

mkLogPos : F -> F
keeprec? : (Q, REC) -> B
negative : F -> Union(B, "failed")
mkKerPos : K -> Union(F, "positive")
posRoot : (UP, B) -> Union(B, "failed")
realRoot : UP -> Union(B, "failed")
var : UP -> Union(Z, "failed")
maprat : UP -> Union(UPZ, "failed")
variation : (UP, F) -> Union(Z, "failed")
infeval : (UP, OFE) -> Union(F, "failed")
checkHalfAx : (UP, F, Z, B) -> Union(B, "failed")
findLimit : (F, K, OFE, String, B) -> Union(OFE, "failed")
checkBudan : (UP, OFE, OFE, B) -> Union(B, "failed")
checkDeriv : (UP, OFE, OFE) -> Union(B, "failed")
sameSign : (UP, OFE, OFE) -> Union(B, "failed")
intrat : (OFE, OFE) -> U
findRealZero: (UPZ, U, B) -> List REC

variation(p, a) == var p(monomial(1, 1)$UP - a::UP)
keeprec?(a, rec) == (a > rec.right) or (a < rec.left)

checkHalfAx(p, a, d, incl?) ==
    posRoot(p(d * (monomial(1, 1)$UP - a::UP)), incl?)
ignore? str ==
    str = IGNOR => true
    error "integrate: last argument must be 'noPole'"

computeInt(k, f, a, b, eval?) ==
    is?(f, "integral"::SE) => "failed"
    if not eval? then f := mkLogPos f
    ((ib := findLimit(f, k, b, "left", eval?)) case "failed") or
    ((ia := findLimit(f, k, a, "right", eval?)) case "failed") => "failed"
    infinite?(ia::OFE) and (ia::OFE = ib::OFE) => "failed"
    ib::OFE - ia::OFE

findLimit(f, k, a, dir, eval?) ==
    r := retractIfCan(a)@Union(F, "failed")
    r case F =>
        eval? => mkLogPos(eval(f, k, r::F))::OFE
        (u := limit(f, equation(k::F, r::F), dir)) case OFE => u::OFE
        "failed"
    (u := limit(f, equation(k::F::OFE, a))) case OFE => u::OFE
    "failed"

mkLogPos f ==
    lk := empty()$List(K)
    lv := empty()$List(F)
    for k in kernels f | is?(k, "log"::SE) repeat
        if (v := mkKerPos k) case F then
            lk := concat(k, lk)
            lv := concat(v::F, lv)
    eval(f, lk, lv)
mkKerPos k ==
  (u := negative(f := first argument k)) case "failed" =>
    log(f**2) / (2::F)
  u::B => log(-f)
"positive"

negative f ==
  (u := sign f) case "failed" => "failed"
  u::Z < 0

checkForZero(p, x, a, b, incl?) ==
  checkForZero(
    map(s+->s::F, univariate(p, x))_ $SparseUnivariatePolynomialFunctions2(P, F),
    a, b, incl?)

checkForZero(q, a, b, incl?) ==
  ground? q => false
  (d := maprat q) case UPZ and not((i := intrat(a, b)) case failed) =>
    not empty? findRealZero(d::UPZ, i, incl?)
  (u := checkBudan(q, a, b, incl?)) case "failed" =>
    incl? => checkDeriv(q, a, b)
    "failed"
  u::B

maprat p ==
  ans:UPQ := 0
  while p ^= 0 repeat
    (r := retractIfCan(c := leadingCoefficient p)@Union(Q,"failed"))
    case "failed" => return "failed"
    ans := ans + monomial(r::Q, degree p)
    p := reductum p
  map(numer,(splitDenominator ans).num
    )$SparseUnivariatePolynomialFunctions2(Q, Z)

intrat(a, b) ==
  (n := whatInfinity a) ^= 0 =>
  (r := retractIfCan(b)@Union(F,"failed")) case "failed" => ["all"]
  (q := retractIfCan(r::F)@Union(Q, "failed")) case "failed" =>
    ["failed"]
    [[q::Q, n]]
  (q := retractIfCan(retract(a)@F)@Union(Q,"failed")) case "failed" =>
    ["failed"]
    [[q::Q, n]]
  (t := retractIfCan(retract(b)@F)@Union(Q,"failed")) case "failed" =>
    ["failed"]
    [[q::Q, t::Q]]

findRealZero(p, i, incl?) ==
  i case fin =>
1 := realZeros(p, r := i.fin)
incl? => l
select_!(s+->keeprec?(r.left, s) and keeprec?(r.right, s), l)
i case all => realZeros p
i case halfinf =>
empty?(1 := realZeros p) => empty()
bounds:REC :=
  i.halfinf.dir > 0 => [i.halfinf.endpoint, "max"/[t.right for t in l]]
  "min"/[t.left for t in l], i.halfinf.endpoint]
l := [u::REC for t in l | (u := refine(p, t, bounds)) case REC]
incl? => l
ep := i.halfinf.endpoint
select_!(s+->keeprec?(ep, s), l)
error "findRealZero: should not happen"

checkBudan(p, a, b, incl?) ==
r := retractIfCan(b)@Union(F, "failed")
(n := whatInfinity a) ^= 0 =>
r case "failed" => realRoot p
checkHalfAx(p, r::F, n, incl?)
(za? := zero? p(aa := retract(a)@F)) and incl? => true
(n := whatInfinity b) ^= 0 => checkHalfAx(p, aa, n, incl?)
(zb? := zero? p(bb := r::F)) and incl? => true
(va := variation(p, aa)) case "failed" or
  (vb := variation(p, bb)) case "failed" => "failed"

m:Z := 0
if za? then m := inc m
if zb? then m := inc m
odd?(v := va::Z - vb::Z) => -- p has an odd number of roots
  incl? or even? m => true
--
  one? v => false
  (v = 1) => false
  "failed"

zero? v => false -- p has no roots
--
  one? m => true
  (m = 1) => true
  "failed"

checkDeriv(p, a, b) ==
(r := retractIfCan(p)@Union(F, "failed")) case F => zero?(r::F)
(s := sameSign(p, a, b)) case "failed" => "failed"
s::B =>
  -- p has the same nonzero sign at a and b
  (u := checkDeriv(differentiate p,a,b)) case "failed" => "failed"
  u::B => "failed"
false
true

realRoot p ==
(b := posRoot(p, true)) case "failed" => "failed"
b::B => true
\begin{verbatim}

posRoot(p(p - monomial(1, 1)$UP), true)

sameSign(p, a, b) ==
  (ea := infeval(p, a)) case "failed" => "failed"
  (eb := infeval(p, b)) case "failed" => "failed"
  (s := sign(ea::F * eb::F)) case "failed" => "failed"
  s::Z > 0

-- returns true if p has a positive root. Include 0 is incl0? is true
posRoot(p, incl0?) ==
  (z0? := zero?(coefficient(p, 0))) and incl0? => true
  (v := var p) case "failed" => "failed"
  odd?(v::Z) => -- p has an odd number of positive roots
    incl0? or not(z0?) => true
  --
  one?(v::Z) => false
  (v::Z) = 1 => false
  "failed"
  zero?(v::Z) => false -- p has no positive roots
  z0? => true -- p has an even number > 0 of positive roots
  "failed"

infeval(p, a) ==
  zero?(n := whatInfinity a) => p(retract(a)@F)
  (u := signAround(p, n, sign)) case "failed" => "failed"
  u::Z::F

var q ==
  i:Z := 0
  (lastCoef := negative leadingCoefficient q) case "failed" =>
    "failed"
  while ((q := reductum q) ^= 0) repeat
    (next := negative leadingCoefficient q) case "failed" =>
      return "failed"
    if ((not(lastCoef::B)) and next::B) or
      ((not(next::B)) and lastCoef::B) then i := i + 1
    lastCoef := next
    i

\end{verbatim}
package DEGRED DegreeReductionPackage

--- DegreeReductionPackage.input ---

)set break resume
)sys rm -f DegreeReductionPackage.output
)spool DegreeReductionPackage.output
)set message test on
)set message auto off
)clear all

--) 1 of 1
)show DegreeReductionPackage
--R
--R DegreeReductionPackage(R1: Ring,R2: Join(IntegralDomain,OrderedSet)) is a package constructor
--R Abbreviation for DegreeReductionPackage is DEGRED
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DEGRED
--R
--R-------------------------------- Operations --------------------------------
--R expand : (Expression(R2),PositiveInteger) -> List(Expression(R2))
--R reduce : SparseUnivariatePolynomial(R1) -> Record(pol: SparseUnivariatePolynomial(R1),deg: PositiveInteger)
--R
--E 1

)spool
)lisp (bye)

---

--- DegreeReductionPackage.help ---

====================================================================
DegreeReductionPackage examples
====================================================================

This package has no description

See Also:
 o )show DegreeReductionPackage

---
DegreeReductionPackage (DEGRED)

Exports:
expand  reduce

— package DEGRED DegreeReductionPackage —

)abbrev package DEGRED DegreeReductionPackage
++ Description:
++ This package has no description

DegreeReductionPackage(R1, R2): Cat == Capsule where
   R1: Ring
   R2: Join(IntegralDomain,OrderedSet)
   I  ==> Integer
   PI ==> PositiveInteger
   UP ==> SparseUnivariatePolynomial
   RE ==> Expression R2

Cat == with
   reduce: UP R1 -> Record(pol: UP R1, deg: PI)
   ++ reduce(p) \ undocumented{}
   expand: (RE, PI) -> List RE
   ++ expand(f,n) \ undocumented{}

Capsule == add

degrees(u: UP R1): List Integer ==
   l: List Integer := []
   while u ^= 0 repeat
      l := concat(degree u,l)
      u := reductum u
      l
   reduce(u: UP R1) ==
      g := "gcd"/[d for d in degrees u]
u := divideExponents(u, g:PI)::(UP R1)
[u, g:PI]

import Fraction Integer

rootOfUnity(j:I,n:I):RE ==
j = 0 => 1
arg:RE := 2*j*pi()/(n::RE)
cos arg + (-1)**(1/2) * sin arg

expand(s, g) ==
g = 1 => [s]
[rootOfUnity(i,g)*s**(1/g) for i in 0..g-1]

package DTP DesingTreePackage

-- DesingTreePackage.input --

)set break resume
)sys rm -f DesingTreePackage.output
)spool DesingTreePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DesingTreePackage
--R
--R DesingTreePackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: ... DesingTreeCategory(InfClsPoint),BLMET: BlowUpMethodCategory) is a package constructor
--R Abbreviation for DesingTreePackage is DTP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DTP
DesingTreePackage examples

The following is all the categories, domains and package used for the desingularisation by means of monoidal transformation (Blowing-up)

See Also:
  o )show DesingTreePackage
DesingTreePackage (DTP)

Exports:

adjunctionDivisor  blowUp  blowUpWithExcpDiv
desingTree  desingTreeAtPoint  divisorAtDesingTree
tfullParamInit  genus  genusNeg
genusTree  genusTreeNeg  inBetweenExcpDiv
initParLocLeaves  initializeParamOfPlaces

— package DTP DesingTreePackage —

)abbrev package DTP DesingTreePackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: 31 jan 95
++ Description:
++ The following is all the categories, domains and package
++ used for the desingularisation be means of
++ monoidal transformation (Blowing-up)
DesingTreePackage(K,
symb,
PolyRing,
E,
ProjPt,
PCS,
Plc,
DIVISOR,
InfClsPoint,
DesTree,
BLMET
):Exports == Implementation where
K:Field
symb: List(Symbol)

OV ==> OrderedVariableList(symb)

E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)

bls ==> ['X,'Y]
BlUpRing ==> DistributedMultivariatePolynomial( bls , K)
E2 ==> DirectProduct( #bls , NonNegativeInteger )
AFP ==> AffinePlane(K)
OV2 ==> OrderedVariableList( bls )
PI ==> PositiveInteger
INT ==> Integer
NNI ==> NonNegativeInteger
LPARSPT ==> LocalParametrizationOfSimplePointPackage
PARAMP ==> ParametrizationPackage
PRALGPK ==> ProjectiveAlgebraicSetPackage

InfClsPoint : InfinitelyClosePointCategory(K,symb,PolyRing,E,ProjPt,-
            PCS,Plc,DIVISOR,BLMET)
DesTree : DesingTreeCategory(InfClsPoint)
BLMET : BlowUpMethodCategory

PackPoly => PackageForPoly(K,PolyRing,E,#symb)
PACKBL => PackageForPoly( K , BlUpRing , E2 , #bls )
NP => NewtonPolygon(K,BlUpRing,E2,#bls)
PPFC1 => PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
PPFC2 => BlowUpPackage(K,symb,PolyRing,E , BLMET)
ParamPackFC => LPARSPT(K,symb,PolyRing,E,ProjPt,PCS,Plc)
ParamPack => PARAMP(K,symb,PolyRing,E,ProjPt,PCS,Plc)
PrjAlgPack => PRALGPK(K,symb,PolyRing,E,ProjPt)

Exports =>
    with

        blowUp: InfClsPoint -> List InfClsPoint

        divisorAtDesingTree: (PolyRing,DesTree) -> DIVISOR
          ++ divisorAtDesingTree(f,tr) computes the local
          ++ divisor of f at a desingularisation tree tr of
          ++ a singular point.

        adjunctionDivisor: DesTree -> DIVISOR
          ++ adjunctionDivisor(tr) compute the local
          ++ adjunction divisor of a desingularisation tree tr of
          ++ a singular point.

        blowUpWithExcpDiv: DesTree -> Void -- DesTree

        desingTreeAtPoint: (ProjPt,PolyRing) -> DesTree
desingTree: PolyRing -> List DesTree
  ++ desingTree(pol) returns all the desingularisation trees of all singular points on the curve defined by pol.

fullParamInit: DesTree -> Void
  ++ fullParamInit(tr) initialize the local parametrization at all places (leaves of tr), computes the local exceptional divisor at each infinitely close points in the tree.
  ++ This function is equivalent to the following called:
  ++ initParLocLeaves(tr)
  ++ initializeParamOfPlaces(tr)
  ++ blowUpWithExcDiv(tr)

initParLocLeaves: DesTree -> Void
  ++ initParLocLeaves(tr) initialize the local parametrization at simple points corresponding to the leaves of tr.

initializeParamOfPlaces: DesTree -> Void
  ++ initializeParamOfPlaces(tr) initialize the local parametrization at places corresponding to the leaves of tr.

initializeParamOfPlaces: (DesTree, List PolyRing) -> Void
  ++ initializeParamOfPlaces(tr, listOfFnc) initialize
  ++ the local parametrization at places corresponding to the leaves of tr according to the given list of functions in listOfFnc.

genus: PolyRing -> NNI
  ++ genus(pol) computes the genus of the curve defined by pol.

genusNeg: PolyRing -> INT
  ++ genusNeg(pol) computes the "genus" of a curve that may be not absolutely irreducible.
  ++ A "negative" genus means that the curve is reducible !!.

genusTree: (NNI, List(DesTree)) -> NNI
  ++ genusTree(n, listOfTrees) computes the genus of a curve, where n is the degree of a polynomial pol defining the curve and listOfTrees is all the desingularisation trees at all singular points.
++ on the curve defined by pol.

inBetweenExcpDiv: DesTree -> DIVISOR

genusTreeNeg: (NNI,List(DesTree)) -> INT
++ genusTreeNeg(n,listOfTrees) computes the "genus"
++ of a curve that may be not absolutely irreducible,
++ where n is the degree of a polynomial pol
++ defining the curve and listOfTrees is all the
++ desingularisation trees at all singular points
++ on the curve defined by pol.
++ A "negative" genus means that
++ the curve is reducible !!.

Implementation ==> add
import PackPoly
import PPFC1
import PPFC2
import PolyRing
import DesTree

divisorAtDesingTreeLocal: (BlUpRing , DesTree ) -> DIVISOR

polyRingToBlUpRing: (PolyRing, BLMET) -> BlUpRing

makeMono: DesTree -> BlUpRing

inBetweenExcpDiv( tr )==
  -- trouve le diviseur excp. d'un pt inf voisin PRECEDENT !
  -- qV est egal a : 1 + nombre de fois que ce point est repete
  -- dans un chaine (le plus un correspond au point d'origine du
  -- point dont il est question ici.
  -- mp est la multiplicitie du point.
  -- cette fonction n'est et ne peut etre qu'utiliser pour
  -- calculer le diviseur d'adjonction ( a cause du mp -1).
  noeud:= value tr
  chart:= chartV noeud
  qV:= quotValuation chart
  one? qV => 0$DIVISOR
  expDiv := divisorAtDesingTreeLocal(makeMono(tr),tr)
  mp:= degree expDiv
  ((qV - 1) * (mp -1)) *$DIVISOR expDiv

polyRingToBlUpRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  l1:= [ d.i for i in 1..3 | ~( i = chartCoord(chart) ) ]
  e:= directProduct( vector( l1)$Vector(NNI) )$E2
  monomial(lc , e )$BlUpRing + polyRingToBlUpRing( reductum pol, chart )
affToProj(pt:AFP, chart:BLMET):ProjPt ==
  nV := chartCoord chart
  d:List(K) := list(pt)$AFP
  ll:List K :=
    nV = 1 => [ 1$K , d.1 , d.2 ]
    nV = 2 => [ d.1 , 01$K , d.2 ]
    [d.1 , d.2 , 1 ]
  projectivePoint( ll )$ProjPt

biringToPolyRing: (BlUpRing, BLMET) -> PolyRing

biringToPolyRing(pol,chart)==
  zero? pol => 0
  lc:= leadingCoefficient pol
  d:=entries degree pol
  nV:= chartCoord chart
  ll:List NNI :=
    nV = 1 => [ 0$NNI , d.1 , d.2 ]
    nV = 2 => [ d.1 , 0$NNI , d.2 ]
    [d.1 , d.2 , 0$NNI ]
  e:= directProduct( vector( ll)$Vector(NNI) )$E
  monomial(lc , e )$PolyRing + biringToPolyRing( reductum pol, chart )

minus : (NNI,NNI) -> NNI

minus(a,b)==
  d:=subtractIfCan(a,b)
  d case "failed" => error "cannot substract a-b if b>a for NNI"
  d

-- returns the exceptional coordinate function

makeExcpDiv: List DesTree -> DIVISOR

desingTreeAtPointLocal: InfClsPoint -> DesTree

subGenus: DesTree -> NNI

lVar:List PolyRing := _
  [monomial(1,index(i pretend PI)$OV,1)$PolyRing for i in 1..#symb]

divisorAtDesingTreeLocal(pol,tr)==
  -- BLMET has QuadraticTransform ; marche aussi avec
  -- Hamburger-Noether mais surement moins efficace
  noeud:=value(tr)
  pt:=localPointV(noeud)
  chart:= chartV noeud
  -- ram:= ramiMult chart -- ????
  -- new way to compute in order not to translate twice pol
polTrans:=translate(pol,list(pt)$AFP)
multPol:=degreeOfMinimalForm(polTrans)
chtr:=children(tr)
parPol:=parametrize(biringToPolyRing(pol,chartV(noeud))_,localParamV(noeud))$ParamPack
ord:=order(parPol)$PCS
ord * excpDivV(noeud) -- Note: le div excp est une fois la place.
(multPol *$DIVISOR excpDivV(noeud)) +$DIVISOR _
reduce("+",[divisorAtDesingTreeLocal(_
    quadTransform(polTrans,multPol,(chartV(value(child))_),
    child)_
    for child in chtr])

desingTreeAtPointLocal(ipt) ==
-- crb:PolyRing,pt:ProjPt,lstnV:List(INT),origPoint:ProjPt,actL:K)==
-- peut etre est-il preferable, avant d'eclater, de tester
-- si le point est simple avec les derives, et non
-- verifier si le point est simple ou non apres translation.
-- ????
blbl:=blowUp ipt
multPt:=multV ipt
one?(multPt) =>
    tree( ipt )$DesTree
subTree:=List DesTree:= [desingTreeAtPointLocal( iipt ) for iipt in blbl]
tree( ipt, subTree )$DesTree

blowUp(ipt)==
crb:=curveV ipt
pt:= localPointV ipt
lstnV := chartV ipt -- CHH no modif needed
actL:= actualExtensionV ipt
origPoint:= pointV ipt
blbl:=stepBlowUp(crb,pt,lstnV,actL) -- CHH no modif needed
multPt:=blbl.mult
sm:= blbl.subMult
-- la multiplicite et la frontiere du polygone de Newton (ou la forme
-- minimale selon BLMET) du point ipt est assigne par effet de bord !
setmult!(ipt,multPt)
setsubmult!(ipt, sm)
one?(multPt) => empty()
[create(origPoint,_
    rec(recTransStr),_
    rec(recPoint),_
    0,_
    rec(recChart),_
    0,0$DIVISOR,_)
CHAPTER 5. CHAPTER D

```plaintext
rec(definingExtension),
new(I)$Symbol$InfClsPoint for rec in bbl1.blUpRec

makeMono(arb)==
  monomial(1,index(excepCoord(chartV(value(arb)))) pretend PI)$OV2, 1)$BlUpRing

makeExcpDiv(lstSsArb)==
  reduce("+", _
    [divisorAtDesingTreeLocal(makeMono(arb),arb) for arb in lstSsArb],0)

adjunctionDivisorForQuadTrans: DesTree -> DIVISOR
adjunctionDivisorForHamburgeNoether: DesTree -> DIVISOR

adjunctionDivisor( tr )==
  BLMET has QuadraticTransform => adjunctionDivisorForQuadTrans( tr )
  BLMET has HamburgerNoether => adjunctionDivisorForHamburgeNoether( tr )
  error _
    " The algorithm to compute the adjunction divisor is not defined for the blowing method you

adjunctionDivisorForHamburgeNoether( tr )==
  noeud:=value tr
  chtr:=children tr
  empty?(chtr) => 0$DIVISOR -- on suppose qu'un noeud sans feuille
  multPt:=multV(noeud)
  ( minus(multPt,1) pretend INT) *$DIVISOR excpDivV(noeud) +$DIVISOR _
    reduce("+",[inBetweenExcpDiv( arb ) for arb in chtr ])+$DIVISOR _
    reduce("+",[adjunctionDivisorForHamburgeNoether(arb) for arb in chtr])

adjunctionDivisorForQuadTrans(tr)==
  noeud:=value(tr)
  chtr:=children(tr)
  empty?(chtr) => 0$DIVISOR
  multPt:=multV(noeud)
  ( minus(multPt,1) pretend INT) *$DIVISOR excpDivV(noeud) +$DIVISOR _
    reduce("+",[adjunctionDivisorForQuadTrans(child) for child in chtr])

divisorAtDesingTree( pol , tr)==
  chart:= chartV value(tr)
  pp:= polyRingToBlUpRing( pol, chart )
  divisorAtDesingTreeLocal( pp, tr )

subGenus(tr)==
  noeud:=value tr
  mult:=multV(noeud)
  chart := chartV noeud
  empty?(chdr:=children(tr)) => 0 -- degree(noeud)* mult* minus(mult,1)
  degree(noeud)* ( mult*minus( mult, 1 ) + subMultV( noeud ) ) +
    reduce("+",[subGenus(ch) for ch in chdr])
```

initializeParamOfPlaces(tr, lpol) ==
  noeud := value(tr)
  pt := localPointV(noeud)
  crb := curveV(noeud)
  chart := chartV(noeud) -- CHH
  nV := chartCoord chart
  chtr := List DesTree := children(tr)
  plc := Plc
  lParam := List PCS
  dd := PositiveInteger := degree noeud
  lcoef := List K
  l := Integer
  lParInf := List (PCS)
  lpar := List PCS
  empty?(chtr) =>
    lPar := localParamOfSimplePt(affToProj(pt, chart),
      biringToPolyRing(crb, chart), nV)$ParamPackFC
    setlocalParam!(noeud, lPar)
    lParam := [parametrize(f, lPar)$ParamPack for f in lpol]
    plc := create(symbNameV(noeud))$Plc
    setParam!(plc, lParam)
    setDegree!(plc, dd)
    itsALeaf!(plc)
    setexcpDiv!(noeud, plc :: DIVISOR)
    void()
  lpolTrans := List PolyRing :=
    [translateToOrigin(pol, affToProj(pt, chart), nV) for pol in lpol]
  lpolBlUp := List PolyRing
  chartBl := BLMET
  for arb in chtr repeat
    chartBl := chartV value arb
    lpolBlUp := [applyTransform(pol, chartBl) for pol in lpolTrans]
    initializeParamOfPlaces(arb, lpolBlUp)
    void()

blowUpWithExcpDiv(tr : DesTree) ==
  noeud := value(tr)
  pt := localPointV(noeud)
  crb := curveV(noeud)
  chtr := List DesTree := children(tr)
  empty?(chtr) => void() -- tr
  for arb in chtr repeat
    blowUpWithExcpDiv(arb)
    setexcpDiv!(noeud, makeExcpDiv(chtr))
    void()

fullParamInit(tr) ==
  initializeParamOfPlaces(tr)
  blowUpWithExcpDiv(tr)
CHAPTER 5. CHAPTER D

void()

initializeParamOfPlaces(tr)==initializeParamOfPlaces(tr,lVar)

desingTreeAtPoint(pt,crb)==
ipt:= create(pt,crb)$InfClsPoint
desingTreeAtPointLocal ipt

genus(crb)==
    if BLMET has HamburgerNoether then _
        print("BUG BUG corige le bug GH ---- ")::OutputForm
    degCrb:=totalDegree(crb)$PackPoly
genusTree(degCrb,desingTree(crb))

genusNeg(crb)==
decrb:=totalDegree(crb)$PackPoly
genusTreeNeg(decrb,desingTree(crb))

desingTree(crb)==
    [desingTreeAtPoint(pt,crb) for pt in singularPoints(crb)$PrjAlgPack]

genusTree(decrb,listArbDes)==
    -- le test suivant est necessaire
    -- ( meme s'il n'y a pas de point singulier dans ce cas)
    -- car avec sousNNI on ne peut retourner un entier negatif
    (decrb <$NNI 3::NNI) and ^empty?(listArbDes) =>
        print("Too many infinitely near points")::OutputForm
        print("The curve may not be absolutely irreducible")::OutputForm
        error "Have a nice day"
    (decrb <$NNI 3::NNI) => 0
    ga:= ( minus(decrb,1)*minus(decrb,2) ) quo$NNI 2
    empty?(listArbDes) => ga
    --calcul du nombre de double point
    dp:= reduce("+",[subGenus(arbD) for arbD in listArbDes]) quo$NNI 2
    (dp >$NNI ga) =>
        print("Too many infinitely near points")::OutputForm
        print("The curve may not be absolutely irreducible")::OutputForm
        error "Have a nice day"
    minus(ga,dp)

genusTreeNeg(decrb,listArbDes)==
    -- (decrb <$NNI 3::NNI) => 0
    ga:= (decrb-1)*(decrb-2) quo$INT 2
    empty?(listArbDes) => ga
    ga- (~reduce("+",[subGenus(arbD) for arbD in listArbDes]) quo$NNI 2)::INT

    |||
package DIOSP DiophantineSolutionPackage

--- DiophantineSolutionPackage.input ---

)set break resume
)sys rm -f DiophantineSolutionPackage.output
)spool DiophantineSolutionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DiophantineSolutionPackage
--R
--R DiophantineSolutionPackage is a package constructor
--R Abbreviation for DiophantineSolutionPackage is DIOSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DIOSP
--R
--R----------------------------------- Operations -----------------------------------
--R dioSolve : Equation(Polynomial(Integer)) -> Record(varOrder: List(Symbol),inhom: Union(List(Vector(NonNegativeInteger)),"failed"),hom: List(Vector(NonNegativeInteger))
--R
--E 1

)spool
)lisp (bye)

---

--- DiophantineSolutionPackage.help ---

DiophantineSolutionPackage examples

Any solution of a homogeneous linear Diophantine equation can be represented as a sum of minimal solutions, which form a "basis" (a
minimal solution cannot be represented as a nontrivial sum of solutions) in the case of an inhomogeneous linear Diophantine equation, each solution is the sum of a inhomogeneous solution and any number of homogeneous solutions therefore, it suffices to compute two sets:

1. all minimal inhomogeneous solutions
2. all minimal homogeneous solutions

the algorithm implemented is a completion procedure, which enumerates all solutions in a recursive depth-first-search it can be seen as finding monotone paths in a graph for more details see Reference

See Also:
- )show DiophantineSolutionPackage

---

DiophantineSolutionPackage (DIOSP)

Exports:
dioSolve

--- package DIOSP DiophantineSolutionPackage ---

)abbrev package DIOSP DiophantineSolutionPackage
++ Author: A. Fortenbacher
++ Date Created: 29 March 1991
++ Date Last Updated: 29 March 1991
++ Reference:
++ M. Clausen, A. Fortenbacher: Efficient Solution of
++ Linear Diophantine Equations. in JSC (1989) 8, 201-216
++ Description:
++ Any solution of a homogeneous linear Diophantine equation
++ can be represented as a sum of minimal solutions, which
++ form a "basis" (a minimal solution cannot be represented
++ as a nontrivial sum of solutions)
++ in the case of an inhomogeneous linear Diophantine equation,
++ each solution is the sum of a inhomogeneous solution and
++ any number of homogeneous solutions
++ therefore, it suffices to compute two sets:
++ 1. all minimal inhomogeneous solutions
++ 2. all minimal homogeneous solutions
++ the algorithm implemented is a completion procedure, which
++ enumerates all solutions in a recursive depth-first-search
++ it can be seen as finding monotone paths in a graph
++ for more details see Reference

DiophantineSolutionPackage(): Cat == Capsule where

B ==> Boolean
I ==> Integer
NI ==> NonNegativeInteger

LI ==> List(I)
VI ==> Vector(I)
VNI ==> Vector(NI)

POLI ==> Polynomial(I)
EPOLI ==> Equation(POLI)
LPOLI ==> List(POLI)

S ==> Symbol
LS ==> List(S)

ListSol ==> List(VNI)
Solutions ==> Record(varOrder: LS, inhom: Union(ListSol,"failed"),
                   hom: ListSol)

Node ==> Record(vert: VI, free: B)
Graph ==> Record(vn: Vector(Node), dim : NI, zeroNode: I)

Cat ==>

dioSolve: EPOLI -> Solutions
  ++ dioSolve(u) computes a basis of all minimal solutions for
  ++ linear homogeneous Diophantine equation u,
  ++ then all minimal solutions of inhomogeneous equation

Capsule ==>

import I
import POLI
-- local function specifications

initializeGraph: (LPOLI, I) -> Graph
createNode: (I, VI, NI, I) -> Node
findSolutions: (VNI, I, I, I, Graph, B) -> ListSol
verifyMinimality: (VNI, Graph, B) -> B
verifySolution: (VNI, I, I, I, Graph) -> B

-- exported functions

dioSolve(eq) ==
  p := lhs(eq) - rhs(eq)
  n := totalDegree(p)
  n = 0 or n > 1 =>
    error "a linear Diophantine equation is expected"
  mon := empty()$LPOLI
  c : I := 0
  for x in monomials(p) repeat
    ground?(x) =>
      c := ground(x) :: I
      mon := cons(x, mon)$LPOLI
  graph := initializeGraph(mon, c)
  sol := zero(graph.dim)$VNI
  hs := findSolutions(sol, graph.zeroNode, 1, 1, graph, true)
  ihs : ListSol :=
    c = 0 => [sol]
    findSolutions(sol, graph.zeroNode + c, 1, 1, graph, false)
  vars := [first(variables(x))$LS for x in mon]
  [vars, if empty?(ihs)$ListSol then "failed" else ihs, hs]

-- local functions

initializeGraph(mon, c) ==
  coeffs := vector([first(coefficients(x))$LI for x in mon])$VI
  k := #coeffs
  m := min(c, reduce(min, coeffs)$VI)
  n := max(c, reduce(max, coeffs)$VI)
  [[createNode(i, coeffs, k, 1 - m) for i in m..n], k, 1 - m]

createNode(ind, coeffs, k, zeroNode) ==
  -- create vertices from node ind to other nodes
  v := zero(k)$VI
  for i in 1..k repeat
    ind > 0 =>
      coeffs.i < 0 =>
        v.i := zeroNode + ind + coeffs.i
      coeffs.i > 0 =>
        v.i := zeroNode + ind + coeffs.i
    [v, true]
findSolutions(sol, ind, m, n, graph, flag) ==
-- return all solutions (paths) from node ind to node zeroNode
sols := empty()$ListSol
node := graph.vn.ind
node.free =>
  node.free := false
v := node.vert
k := if ind < graph.zeroNode then m else n
for i in k..graph.dim repeat
  x := sol.i
  v.i > 0 => -- vertex exists to other node
    sol.i := x + 1
  v.i = graph.zeroNode => -- solution found
    verifyMinimality(sol, graph, flag) =>
      sols := cons(copy(sol)$VNI, sols)$ListSol
      sol.i := x
      sol.i := x
      s :=
      ind < graph.zeroNode =>
        findSolutions(sol, v.i, i, n, graph, flag)
        findSolutions(sol, v.i, m, i, graph, flag)
      sols := append(s, sols)$ListSol
      sol.i := x
    node.free := true
  sols
verifyMinimality(sol, graph, flag) ==
-- test whether sol contains a minimal homogeneous solution
flag => -- sol is a homogeneous solution
  i := 1
  while sol.i = 0 repeat
    i := i + 1
    x := sol.i
    sol.i := (x - 1) :: NI
    flag := verifySolution(sol, graph.zeroNode, 1, 1, graph)
    sol.i := x
    flag
  verifySolution(sol, graph.zeroNode, 1, 1, graph)
verifySolution(sol, ind, m, n, graph) ==
-- test whether sol contains a path from ind to zeroNode
flag := true
node := graph.vn.ind
v := node.vert
k := if ind < graph.zeroNode then m else n
for i in k..graph.dim while flag repeat
  x := sol.i
  x > 0 and v.i > 0 => -- vertex exists to other node
sol.i := (x - 1) :: NI
v.i = graph.zeroNode => -- solution found
   flag := false
   sol.i := x
   flag :=
   ind < graph.zeroNode =>
      verifySolution(sol, v.i, i, n, graph)
      verifySolution(sol, v.i, m, i, graph)
      sol.i := x
   flag

---

package DIRPROD2 DirectProductFunctions2

--- DirectProductFunctions2.input ---

)set break resume
)sys rm -f DirectProductFunctions2.output
)spool DirectProductFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DirectProductFunctions2
--R
--R DirectProductFunctions2(dim: NonNegativeInteger,A: Type,B: Type) is a package constructor
--R Abbreviation for DirectProductFunctions2 is DIRPROD2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DIRPROD2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((A -> B),DirectProduct(dim,A)) -> DirectProduct(dim,B)
--R reduce : (((A,B) -> B),DirectProduct(dim,A),B) -> B
--R scan : (((A,B) -> B),DirectProduct(dim,A),B) -> DirectProduct(dim,B)
This package provides operations which all take as arguments direct products of elements of some type A and functions from A to another type B. The operations all iterate over their vector argument and either return a value of type B or a direct product over B.

See Also:
o )show DirectProductFunctions2
++ Description:
++ This package provides operations which all take as arguments direct
++ products of elements of some type \texttt{spad\{A\}} and functions from \texttt{spad\{A\}}
++ to another type B. The operations all iterate over their vector argument
++ and either return a value of type B or a direct product over B.

DirectProductFunctions2(dim, \texttt{A}, \texttt{B}): Exports == Implementation where
\begin{verbatim}
   dim : NonNegativeInteger
   \texttt{A}, \texttt{B}: Type

   \texttt{DA} ==> DirectProduct(dim, \texttt{A})
   \texttt{DB} ==> DirectProduct(dim, \texttt{B})
   \texttt{VA} ==> Vector \texttt{A}
   \texttt{VB} ==> Vector \texttt{B}
   \texttt{O2} ==> FiniteLinearAggregateFunctions2(\texttt{A}, \texttt{VA}, \texttt{B}, \texttt{VB})
\end{verbatim}

Exports ==>
\begin{verbatim}
   with
   \texttt{scan} : ((\texttt{A}, \texttt{B}) -> \texttt{B}, \texttt{DA}, \texttt{B}) -> \texttt{DB}
      \begin{verbatim}
         \texttt{scan}(func,vec,ident) creates a new vector whose elements are
         \texttt{the result of applying \texttt{reduce} to the binary function \texttt{func},}
         \texttt{increasing initial subsequences of the vector \texttt{vec},}
         \texttt{and the element \texttt{ident}.}
      \end{verbatim}
   \texttt{reduce} : ((\texttt{A}, \texttt{B}) -> \texttt{B}, \texttt{DA}, \texttt{B}) -> \texttt{B}
      \begin{verbatim}
         \texttt{reduce(func,vec,ident) combines the elements in \texttt{vec} using the}
         \texttt{binary function \texttt{func}. Argument \texttt{ident} is returned if the vector is empty.}
      \end{verbatim}
   \texttt{map} : (\texttt{A} -> \texttt{B}, \texttt{DA}) -> \texttt{DB}
      \begin{verbatim}
         \texttt{map(f, v) applies the function \texttt{f} to every element of the vector \texttt{v}}
         \texttt{producing a new vector containing the values.}
      \end{verbatim}
\end{verbatim}

Implementation ==>
\begin{verbatim}
   import FiniteLinearAggregateFunctions2(\texttt{A}, \texttt{VA}, \texttt{B}, \texttt{VB})

   \texttt{map(f, v)} == \texttt{directProduct map(f, v::VA)}
   \texttt{scan(f, v, b)} == \texttt{directProduct scan(f, v::VA, b)}
   \texttt{reduce(f, v, b)} == \texttt{reduce(f, v::VA, b)}
\end{verbatim}
package DLP DiscreteLogarithmPackage

— DiscreteLogarithmPackage.input —

)set break resume
)sys rm -f DiscreteLogarithmPackage.output
)spool DiscreteLogarithmPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DiscreteLogarithmPackage
--R
--R DiscreteLogarithmPackage(M) where
--R M: Join(Monoid,Finite)with
--R **? : (M,Integer) -> M is a package constructor
--R Abbreviation for DiscreteLogarithmPackage is DLP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DLP
--R
--R----------------------------- Operations -----------------------------
--R shanksDiscLogAlgorithm : (M,M,NonNegativeInteger) -> Union(NonNegativeInteger,"failed")
--R
--E 1

)spool
)lisp (bye)

———

— DiscreteLogarithmPackage.help —

DiscreteLogarithmPackage examples

DiscreteLogarithmPackage implements help functions for discrete logarithms in monoids using small cyclic groups.

See Also:
  o )show DiscreteLogarithmPackage

———
DiscreteLogarithmPackage (DLP)

Exports:
shanksDiscLogAlgorithm

— package DLP DiscreteLogarithmPackage —

)abbrev package DLP DiscreteLogarithmPackage
++ Author: J. Grabmeier, A. Scheerhorn
++ Date Created: 12 March 1991
++ Date Last Updated: 31 March 1991
++ References:
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ DiscreteLogarithmPackage implements help functions for discrete logarithms
++ in monoids using small cyclic groups.

DiscreteLogarithmPackage(M): public == private where
M : Join(Monoid,Finite) with
"**": (M,Integer) -> M
  ++ x ** n returns x raised to the integer power n
public ==> with
  shanksDiscLogAlgorithm:(M,M,NonNegativeInteger)-> Union(NonNegativeInteger, "failed")
  ++ shanksDiscLogAlgorithm(b,a,p) computes s with \spad{b**s = a} for
  ++ assuming that \spad{a} and b are elements in a 'small' cyclic group of
  ++ order p by Shank's algorithm.
  ++ Note that this is a subroutine of the function \spadfun{discreteLog}.
I ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
SUP ==> SparseUnivariatePolynomial
DLP ==> DiscreteLogarithmPackage

private ==> add
  shanksDiscLogAlgorithm(logbase,c,p) ==
limit:Integer:=30
-- for logarithms up to cyclic groups of order limit a full
-- logarithm table is computed
p < limit =>
a:M:=1
disclog:Integer:=0
found:Boolean:=false
for i in 0..p-1 while not found repeat
  a = c =>
    disclog:=i
    found:=true
    a:=a*logbase
  not found =>
    messagePrint("discreteLog: second argument not in cyclic group_
generated by first argument")$OutputForm
    "failed"
    disclog pretend NonNegativeInteger
l:Integer:=length(p)$Integer
if odd?(l)$Integer then n:Integer:= shift(p,-(l quo 2))
  else n:Integer:= shift(1,(l quo 2))
a:M:=1
exptable : Table(PI,NNI) :=table()$Table(PI,NNI)
for i in (0::NNI)..(n-1)::NNI repeat
  insert_!(lookup(a),i::NNI)$Record(key:PI,entry:NNI),
exptable)$Table(PI,NNI)
a:=a*logbase
found := false
end := (p-1) quo n
disclog:Integer:=0
a := c
b := logbase ** (-n)
for i in 0..end while not found repeat
  rho:= search(lookup(a),exptable)_
    $Table(PositiveInteger,NNI)
rho case NNI =>
    found := true
    disclog:= n * i + rho pretend Integer
    a := a * b
  not found =>
    messagePrint("discreteLog: second argument not in cyclic group_
generated by first argument")$OutputForm
    "failed"
    disclog pretend NonNegativeInteger

———
— DLP.dotabb —
package DISPLAY DisplayPackage

— DisplayPackage.input —

)set break resume
)sys rm -f DisplayPackage.output
)spool DisplayPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DisplayPackage
--R
--R DisplayPackage is a package constructor
--R Abbreviation for DisplayPackage is DISPLAY
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DISPLAY
--R
--R-------------------------------- Operations --------------------------------
--R bright : String -> List(String) copies : (Integer,String) -> String
--R newLine : () -> String say : String -> Void
--R say : List(String) -> Void sayLength : String -> Integer
--R sayLength : List(String) -> Integer
--R bright : List(String) -> List(String)
--R center : (String,Integer,String) -> String
--R center : (List(String),Integer,String) -> List(String)
--R
--E 1

)spool
)lisp (bye)

— DisplayPackage.help —

====================================================================

DisplayPackage examples
DisplayPackage allows one to print strings in a nice manner, including highlighting substrings.

See Also:
- )show DisplayPackage

Exports:
bright  center  copies  newLine  say  sayLength

--- package DISPLAY DisplayPackage ---

)abbrev package DISPLAY DisplayPackage
++ Author: Robert S. Sutor
++ Date Created: September 1986
++ Description:
++ DisplayPackage allows one to print strings in a nice manner,
++ including highlighting substrings.

DisplayPackage: public == private where
  I ==> Integer
  L ==> List
  S ==> String
  RECLR ==> Record(lhs : S, rhs : S)

public == with
  bright: S -> L S
  ++ bright(s) sets the font property of the string s to bold-face type.
CHAPTER 5. CHAPTER D

bright: (L S) -> L S
++ bright(l) sets the font property of a list of strings, l, to ++ bold-face type.
newLine: () -> S
++ newLine() sends a new line command to output.
copies: (I,S) -> S
++ copies(i,s) will take a string s and create a new string composed of ++ i copies of s.
center: (S,I,S) -> S
++ center(s,i,s) takes the first string s, and centers it within a string ++ of length i, in which the other elements of the string are composed ++ of as many replications as possible of the second indicated string, s ++ which must have a length greater than that of an empty string.
center: (L S,I,S) -> L S
++ center(l,i,s) takes a list of strings l, and centers them within a ++ list of strings which is i characters long, in which the remaining ++ spaces are filled with strings composed of as many repetitions as ++ possible of the last string parameter s.
say: S -> Void
++ say(s) sends a string s to output.
say: L S -> Void
++ say(l) sends a list of strings l to output.
sayLength: S -> I
++ sayLength(s) returns the length of a string s as an integer.
sayLength: L S -> I
++ sayLength(l) returns the length of a list of strings l as an integer.

private == add
--StringManipulations()

center0: (I,I,S) -> RECLR

s : S
l : L S
HION : S := " "
HIOFF : S := " "
NEWLINE : S := "%l"
bright s == [HION,s,HIOFF](L S)
bright l == cons(HION,append(l,list HIOFF))
newLine() == NEWLINE

copies(n : I, s : S) ==
n < 1 => ""
n = 1 => s
t : S := copies(n quo 2, s)
odd? n => concat [s,t,t]
concat [t,t]

center0(len : I, wid : I, fill : S) : RECLR ==
  (wid < 1) or (len >= wid) => ["",""]$RECLR
  m : I := (wid - len) quo 2
  t : S := copies(1 + (m quo (sayLength fill)),fill)
  [t(1..m),t(1..wid-len-m)]$RECLR

center(s, wid, fill) ==
  wid < 1 => ""
  len : I := sayLength s
  len = wid => s
  len > wid => s(1..wid)
  rec : RECLR := center0(len,wid,fill)
  concat [rec.lhs,s,rec.rhs]

center(l, wid, fill) ==
  wid < 1 => [""]$(L S)
  len : I := sayLength l
  len = wid => l
  len > wid => s(1..wid)
  rec : RECLR := center0(len,wid,fill)
  cons(rec.lhs,append(l,list rec.rhs))

say s ==
  sayBrightly$Lisp s
  void()$Void

say l ==
  sayBrightly$Lisp l
  void()$Void

sayLength s == #s

sayLength l ==
  sum : I := 0
  for s in l repeat
    s = HION => sum := sum + 1
    s = HIOFF => sum := sum + 1
    s = NEWLINE => sum
    sum := sum + sayLength s
  sum

| DISPLAY.dotabb |

"DISPLAY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DISPLAY"]
package DDFACT DistinctDegreeFactorize

  — DistinctDegreeFactorize.input —

)set break resume
)sys rm -f DistinctDegreeFactorize.output
)spool DistinctDegreeFactorize.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show DistinctDegreeFactorize
--R
--R DistinctDegreeFactorize(F: FiniteFieldCategory,FP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for DistinctDegreeFactorize is DDFACT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DDFACT
--R
--R------------------------------ Operations ------------------------------
--R factor : FP -> Factored(FP) irreducible? : FP -> Boolean
--R distdfact : (FP,Boolean) -> Record(cont: F,factors: List(Record(irr: FP,pow: Integer)))
--R exptMod : (FP,NonNegativeInteger,FP) -> FP
--R factorSquareFree : FP -> Factored(FP)
--R separateDegrees : FP -> List(Record(deg: NonNegativeInteger,prod: FP))
--R separateFactors : List(Record(deg: NonNegativeInteger,prod: FP)) -> List(FP)
--R trace2PowMod : (FP,NonNegativeInteger,FP) -> FP
--R tracePowMod : (FP,NonNegativeInteger,FP) -> FP
--R
--E 1

)spool
)lisp (bye)

— DistinctDegreeFactorize.help —

====================================================================
DistinctDegreeFactorize examples
Package for the factorization of a univariate polynomial with coefficients in a finite field. The algorithm used is the "distinct degree" algorithm of Cantor-Zassenhaus, modified to use trace instead of the norm and a table for computing Frobenius as suggested by Naudin and Quitte.

See Also:
o )show DistinctDegreeFactorize

DistinctDegreeFactorize (DDFACT)

Exports:
distdfact exptMod factor factorSquareFree irreducible?
separateDegrees separateFactors trace2PowMod tracePowMod

-- package DDFACT DistinctDegreeFactorize --

DistinctDegreeFactorize(F,FP): C == T
where
F : FiniteFieldCategory
FP : UnivariatePolynomialCategory(F)

fUnion ==> Union("nil", "sqfr", "irred", "prime")
FFE == Record(flg:fUnion, fctr:FP, xpnt:Integer)
NNI == NonNegativeInteger
Z == Integer
fact == Record(deg : NNI, prod : FP)
ParFact == Record(iri:FP, pow:Z)
FinalFact == Record(cont:F, factors:List(ParFact))

C == with
  factor : FP -> Factored FP
    ++ factor(p) produces the complete factorization of the polynomial p.
factorSquareFree : FP -> Factored FP
    ++ factorSquareFree(p) produces the complete factorization of the
    ++ square free polynomial p.
distdfact : (FP, Boolean) -> FinalFact
    ++ distdfact(p,sqfrflag) produces the complete factorization
    ++ of the polynomial p returning an internal data structure.
    ++ If argument sqfrflag is true, the polynomial is assumed square free.
separateDegrees : FP -> List fact
    ++ separateDegrees(p) splits the squarfree polynomial p into
    ++ factors each of which is a product of irreducibles of the
    ++ same degree.
separateFactors : List fact -> List FP
    ++ separateFactors(1fact) takes the list produced by separateDegrees
    ++ and produces the complete list of factors.
exptMod : (FP, NNI, FP) -> FP
    ++ exptMod(u,k,v) raises the polynomial u to the kth power
    ++ modulo the polynomial v.
trace2PowMod : (FP, NNI, FP) -> FP
    ++ trace2PowMod(u,k,v) produces the sum of u**(2**i) for i running
    ++ from 1 to k all computed modulo the polynomial v.
tracePowMod : (FP, NNI, FP) -> FP
    ++ tracePowMod(u,k,v) produces the sum of u**(q**i)
    ++ for i running and q= size F
irreducible? : FP -> Boolean
    ++ irreducible?(p) tests whether the polynomial p is irreducible.

T == add
  --declarations
  D:==ModMonic(F,FP)
  import UnivariatePolynomialSquareFree(F,FP)
  --local functions
  notSqFr : (FP, FP -> List(FP)) -> List(ParFact)
ddffact : FP -> List(FP)
ddffact1 : (FP, Boolean) -> List fact
ranpol : NNI -> FP

charF : Boolean := characteristic()$F = 2

--construct a random polynomial of random degree < d
ranpol(d:NNI):FP ==
   k1: NNI := 0
   while k1 = 0 repeat k1 := random d
   -- characteristic F = 2
   charF =>
   u := 0$FP
   for j in 1..k1 repeat u := u + monomial(random()$F,j)
   u
   u := monomial(1,k1)
   for j in 0..k1-1 repeat u := u + monomial(random()$F,j)
   u

notSqFr(m:FP,appl: FP->List(FP)):List(ParFact) ==
   factlist : List(ParFact) := empty()
   llf : List FFE
   fln : List(FP) := empty()
   if (lcm:=leadingCoefficient m)^=1 then m:=(inv lcm)*m
   llf := factorList(squareFree(m))
   for lf in llf repeat
      d1:= lf.xpnt
      pol := lf.fctr
      if (lcp:=leadingCoefficient pol)^=1 then pol := (inv lcp)*pol
      degree pol=1 => factlist:=cons([pol,d1]$ParFact,factlist)
      fln := appl(pol)
      factlist := append([[pf,d1]$ParFact for pf in fln],factlist)
   factlist

-- compute u**k mod v (requires call to setPoly of multiple of v)
-- characteristic not equal 2
exptMod(u:FP,k:NNI,v:FP):FP == (reduce(u)$D**k):FP rem v

-- compute u**k mod v (requires call to setPoly of multiple of v)
-- characteristic equal 2
trace2PowMod(u:FP,k:NNI,v:FP):FP ==
   uu:=u
   for i in 1..k repeat uu:=(u+uu*uu) rem v
   uu

-- compute u+u**q+.u**(q**k) mod v
-- (requires call to setPoly of multiple of v) where q=size< F
tracePowMod(u:FP,k:NNI,v:FP):FP ==
   u1 :D := reduce(u)$D
   uu : D := u1
   for i in 1..k repeat uu:=(u1+frobenius uu)
   (lift uu) rem v
-- compute \( u^{(1+q+\ldots+q^k)} \) \( \text{rem} \) \( v \) where \( q = \#F \)
-- (requires call to \( \text{setPoly} \) of multiple of \( v \))
-- frobenius map is used
\[
\text{normPowMod}(u: \mathbb{F}, k: \mathbb{N}, v: \mathbb{F}) := u^1 : D := \text{reduce}(u) \in D
\]
\[
\text{uu} : D := u^1
\]
\[
\text{for} \ i \ \text{in} \ 1..k \ \text{repeat} \ \text{uu} := (u^1 \ast \text{frobenius uu})
\]
\[
(\text{lift uu}) \ \text{rem} \ v
\]

-- find the factorization of \( m \) as product of factors each containing
-- terms of equal degree.
-- if \( \text{testirr} = \text{true} \) the function returns the first factor found
\[
\text{ddffact1}(m: \mathbb{F}, \text{testirr}: \mathbb{B}): \mathbb{L}(\text{fact}) :=
\]
\[
p := \text{size} \in \mathbb{F}
\]
\[
dg := \mathbb{N} : = 0
\]
\[
\text{ddfact} : \mathbb{L}(\text{fact}) := \text{empty}()
\]
-- evaluation of \( x^p \mod m \)
\[
k1 : \mathbb{N}
\]
\[
u := m
\]
\[
\text{du} := \text{degree} u
\]
\[
\text{setPoly} \ u
\]
\[
\text{mon} : \mathbb{F} := \text{monomial}(1,1)
\]
\[
v := \text{mon}
\]
\[
\text{for} \ k1 \ \text{in} \ 1.. \text{while} \ k1 \ \text{<=} \ (\text{du} \ \text{quo} \ 2) \ \text{repeat}
\]
\[
v := \text{lift frobenius reduce(v)} \in D
\]
\[
g := \gcd(v - \text{mon},u)
\]
\[
dg := \text{degree} g
\]
\[
g \neq 0 \Rightarrow \ "\text{next} \ k1"\]
\[
\text{if leadingCoefficient} g \neq 1 \ \text{then} \ g := (\text{inv leadingCoefficient} g) \ast g
\]
\[
\text{ddfact} := \text{cons}([k1,g]$\text{fact},\text{ddfact})
\]
\[
\text{testirr} \Rightarrow \text{return} \ \text{ddfact}
\]
\[
u := u \ \text{quo} \ g
\]
\[
\text{du} := \text{degree} u
\]
\[
\text{du} = 0 \Rightarrow \text{return} \ \text{ddfact}
\]
\[
\text{setPoly} \ u
\]
\[
\text{cons}([\text{du},u]$\text{fact},\text{ddfact})
\]

-- test irreducibility
\[
\text{irreducible?}(m: \mathbb{F}) : \mathbb{B} :=
\]
\[
mf : \text{fact} := \text{first} \ \text{ddffact1}(m, \text{true})
\]
\[
\text{degree} \ m = \text{mf} \ . \text{deg}
\]

-- export \text{ddffact1}
\[
\text{separateDegrees}(m: \mathbb{F}) : \mathbb{L}(\text{fact}) := \text{ddffact1}(m, \text{false})
\]

-- find the complete factorization of \( m \), using the result of \text{ddfact1}
\[
\text{separateFactors}(: \text{distf} : \mathbb{L} \text{fact}) : \mathbb{L} \mathbb{F} :=
\]
\[
\text{ddfact} := \text{distf}
\]
\[
n1 : \mathbb{I}
\]
\begin{verbatim}

package DDFACT DISTINCTDEGREEFACTORIZE

pl:=size(F)
if charF then n1:=length(pl)-1
newaux,aux,ris : List FP
ris := empty()
t,fprod : FP
for ffprod in ddfact repeat
  fprod := ffprod.prod
d := ffprod.deg
degree fprod = d => ris := cons(fprod,ris)
aux:=[fprod]
setPoly fprod
while ^(empty? aux) repeat
  t := ranpol(2*deg)
  if charF then t:=trace2PowMod(t,(n1*d-1)::NNI,fprod)
  else t:=exptMod(tracePowMod(t,(d-1)::NNI,fprod),
                  (pl quo 2)::NNI,fprod)-1$FP
  newaux:=empty()
  for u in aux repeat
    g := gcd(u,t)
dg:= degree g
dg=0 or dg = degree u => newaux:=cons(u,newaux)
v := u quo g
  if dg=d then ris := cons(inv(leadingCoefficient g)*g,ris)
    else newaux := cons(g,newaux)
  if degree v=d then ris := cons(inv(leadingCoefficient v)*v,ris)
    else newaux := cons(v,newaux)
aux:=newaux
ris

--distinct degree algorithm for monic ,square-free polynomial
ddfact(m:FP):List(FP)==
ddfact:=ddffact1(m,false)
empty? ddfact => [m]
separateFactors ddfact

--factorize a general polynomial with distinct degree algorithm
--if test=true no check is executed on square-free
distdfact(m:FP,test:Boolean):FinalFact ==
  factlist: List(ParFact):= empty()
  fln : List(FP) :=empty()
  --make m monic
  if (lcm := leadingCoefficient m) ^=1 then m := (inv lcm)*m
  --is x**d factor of m?
  if (d := minimumDegree m)>0 then
    m := (monicDivide (m,monomial(1,d))).quotient
    factlist := [[monomial(1,1),d]$ParFact]
  d:=degree m

\end{verbatim}
--is m constant?
d=0 => [lcm,factlist]$FinalFact

--is m linear?
d=1 => [lcm,cons([m,d]$ParFact,factlist)]$FinalFact

--m is square-free
test =>
fln := ddffact m
factlist := append([[pol,1]$ParFact for pol in fln],factlist)
[lcm,factlist]$FinalFact

--factorize the monic,square-free terms
factlist:= append(notSqFr(m,ddffact),factlist)
[lcm,factlist]$FinalFact

--factorize the polynomial m
factor(m:FP) ==
m = 0 => 0
flist := distdfact(m,false)
makeFR(flist.cont::FP,[["prime",u.irr,u.pow]$FFE
                        for u in flist.factors])

--factorize the square free polynomial m
factorSquareFree(m:FP) ==
m = 0 => 0
flist := distdfact(m,true)
makeFR(flist.cont::FP,[["prime",u.irr,u.pow]$FFE
                        for u in flist.factors])

package DFSFUN DoubleFloatSpecialFunctions

The special functions in this section are developed as special cases but can all be expressed in terms of generalized hypergeometoric functions $pFq$ or its generalization, the Meijer G

--- DoubleFloatSpecialFunctions.input ---

)set break resume
)sys rm -f DoubleFloatSpecialFunctions.output
)spool DoubleFloatSpecialFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
)show DoubleFloatSpecialFunctions
--R
--R DoubleFloatSpecialFunctions is a package constructor
--R Abbreviation for DoubleFloatSpecialFunctions is DFSFUN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DFSFUN
--R
--R-------------------------------------- Operations --------------------------------------
--R Gamma : DoubleFloat -> DoubleFloat airyAi : DoubleFloat -> DoubleFloat
--R airyBi : DoubleFloat -> DoubleFloat digamma : DoubleFloat -> DoubleFloat
--R fresnelC : Float -> Float fresnelS : Float -> Float
--R Beta : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R Beta : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R E1 : DoubleFloat -> OnePointCompletion(DoubleFloat)
--R Ei : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei1 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei2 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei3 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei4 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei5 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Ei6 : OnePointCompletion(DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R En : (Integer,DoubleFloat) -> OnePointCompletion(DoubleFloat)
--R Gamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R airyAi : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R airyBi : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R besselI : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselI : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselJ : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselJ : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselK : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselK : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R besselY : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R besselY : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R digamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
--R hypergeometric0F1 : (DoubleFloat,DoubleFloat) -> DoubleFloat
--R hypergeometric0F1 : (Complex(DoubleFloat),Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R logGamma : DoubleFloat -> DoubleFloat
--R logGamma : Complex(DoubleFloat) -> Complex(DoubleFloat)
356

CHAPTER 5. CHAPTER D

--R polygamma : (NonNegativeInteger,DoubleFloat) -> DoubleFloat
--R polygamma : (NonNegativeInteger,Complex(DoubleFloat)) -> Complex(DoubleFloat)
--R
--E 1
--S 2 of 5
pearceyC:=_
[ [0.00, 0.0000000], [0.25,
[1.00, 0.7217059], [1.25,
[2.00, 0.753302], [2.25,
[3.00, 0.561020], [3.25,
[4.00, 0.368193], [4.25,
[5.00, 0.328457], [5.25,
[6.00, 0.443274], [6.25,
[7.00, 0.590116], [7.25,
[8.00, 0.639301], [8.25,
[9.00, 0.560804], [9.25,
[10.00, 0.436964], [10.25,
[11.00, 0.380390], [11.25,
[12.00, 0.434555], [12.25,
[13.00, 0.542511], [13.25,
[14.00, 0.604721], [14.25,
[15.00, 0.569335], [15.25,
[16.00, 0.474310], [16.25,
[17.00, 0.407985], [17.25,
[18.00, 0.427837], [18.25,
[19.00, 0.511332], [19.25,
[20.00, 0.580389], [20.25,
[21.00, 0.573842], [21.25,
[22.00, 0.501167], [22.25,
[23.00, 0.430662], [23.25,
[24.00, 0.425635], [24.25,
[25.00, 0.487880], [25.25,
[26.00, 0.558626], [26.25,
[27.00, 0.573766], [27.25,
[28.00, 0.521695], [28.25,
[29.00, 0.451832], [29.25,
[30.00, 0.427908], [30.25,
[31.00, 0.470019], [31.25,
[32.00, 0.537944], [32.25,
[33.00, 0.569407], [33.25,
[34.00, 0.537026], [34.25,
[35.00, 0.472012], [35.25,
[36.00, 0.434212], [36.25,
[37.00, 0.457140], [37.25,
[38.00, 0.518359], [38.25,
[39.00, 0.561321], [39.25,
[40.00, 0.547503], [40.25,
[41.00, 0.490870], [41.25,
[42.00, 0.443897], [42.25,

0.3964561], [0.50,
0.762404], [1.50,
0.717446], [2.50,
0.504745], [3.50,
0.341021], [4.50,
0.346058], [5.50,
0.482966], [6.50,
0.614951], [7.50,
0.629969], [8.50,
0.529344], [9.50,
0.413053], [10.50,
0.384231], [11.50,
0.460384], [12.50,
0.565798], [13.50,
0.605048], [14.50,
0.547984], [15.50,
0.451659], [16.50,
0.404300], [17.50,
0.445331], [18.50,
0.533222], [19.50,
0.586847], [20.50,
0.559824], [21.50,
0.480207], [22.50,
0.421906], [23.50,
0.436444], [24.50,
0.507725], [25.50,
0.569272], [26.50,
0.565954], [27.50,
0.503146], [28.50,
0.439675], [29.50,
0.432913], [30.50,
0.487100], [31.50,
0.551266], [32.50,
0.567026], [33.50,
0.521566], [34.50,
0.457857], [35.50,
0.434156], [36.50,
0.470848], [37.50,
0.533031], [38.50,
0.563619], [39.50,
0.535653], [40.50,
0.475980], [41.50,
0.439565], [42.50,

0.5502472], [0.75,
0.779084], [1.75,
0.670986], [2.75,
0.452047], [3.75,
0.325249], [4.75,
0.372439], [5.75,
0.522202], [6.75,
0.631845], [7.75,
0.612868], [8.75,
0.496895], [9.75,
0.395087], [10.75,
0.395149], [11.75,
0.488146], [12.75,
0.584583], [13.75,
0.598871], [14.75,
0.524009], [15.75,
0.432343], [16.75,
0.406589], [17.75,
0.465972], [18.75,
0.552774], [19.75,
0.587849], [20.75,
0.542266], [21.75,
0.460707], [22.75,
0.418080], [23.75,
0.451078], [24.75,
0.526896], [25.75,
0.575524], [26.75,
0.554127], [27.75,
0.484566], [28.75,
0.431359], [29.75,
0.442034], [30.75,
0.504844], [31.75,
0.561307], [32.75,
0.560508], [33.75,
0.504881], [34.75,
0.446415], [35.75,
0.438182], [36.75,
0.486272], [37.75,
0.545560], [38.75,
0.561957], [39.75,
0.521665], [40.75,
0.462670], [41.75,
0.439006], [42.75,

0.6531193],_
0.774978],_
0.617615],_
0.405762],_
0.321186],_
0.405610],_
0.558620],_
0.640034],_
0.589271],_
0.465469],_
0.384027],_
0.412319],_
0.516096],_
0.597795],_
0.586682],_
0.498930],_
0.417502],_
0.414627],_
0.488443],_
0.568812],_
0.583401],_
0.522293],_
0.443854],_
0.419367],_
0.468594],_
0.544215],_
0.577038],_
0.539054],_
0.467104],_
0.427366],_
0.454673],_
0.522148],_
0.567471],_
0.550288],_
0.488015],_
0.438375],_
0.446014],_
0.502444],_
0.555182],_
0.556463],_
0.506420],_
0.451755],_
0.442234],_


[43.00, 0.449025], [43.25, 0.458938], [43.50, 0.471341], [43.75, 0.485450], ...
[44.00, 0.500382], [44.25, 0.515205], [44.50, 0.534676], [44.75, 0.551883], ...
[45.00, 0.550239], [45.25, 0.566387], [45.50, 0.599004], [45.75, 0.557947], ...
[46.00, 0.553301], [46.25, 0.545246], [46.50, 0.534676], [46.75, 0.521883], ...
[47.00, 0.507802], [47.25, 0.493312], [47.50, 0.479313], [47.75, 0.466670], ...
[48.00, 0.456160], [48.25, 0.448425], [48.50, 0.443930], [48.75, 0.442946], ...
[49.00, 0.445486], [49.25, 0.451406], [49.50, 0.460311], [49.75, 0.471633], ...
[50.00, 0.484658]
[(x.1, x.2, fresnelC(x.1), fresnelC(x.1) - x.2) for x in pearceyC]
PACKAGE DFSFUN DOUBLEFLOATSPECIALFUNCTIONS
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R

[7.5,0.631845,0.6318452111 5510492853,0.2111551049 285 E -6],
[7.75,0.640034,0.6400345450 8057441808,0.5450805744 1808 E -6],
[8.0,0.639301,0.6393012479 3060490759,0.2479306049 076 E -6],
[8.25,0.629969,0.6299689859 2595953795,- 0.1407404046 2 E -7],
[8.5,0.612868,0.6128678201 6845088171,- 0.1798315491 183 E -6],
[8.75,0.589271,0.5892704028 202327594,- 0.5971797672 406 E -6],
[9.0,0.560804,0.5608039810 6395486433,- 0.1893604513 57 E -7],
[9.25,0.529344,0.5293438831 4394301245,- 0.1168560569 876 E -6],
[9.5,0.496895,0.4968951155 6828252077,0.1155682825 208 E -6],
[9.75,0.465469,0.4654692556 4195264614,0.2556419526 4614 E -6],
[10.0,0.436964,0.4369639527 2938203483,- 0.4727061796 517 E -7],
[10.25,0.413053,0.4130520539 2945147154,- 0.9460705485 2846 E -6],
[10.5,0.395087,0.3950866689 6445290526,- 0.3310355470 9474 E -6],
[10.75,0.384027,0.3840274319 9186745464,0.4319918674 5464 E -6],
[11.0,0.38039,0.3803918718 5818433242,0.0000018718 581843324],
[11.25,0.384231,0.3842342501 3415159269,0.0000032501 3415159269],
[11.5,0.395149,0.3951525621 4136633426,0.0000035621 4136633426],
[11.75,0.412319,0.4123227194 2948890487,0.0000037194 2948890487],
[12.0,0.434555,0.4345573415 1310106383,0.0000023415 1310106383],
[12.25,0.460384,0.4603851724 4692111457,0.0000011724 469211146],
[12.5,0.488146,0.4881459845 7100939501,- 0.1542899060 5 E -7],
[12.75,0.516096,0.5160950016 9800402129,- 0.9983019959 7871 E -6],
[13.0,0.542511,0.5425104114 0076790311,- 0.5885992320 9689 E -6],
[13.25,0.565798,0.5657974628 3445804807,- 0.5371655419 5193 E -6],
[13.5,0.584583,0.5845829612 9626646639,- 0.3870373353 36 E -7],
[13.75,0.597795,0.5977946491 2254037734,- 0.3508774596 227 E -6],
[14.0,0.604721,0.6047209589 3428343112,- 0.4106571656 89 E -7],
[14.25,0.605048,0.6050478757 7470272898,- 0.1242252972 71 E -6],
[14.5,0.598871,0.5988710711 7868251227,0.7117868251 227 E -7],
[14.75,0.586682,0.5866829870 7071159647,0.9870707115 9647 E -6],
[15.0,0.569335,0.5693360588 8342021462,0.0000010588 834202146],
[15.25,0.547984,0.5479846850 9637199303,0.6850963719 9303 E -6],
[15.5,0.524009,0.5240097909 4969920392,0.7909496992 0392 E -6],
[15.75,0.49893,0.4989308254 9359679937,0.8254935967 9937 E -6],
[16.0,0.47431,0.4743107173 2032792592,0.7173203279 2592 E -6],
[16.25,0.451659,0.4516596582 0625475374,0.6582062547 5374 E -6],
[16.5,0.432343,0.4323435693 667817725,0.5693667817 725 E -6],
[16.75,0.417502,0.4175027376 9772555286,0.7376977255 5286 E -6],
[17.0,0.407985,0.4079854159 5598154173,0.4159559815 4173 E -6],
[17.25,0.4043,0.4043002072 9975704736,0.2072997570 474 E -6],
[17.5,0.406589,0.4065898632 2726313915,0.8632272631 3915 E -6],
[17.75,0.414627,0.4146277893 4285610292,0.7893428561 0292 E -6],
[18.0,0.427837,0.4278371578 9257267748,0.1578925726 775 E -6],
[18.25,0.445331,0.4453311546 869756159,0.1546869756 159 E -6],
[18.5,0.465972,0.4659716234 4840774835,- 0.3765515922 5165 E -6],
[18.75,0.488443,0.4884422879 5458921791,- 0.7120454107 8209 E -6],
[19.0,0.511332,0.5113318949 159239085,- 0.1050840760 915 E -6],
[19.25,0.533222,0.5332220760 9031239166,0.7609031239 166 E -7],
[19.5,0.552774,0.5527745062 1484209042,0.5062148420 9042 E -6],
[19.75,0.568812,0.5688120435 8652883009,0.4358652883 01 E -7],

359


<table>
<thead>
<tr>
<th>R</th>
<th>Type: List(List(Float))</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>E 3</td>
</tr>
<tr>
<td>S</td>
<td>4 of 4</td>
</tr>
</tbody>
</table>

```
pearcys:=

[ [0.00, 0.0000000], [0.25, 0.0330970], [0.50, 0.0923658], [0.75, 0.1659294],
  [1.00, 0.2475583], [1.25, 0.32216], [1.50, 0.41583], [1.75, 0.493469],
  [2.00, 0.469849], [2.25, 0.520994], [2.50, 0.565787], [2.75, 0.596174],
  [3.00, 0.711685], [3.25, 0.712666], [3.50, 0.700180], [3.75, 0.675925],
  [4.00, 0.642119], [4.25, 0.601362], [4.50, 0.556489], [4.75, 0.510408],
  [5.00, 0.469546], [5.25, 0.42577], [5.50, 0.391834], [5.75, 0.366161],
  [6.00, 0.349852], [6.25, 0.34503], [6.50, 0.347099], [6.75, 0.360040],
  [7.00, 0.381195], [7.25, 0.340889], [7.50, 0.441486], [7.75, 0.476568],
  [8.00, 0.512010], [8.25, 0.546383], [8.50, 0.575457], [8.75, 0.599788],
  [9.00, 0.617214], [9.25, 0.626948], [9.50, 0.628573], [9.75, 0.622204],
  [10.00, 0.608436], [10.25, 0.588297], [10.50, 0.563176], [10.75, 0.53731],
  [11.00, 0.504784], [11.25, 0.475208], [11.50, 0.447809], [11.75, 0.424220],
  [12.00, 0.405810], [12.25, 0.393601], [12.50, 0.388217], [12.75, 0.389852],
  [13.00, 0.398268], [13.25, 0.412817], [13.50, 0.432489], [13.75, 0.455978],
  [14.00, 0.481770], [14.25, 0.508326], [14.50, 0.533736], [14.75, 0.556716],
  [15.00, 0.575803], [15.25, 0.589887], [15.50, 0.598183], [15.75, 0.600273],
  [16.00, 0.596126], [16.25, 0.586095], [16.50, 0.570890], [16.75, 0.551526],
  [17.00, 0.529259], [17.25, 0.505605], [17.50, 0.481750], [17.75, 0.459460],
  [18.00, 0.439989], [18.25, 0.424500], [18.50, 0.413893], [18.75, 0.408757],
  [19.00, 0.409336], [19.25, 0.415520], [19.50, 0.426853], [19.75, 0.442271],
  [20.00, 0.461664], [20.25, 0.482860], [20.50, 0.504876], [20.75, 0.526323],
  [21.00, 0.545885], [21.25, 0.562375], [21.50, 0.574811], [21.75, 0.582472],
  [22.00, 0.584939], [22.25, 0.582119], [22.50, 0.574246], [22.75, 0.561862],
  [23.00, 0.545782], [23.25, 0.527040], [23.50, 0.506824], [23.75, 0.486399],
```
[24.00, 0.467029], [24.25, 0.449901], [24.50, 0.436051], [24.75, 0.426303],
[25.00, 0.421217], [25.25, 0.421062], [25.50, 0.425797], [25.75, 0.435083],
[26.00, 0.448300], [26.25, 0.464594], [26.50, 0.482927], [26.75, 0.502146],
[27.00, 0.521054], [27.25, 0.538483], [27.50, 0.553369], [27.75, 0.564814],
[28.00, 0.572142], [28.25, 0.574935], [28.50, 0.573060], [28.75, 0.564674],
[29.00, 0.566212], [29.25, 0.542357], [29.50, 0.525995], [29.75, 0.508160],
[30.00, 0.489969], [30.25, 0.472549], [30.50, 0.456974], [30.75, 0.444193],
[31.00, 0.434973], [31.25, 0.429857], [31.50, 0.429129], [31.75, 0.432799],
[32.00, 0.440605], [32.25, 0.452031], [32.50, 0.466343], [32.75, 0.482632],
[33.00, 0.499873], [33.25, 0.516992], [33.50, 0.532930], [33.75, 0.546708],
[34.00, 0.557490], [34.25, 0.564629], [34.50, 0.567709], [34.75, 0.566570],
[35.00, 0.563133], [35.25, 0.552293], [35.50, 0.540994], [35.75, 0.525495],
[36.00, 0.509417], [36.25, 0.492866], [36.50, 0.476871], [36.75, 0.462420],
[37.00, 0.437971], [37.25, 0.444626], [37.50, 0.454670], [37.75, 0.467461],
[38.00, 0.433771], [38.25, 0.437924], [38.50, 0.453635], [38.75, 0.473144],
[39.00, 0.428178], [39.25, 0.409724], [39.50, 0.513690], [39.75, 0.528507],
[40.00, 0.541588], [40.25, 0.557628], [40.50, 0.558799], [40.75, 0.562140],
[41.00, 0.561608], [41.25, 0.557258], [41.50, 0.549384], [41.75, 0.538494],
[42.00, 0.525282], [42.25, 0.510580], [42.50, 0.495309], [42.75, 0.480418],
[43.00, 0.466829], [43.25, 0.455375], [43.50, 0.446755], [43.75, 0.441847],
[44.00, 0.439878], [44.25, 0.442007], [44.50, 0.447720], [44.75, 0.456645],
[45.00, 0.468209], [45.25, 0.481681], [45.50, 0.496215], [45.75, 0.510904],
[46.00, 0.526238], [46.25, 0.537153], [46.50, 0.547099], [46.75, 0.554070],
[47.00, 0.557650], [47.25, 0.556835], [47.50, 0.550404], [47.75, 0.547120],
[48.00, 0.537308], [48.25, 0.525234], [48.50, 0.511657], [48.75, 0.497426],
[49.00, 0.483428], [49.25, 0.470529], [49.50, 0.459523], [49.75, 0.451084],
[50.00, 0.445722]]


```plaintext
CHAPTER 5. CHAPTER D

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

---

```

```
PACKAGE DFSFUN DOUBLEFLOATSPECIALFUNCTIONS
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R

[2.75,0.696174,0.6961742312 5801822896,0.2312580182 29 E -6],
[3.0,0.711685,0.7116850216 0753003252,0.2160753003 25 E -7],
[3.25,0.712666,0.7126658336 0193196899,- 0.1663980680 31 E -6],
[3.5,0.70018,0.7001803262 1710404986,0.3262171040 499 E -6],
[3.75,0.675925,0.6759254869 8611906826,0.4869861190 6826 E -6],
[4.0,0.642119,0.6421187357 4451469534,- 0.2642554853 047 E -6],
[4.25,0.601362,0.6013618861 0406611871,- 0.1138959338 813 E -6],
[4.5,0.556489,0.5564893045 0127589586,0.3045012758 959 E -6],
[4.75,0.510408,0.5104084694 9752509967,0.4694975250 997 E -6],
[5.0,0.465942,0.4659414967 6625853241,- 0.5032337414 6759 E -6],
[5.25,0.425677,0.4256760562 5372584118,- 0.9437462741 5882 E -6],
[5.5,0.391834,0.3918334884 2241092549,- 0.5115775890 7451 E -6],
[5.75,0.366161,0.3661608711 7846843575,- 0.1288215315 642 E -6],
[6.0,0.349852,0.3498523653 5397811419,0.3653539781 1419 E -6],
[6.25,0.343503,0.3435034609 0379470942,0.4609037947 0942 E -6],
[6.5,0.347099,0.3470998591 9509107808,0.8591950910 7808 E -6],
[6.75,0.36004,0.3600407686 5353610359,0.7686535361 0359 E -6],
[7.0,0.381195,0.3811944739 4496760982,- 0.5260550323 9018 E -6],
[7.25,0.408982,0.4089822714 33853148,0.2714338531 48 E -6],
[7.5,0.441485,0.4414853446 1004457156,0.3446100445 7156 E -6],
[7.75,0.476568,0.4765679658 3912270914,- 0.3416087729 086 E -7],
[8.0,0.51201,0.5120096184 674641167,- 0.3815325358 833 E -6],
[8.25,0.545638,0.5456382758 4836770255,0.2758483677 026 E -6],
[8.5,0.575457,0.5754571656 0424747441,0.1656042474 744 E -6],
[8.75,0.599758,0.5997578767 1218929607,- 0.1232878107 039 E -6],
[9.0,0.617214,0.6172135970 241896115,- 0.4029758103 885 E -6],
[9.25,0.626948,0.6269475401 6193698812,- 0.4598380630 119 E -6],
[9.5,0.628573,0.6285731549 3626283028,0.1549362628 303 E -6],
[9.75,0.622204,0.6222044149 0114838121,0.4149011483 812 E -6],
[10.0,0.608436,0.6084362590 651108963,0.2590651108 963 E -6],
[10.25,0.588297,0.5882969931 8226702044,- 0.6817732979 56 E -8],
[10.5,0.563176,0.5631760638 822475396,0.6388224753 96 E -7],
[10.75,0.534731,0.5347319938 6555282657,0.9938655528 2656 E -6],
[11.0,0.504784,0.5047863386 4734203894,0.0000023386 473420389],
[11.25,0.475208,0.4752102358 2255398909,0.0000022358 2255398909],
[11.5,0.447809,0.4478104304 5274307166,0.0000014304 527430717],
[11.75,0.42422,0.4242215626 5460748949,0.0000015626 546074895],
[12.0,0.40581,0.4058110077 5914323067,0.0000010077 591432307],
[12.25,0.393601,0.3936017007 1076950741,0.7007107695 0741 E -6],
[12.5,0.388217,0.3882172107 7941192612,0.2107794119 261 E -6],
[12.75,0.389852,0.3898519335 8673393733,- 0.6641326606 266 E -7],
[13.0,0.398268,0.3982677211 0844849665,- 0.2788915515 0335 E -6],
[13.25,0.412817,0.4128166704 5498856625,- 0.3295450114 3375 E -6],
[13.5,0.432489,0.4324882345 041217611,- 0.7654958782 389 E -6],
[13.75,0.455978,0.4559773937 3230061439,- 0.6062676993 8561 E -6],
[14.0,0.48177,0.4817694215 5974436016,- 0.5784402556 3984 E -6],
[14.25,0.508236,0.5082358539 0961301684,- 0.1460903869 832 E -6],
[14.5,0.533736,0.5337356883 2480717037,- 0.3116751928 296 E -6],
[14.75,0.556716,0.5567156190 4121537424,- 0.3809587846 258 E -6],
[15.0,0.575803,0.5758032698 0780549342,0.2698078054 934 E -6],

365


PACKAGE DFSFUN DOUBLEFLOATSPECIALFUNCTIONS
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R
--R

[27.75,0.564814,0.5648148314 9166491027,0.8314916649 1027 E -6],
[28.0,0.572142,0.5721420631 6790330366,0.6316790330 366 E -7],
[28.25,0.574935,0.5749345026 8645175971,- 0.4973135482 4029 E -6],
[28.5,0.57306,0.5730594822 652482619,- 0.5177347517 3809 E -6],
[28.75,0.566674,0.5666739801 1092060771,- 0.1988907939 23 E -7],
[29.0,0.556212,0.5562123974 5318039664,0.3974531803 966 E -6],
[29.25,0.542357,0.5423573357 3057143592,0.3357305714 359 E -6],
[29.5,0.525995,0.5259953183 8591850851,0.3183859185 085 E -6],
[29.75,0.50816,0.5081603114 6691447077,0.3114669144 708 E -6],
[30.0,0.489969,0.4899686293 6294561315,- 0.3706370543 8685 E -6],
[30.25,0.472549,0.4725493003 70056332,0.3003700563 32 E -6],
[30.5,0.456974,0.4569742329 8007908723,0.2329800790 872 E -6],
[30.75,0.444193,0.4441924732 1202200062,- 0.5267879779 9938 E -6],
[31.0,0.434973,0.4349725872 2359219467,- 0.4127764078 0533 E -6],
[31.25,0.429857,0.4298566440 1538454345,- 0.3559846154 5655 E -6],
[31.5,0.429129,0.4291285439 2130018393,- 0.4560786998 1607 E -6],
[31.75,0.432799,0.4327985404 7868362044,- 0.4595213163 7956 E -6],
[32.0,0.440605,0.4406047705 8303021964,- 0.2294169697 804 E -6],
[32.25,0.452031,0.4520315818 7779157328,0.5818777915 7328 E -6],
[32.5,0.466343,0.4663433562 2850339586,0.3562285033 9586 E -6],
[32.75,0.482632,0.4826316757 5316315801,- 0.3242468368 4199 E -6],
[33.0,0.499873,0.4998728498 5761360978,- 0.1501423863 902 E -6],
[33.25,0.516992,0.5169919568 4608062411,- 0.4315391937 59 E -7],
[33.5,0.53293,0.5329297341 0588518507,- 0.2658941148 149 E -6],
[33.75,0.546708,0.5467080330 374204341,0.3303742043 41 E -7],
[34.0,0.55749,0.5574894980 70161353,- 0.5019298386 47 E -6],
[34.25,0.564629,0.5646284739 1529463715,- 0.5260847053 6285 E -6],
[34.5,0.567709,0.5677093003 3772777525,0.3003377277 753 E -6],
[34.75,0.56657,0.5665706285 1241575866,0.6285124157 5866 E -6],
[35.0,0.561313,0.5613133551 8174616414,0.3551817461 641 E -6],
[35.25,0.552293,0.5522925303 0361142379,- 0.4696963885 762 E -6],
[35.5,0.540094,0.5400936428 6312004459,- 0.3571368799 554 E -6],
[35.75,0.525495,0.5254947452 6478477127,- 0.2547352152 287 E -6],
[36.0,0.509417,0.5094168513 7281277168,- 0.1486271872 283 E -6],
[36.25,0.492866,0.4928661334 9354600731,0.1334935460 073 E -6],
[36.5,0.476871,0.4768716484 1410150512,0.6484141015 0512 E -6],
[36.75,0.46242,0.4624199779 4599876237,- 0.2205400123 76 E -7],
[37.0,0.450396,0.4503960869 296054767,0.8692960547 67 E -7],
[37.25,0.441528,0.4415290525 3642844985,0.0000010525 364284499],
[37.5,0.436345,0.4363459415 0326090466,0.9415032609 0466 E -6],
[37.75,0.435144,0.4351444622 142641974,0.4622142641 974 E -6],
[38.0,0.437971,0.4379709919 0155134798,- 0.8098448652 02 E -8],
[38.25,0.444626,0.4446262686 7031444849,0.2686703144 4849 E -6],
[38.5,0.45467,0.4546708406 3699119293,0.8406369911 9293 E -6],
[38.75,0.467461,0.4674625380 8725087486,0.0000015380 872508749],
[39.0,0.482187,0.4821902969 1952370995,0.0000032969 1952370995],
[39.25,0.497924,0.4979286483 1116955857,0.0000046483 1116955857],
[39.5,0.51369,0.5136882217 6583353419,- 0.0000017782 341664658],
[39.75,0.528507,0.5285123707 6397702394,0.0000053707 6397702393],
[40.0,0.541464,0.5414672429 1598081817,0.0000032429 159808182],

367


--R [40.25,0.551768,0.5517745240 9094539537,0.0000065240 9094539537],
--R [40.5,0.558799,0.5588050506 5912486677,0.0000060506 5912486677],
--R [40.75,0.56214,0.5621369791 6412056122,- 0.0000030208 358794388],
--R [41.0,0.561608,0.5616247711 0814605692,0.0000067634 9485623221],
--R [41.25,0.557258,0.5572675955 430330709,0.0000096955 430330709],
--R [41.5,0.549384,0.5493773200 0894645567,- 0.0000066799 9105354433],
--R [41.75,0.538494,0.5385616342 9485623221,0.0000067634 9485623221],
--R [42.0,0.525282,0.5252752111 2342898796,- 0.0000067888 7657101203],
--R [42.25,0.51058,0.5104939314 7702373041,- 0.0000068065 2297626959],
--R [42.5,0.495309,0.4953417684 038968965,0.0000327684 038968965],
--R [42.75,0.480418,0.4806241406 5752533599,0.0000261406 5752533599],
--R [43.0,0.466829,0.4667676898 4497320465,- 0.0000061310 1502679535],
--R [43.25,0.455375,0.4553638694 6250373139,- 0.0000111305 3749626868],
--R [43.5,0.446755,0.446829634 3044199959,- 0.0000072036 6955580042],
--R [43.75,0.441487,0.4413450709 5947820527,- 0.0001419290 4052179471],
--R [44.0,0.439878,0.4398575523 27691464,- 0.000204476 72308536],
--R [44.25,0.442007,0.4420222062 7982284284,0.000152062 7982284284],
--R [44.5,0.44772,0.4478984397 6769945092,0.0001784397 6769945092],
--R [44.75,0.456645,0.4569237692 7106515615,- 0.0007176307 2890484385],
--R [45.0,0.468209,0.4676487917 6051833044,- 0.0005620822 3948169966],
--R [45.25,0.481681,0.4819138773 9121176935,- 0.0002328773 9121176935],
--R [45.5,0.496215,0.4954833940 8753370091,- 0.0007316509 1246629909],
--R [45.75,0.510904,0.5106859986 5658467469,- 0.0002180013 434153253],
--R [46.0,0.524837,0.524275127 6791944478,- 0.0005644872 3028055222],
--R [46.25,0.537153,0.5375126919 1563930134,- 0.0035969195 1563903413],
--R [46.5,0.547099,0.5469626247 9105833281,- 0.0010027352 089411672],
--R [46.75,0.55407,0.5524834800 5502643628,- 0.0015865199 449735637],
--R [47.0,0.55765,0.5580597398 2343230309,- 0.0004097398 2343230309],
--R [47.25,0.55635,0.5590882802 4170920081,- 0.0014530282 4170920081],
--R [47.5,0.55404,0.5475186455 874275914,- 0.0065255344 1257240864],
--R [47.75,0.54712,0.545984940 601030917,- 0.0015215059 3989690833],
--R [48.0,0.537309,0.5196318344 7879609501,- 0.0176771655 210239404],
--R [48.25,0.525234,0.5131703830 5664910816,- 0.0120636199 433508919],
--R [48.5,0.51667,0.487031698 1757360562,- 0.0243563061 982426393],
--R [48.75,0.497426,0.5181479895 3691666632,0.0207219895 3691666632],
--R [49.0,0.483428,0.4596231777 3318010769,- 0.0235656822 6861989923],
--R [49.25,0.470629,0.4397312463 9609993647,- 0.0307797756 039000035],
--R [49.5,0.459523,0.4154357479 1719610661,- 0.0438794250 8280389339],
--R [49.75,0.451084,0.4230292350 336881258,0.0280647649 663118742],
--R [50.0,0.445722,0.3044252284 9276786818,- 0.1412967715 072321138]]
--R Type: List(List(Float))
--E 5

)spool
)lisp (bye)
DoubleFloatSpecialFunctions examples
====================================================================
This package provides special functions for double precision real and complex floating point.

The formula used will agree with the Table of the Fresnel Integral by Pearcey (1959) to 6 decimal places up to an argument of about 35.0. After that the summation gets slowly worse, agreeing to only 2 digits at about 45.0.

fresnelC(1.5)
    0.7790837385 0396370968

fresnelS(1.5)
    0.4154833182 6565542581

See Also:
o )show DoubleFloatSpecialFunctions

DoubleFloatSpecialFunctions (DFSFUN)

Exports:
airyAi        airyBi        besselI        besselJ        besselK
besselY       Beta          digamma       E1             Ei
Ei1           Ei2           Ei3            Ei4            Ei5
Ei6           En            fresnelC       fresnelS       Gamma
hypergeometric0F1 logGamma     polygamma
--- package DFSFUN DoubleFloatSpecialFunctions ---

)abbrev package DFSFUN DoubleFloatSpecialFunctions
++ Author: Bruce W. Char, Timothy Daly, Stephen M. Watt
++ Date Created: 1990
++ Date Last Updated: Jan 19, 2008
++ Description:
++ This package provides special functions for double precision
++ real and complex floating point.

DoubleFloatSpecialFunctions(): Exports == Impl where
  NNI ==> NonNegativeInteger
  PI  ==> Integer
  R   ==> DoubleFloat
  C   ==> Complex DoubleFloat
  OPR ==> OnePointCompletion R
  F   ==> Float
  LF  ==> List Float

Exports ==> with
  Gamma: R -> R
  ++ Gamma(x) is the Euler gamma function, \spad{Gamma(x)}, defined by
  ++ \spad{Gamma(x) = integrate(t^(x-1)*exp(-t), t=0..%infinity)}.
  Gamma: C -> C
  ++ Gamma(x) is the Euler gamma function, \spad{Gamma(x)}, defined by
  ++ \spad{Gamma(x) = integrate(t^(x-1)*exp(-t), t=0..%infinity)}.
  E1: R -> OPR
  ++ E1(x) is the Exponential Integral function
  ++ The current implementation is a piecewise approximation
  ++ involving one poly from -4..4 and a second poly for x > 4
  En: (PI,R) -> OPR
  ++ En(n,x) is the nth Exponential Integral Function
  Ei: (OPR) -> OPR
  ++ Ei is the Exponential Integral function
  ++ This is computed using a 6 part piecewise approximation.
  ++ DoubleFloat can only preserve about 16 digits but the
  ++ Chebyshev approximation used can give 30 digits.
  Ei1: (OPR) -> OPR
  ++ Ei1 is the first approximation of Ei where the result is
  ++ x*%e^-x*Ei(x) from -infinity to -10 (preserves digits)
  Ei2: (OPR) -> OPR
  ++ Ei2 is the first approximation of Ei where the result is
  ++ x*%e^-x*Ei(x) from -10 to -4 (preserves digits)
Ei3: (OPR) -> OPR  
++ Ei3 is the first approximation of Ei where the result is  
++ (Ei(x) - log |x| - gamma)/x from -4 to 4 (preserves digits)

Ei4: (OPR) -> OPR  
++ Ei4 is the first approximation of Ei where the result is  
++ x*%e^-x*Ei(x) from 4 to 12 (preserves digits)

Ei5: (OPR) -> OPR  
++ Ei5 is the first approximation of Ei where the result is  
++ x*%e^-x*Ei(x) from 12 to 32 (preserves digits)

Ei6: (OPR) -> OPR  
++ Ei6 is the first approximation of Ei where the result is  
++ x*%e^-x*Ei(x) from 32 to infinity (preserves digits)

Beta: (R, R) -> R  
++ Beta(x, y) is the Euler beta function, \( B(x,y) \), defined by  
++ \( \int_0^1 t^{x-1}(1-t)^{y-1} dt \).  
++ This is related to \( \Gamma(x) \) by  
++ \( \Gamma(x)*\Gamma(y) / \Gamma(x + y) \).

Beta: (C, C) -> C  
++ Beta(x, y) is the Euler beta function, \( B(x,y) \), defined by  
++ \( \int_0^1 t^{x-1}(1-t)^{y-1} dt \).  
++ This is related to \( \Gamma(x) \) by  
++ \( \Gamma(x)*\Gamma(y) / \Gamma(x + y) \).

logGamma: R -> R  
++ logGamma(x) is the natural log of \( \Gamma(x) \).  
++ This can often be computed even if \( \Gamma(x) \) cannot.

logGamma: C -> C  
++ logGamma(x) is the natural log of \( \Gamma(x) \).  
++ This can often be computed even if \( \Gamma(x) \) cannot.

digamma: R -> R  
++ digamma(x) is the function, \( \psi(x) \), defined by  
++ \( \psi(x) = \Gamma'(x)/\Gamma(x) \).

digamma: C -> C  
++ digamma(x) is the function, \( \psi(x) \), defined by  
++ \( \psi(x) = \Gamma'(x)/\Gamma(x) \).

polygamma: (NNI, R) -> R  
++ polygamma(n, x) is the n-th derivative of \( \psi(x) \).

polygamma: (NNI, C) -> C  
++ polygamma(n, x) is the n-th derivative of \( \psi(x) \).

besselJ: (R,R) -> R  
++ besselJ(v, x) is the Bessel function of the first kind,  
++ \( J(v, x) \).  
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.

\textbf{besselJ: (C,C) \to C}
++ \text{besselJ}(v,x) is the Bessel function of the first kind,
++ \spad{J(v,x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.

\textbf{besselY: (R, R) \to R}
++ \text{besselY}(v,x) is the Bessel function of the second kind,
++ \spad{Y(v,x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) + (x^2-v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \spad{Y(v,x) = (J(v,x) \cos(v\%pi) - J(-v,x))/\sin(v\%pi)}
++ so is not valid for integer values of \(v\).

\textbf{besselI: (R,R) \to R}
++ \text{besselI}(v,x) is the modified Bessel function of the first kind,
++ \spad{I(v,x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.

\textbf{besselI: (C,C) \to C}
++ \text{besselI}(v,x) is the modified Bessel function of the first kind,
++ \spad{I(v,x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.

\textbf{besselK: (R, R) \to R}
++ \text{besselK}(v,x) is the modified Bessel function of the second kind,
++ \spad{K(v,x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \spad{K(v,x) = \%pi/2*(I(-v,x) - I(v,x))/\sin(v\%pi)}.
++ so is not valid for integer values of \(v\).

\textbf{besselK: (C, C) \to C}
++ \text{besselK}(v,x) is the modified Bessel function of the second kind,
++ \spad{K(v,x)}.
++ This function satisfies the differential equation:
++ \spad{x^2 w''(x) + x w'(x) - (x^2+v^2)w(x) = 0}.
++ Note that the default implementation uses the relation
++ \spad{K(v,x) = \%pi/2*(I(-v,x) - I(v,x))/\sin(v\%pi)}.
++ so is not valid for integer values of \(v\).

\[\text{airyAi: } \mathbb{C} \rightarrow \mathbb{C}\]
++ \(\text{airyAi}(x)\) is the Airy function \(\spad{Ai(x)}\).
++ This function satisfies the differential equation:
++ \(\spad{Ai''(x) - x \cdot Ai(x) = 0}\).

\[\text{airyAi: } \mathbb{R} \rightarrow \mathbb{R}\]
++ \(\text{airyAi}(x)\) is the Airy function \(\spad{Ai(x)}\).
++ This function satisfies the differential equation:
++ \(\spad{Ai''(x) - x \cdot Ai(x) = 0}\).

\[\text{airyBi: } \mathbb{R} \rightarrow \mathbb{R}\]
++ \(\text{airyBi}(x)\) is the Airy function \(\spad{Bi(x)}\).
++ This function satisfies the differential equation:
++ \(\spad{Bi''(x) - x \cdot Bi(x) = 0}\).

\[\text{airyBi: } \mathbb{C} \rightarrow \mathbb{C}\]
++ \(\text{airyBi}(x)\) is the Airy function \(\spad{Bi(x)}\).
++ This function satisfies the differential equation:
++ \(\spad{Bi''(x) - x \cdot Bi(x) = 0}\).

\[\text{hypergeometric0F1: } \mathbb{R} \rightarrow \mathbb{R}\]
++ \(\text{hypergeometric0F1}(c,z)\) is the hypergeometric function
++ \(\spad{0F1(; c; z)}\).

\[\text{hypergeometric0F1: } \mathbb{C} \rightarrow \mathbb{C}\]
++ \(\text{hypergeometric0F1}(c,z)\) is the hypergeometric function
++ \(\spad{0F1(; c; z)}\).

\[\text{fresnelS : } \mathbb{F} \rightarrow \mathbb{F}\]
++ \(\text{fresnelS}(f)\) denotes the Fresnel integral \(S\)
++ \(\spaddot{X}\) \(\text{fresnelS}(1.5)\)

\[\text{fresnelC : } \mathbb{F} \rightarrow \mathbb{F}\]
++ \(\text{fresnelC}(f)\) denotes the Fresnel integral \(C\)
++ \(\spaddot{X}\) \(\text{fresnelC}(1.5)\)

\begin{verbatim}
Impl ==> add
  a, v, w, z: C
  n, x, y: R

  Gamma z == CGAMMA(z)$Lisp
  Gamma x == RGAMMA(x)$Lisp

\end{verbatim}
The Exponential Integral

The E1 function

(Quoted from Segletes[Segl98]):

A number of useful integrals exist for which no exact solutions have been found. In other cases, an exact solution, if found, may be impractical to utilize over the complete domain of the function because of precision limitations associated with what usually ends up as a series solution to the challenging integral. For many of these integrals, tabulated values may be published in various mathematical handbooks and articles. In some handbooks, fits (usually piecewise) also are offered. In some cases, an application may be forced to resort to numerical integration in order to acquire the integrated function. In this context, compact (i.e. not piecewise) analytical fits to some of these problematic integrals, accurate to within a small fraction of the numerically integrated value, serve as a useful tool to applications requiring the results of the integration, especially when the integration is required numerous times throughout the course of the application. Furthermore, the ability and methodology to develop intelligent fits, in contract to the more traditional “brute force” fits, provide the means to minimize parameters and maximize accuracy when tackling some of these difficult functions. The exponential integral will be used as an opportunity to both demonstrate a methodology for intelligent fitting as well as for providing an accurate, compact, analytical fit to the exponential integral.

The exponential integral is a useful class of functions that arise in a variety of applications [...]. The real branch of the family of exponential integrals may be defined as

\[ E_n(x) = x^{n-1} \int_x^\infty \frac{e^{-t}}{t^n} \, dt \] (5.1)

where \( n \), a positive integer, denotes the specific member of the exponential integral family. The argument of the exponential integral, rather than expressing a lower limit of integration as in (1), may be thought of as describing the exponential decay constant, as given in this equivalent (and perhaps more popular) definition of the integral:

\[ E_n(x) = \int_1^\infty \frac{e^{-xt}}{t^n} \, dt \] (5.2)

Integration by parts permits any member of the exponential integral family to be converted to an adjacent member of the family, by way of

\[ \int_x^\infty \frac{e^{-t}}{t^{n+1}} \, dt = \frac{1}{n} \left( \frac{e^{-x}}{x^n} - \int_x^\infty \frac{e^{-t}}{t^n} \, dt \right) \] (5.3)

expressable in terms of \( E_n \) as

\[ E_{n+1}(x) = \frac{1}{n} \left[ e^{-x} - xE_n(x) \right] \quad (n = 1, 2, 3) \] (5.4)

Through recursive employment of this equation, all members of the exponential integral family may be analytically related. However, this technique only allows for the transformation
of one integral into another. There remains the problem of evaluating $E_1(x)$. There is an exact solution to the integral of $(e^{-t}/t)$, appearing in a number of mathematical references which is obtainable by expanding the exponential into a power series and integrating term by term. That exact solution, which is convergent, may be used to specify $E_1(x)$ as

$$E_1(x) = -\gamma - \ln(x) + \frac{x}{1!} - \frac{x^2}{2 \cdot 2!} + \frac{x^3}{3 \cdot 3!} - \ldots$$

(5.5)

Euler's constant, $\gamma$, equal to 0.57721..., arises when the power series expansion for $(e^{-t}/t)$ is integrated and evaluated at its upper limit, as $x \to \infty$.

Employing eqn (5), however, to evaluate $E_1(x)$ is problematic for finite $x$ significantly larger than unity. One may well ask of the need to evaluate the exponential integral for large $x$, since the function to be integrated drops off so rapidly that the integral is surely a very flat function. Such reasoning is true when comparing the integrand at large $x$ to that at small $x$. However, the definition of eqn (1) has as its upper limit not a small value of $x$, but rather that of $\infty$. Therefore, the actual values for $E_n(x)$ are extremely small numbers for large values of $x$. Thus, it is not sufficient merely to select enough terms of eqn (5) to evaluate the integral to within a value of, for example ±0.0001 because the actual integral value for large $x$ would be smaller than this arbitrary tolerance. To draw an analogy, it would be like saying that it is good enough to approximate $e^{-x}$ as 0.0 for $x > 10$, since its actual value is within 0.0001 of zero. For some applications, such an approximation may be warranted. In general, though, such an approximation is mathematically unacceptable. Worse yet, as seen from eqns (1) and (2), the need to evaluate the exponential integral for large arguments can arise in real-world problems from either a large integration limit or a large value of an exponential decay constant. Thus, the need to evaluate exponential integrals for large values of the argument is established. It is here that the practical problems with the evaluation of eqn (5) become manifest.

First, the number of terms, $N$, required to achieve convergence rises rapidly with increasing $x$, making the summation an inefficient tool, even when expressed as a recursion relation (for three digits of accuracy, $N$ is observed to vary roughly as $9 + 1.6x$, for $1 < x < 7$). More important, however, is the fact that, for calculations of finite precision, the accuracy of the complete summation will be governed by the individual term of greatest magnitude. The source of the problem is that as $x$ is increased, the total summation decreases in magnitude more rapidly than a decaying exponential, while at the same time, the largest individual term in the series is observed to grow rapidly with increasing $x$ ($10^1$ for $x = 7$, $10^2$ for $x = 10$, $10^3$ for $x = 13$, etc.). The magnitude of this largest individual term consumes the available precision and, as a result, leaves little or none left for the ever-diminishing net sum that constitutes the desired integral.

Literally, the use of eqn (5), even with (32-bit) double precision, does not permit the exponential integral to be evaluated to three places for $x > 14$ in any case, and with the situation worsening for lesser precision. For these reasons, the use of eqn (5) to evaluate the exponential integral numerically for large $x$ is wholly unsuitable.
\[ E_1(x) = e^{-x} \cdot \frac{1}{x + \frac{1}{1 + \frac{2}{x + \frac{2}{1 + \cdots}}} \right) \]  

(5.6)

But as \( x \) becomes smaller, the number of terms required for convergence rises quickly. Similar arguments apply for the use of an asymptotic expansion for \( E_1 \), which also converges for large \( x \). As such, the more typical approach employed by handbooks is that of a fit. While some steps are taken to make the fits intelligent (e.g., transformation of variables), the fits are all piecewise over the domain of the integral.

Cody and Thatcher performed what is perhaps the definitive work, with the use of Chebyshev approximations to the exponential integral \( E_1 \). Like others, they fit the integral over a piecewise series of subdomains (three in their case) and provide the fitting parameters necessary to evaluate the function to various required precisions, down to relative errors of \( 10^{-20} \). One of the problems with piecewise fitting over two or more subdomains is that functional value and derivatives of the spliced fits will not, in general, match at the domain transition point, unless special accommodations are made. This sort of discontinuity in functional value and/or slope, curvature, etc., may cause difficulties for some numerical algorithms operating upon the fitted function. Numerical splicing/smoothing algorithms aimed at eliminating discontinuities in the value and/or derivatives of a piecewise fit are not, in general, computationally insignificant. Problems associated with piecewise splicing of fits may also be obviated by obtaining an accurate enough fit, such that the error is on the order of magnitude of the limiting machine precision. This alternative, however, requires the use of additional fitting parameters to acquire the improved precision. Thus, regardless of approach, the desire to eliminate discontinuities in the function and its derivatives, between piecewise splices, requires extra computational effort. One final benefit to be had by avoiding the use of piecewise fits is the concomitant avoidance of conditional (i.e., IF...THEN) programming statements in the coding of the routine. The use of conditional statements can preclude maximum computing efficiency on certain parallel computing architectures.

Segletes constructs an analytic, non-piecewise fit to the Exponential Integral but the precision is on the order of 4 decimal places and is not sufficient to compare against the Abramowitz and Stegun Handbook.

Instead we have chosen to use a two piece fitting function based on the Chebyshev polynomial for computing \( E_1 \). This agrees with the handbook values to almost the last published digit. See the e1.input pamphlet for regression testing against the handbook tables.

\section*{E1:R\rightarrow OPR}

The special function E1 below was originally derived from a function written by T.Haavie as the expint.c function in the Numlibe library by Lars Erik Lund. Haavie approximates the E1 function by two Chebyshev polynomials. For the range \(-4 < x < 4\) the Chebyshev coefficients are:
and for the range $x > 4$ the Chebyshev coefficients are:

- $0.2155283776715125$, $0.1028106215227030$, $-0.0045526707131788$, $0.000379341616932$, $0.00004914394914$, $-0.000007356024922$, $0.0000001230603606$, $-0.000000225236907$, $0.00000044412375$, $-0.00000009328509$, $0.0000002069297$, $-0.0000000481502$, $0.00000000116891$, $-0.0000000029474$, $0.000000000573$, $-0.000000000163$, $0.000000000000047$, $-0.000000000000001$

I’ve rewritten the polynomial to use precomputed coefficients that take into account the scaling used by Haavie. I’ve also rewritten the polynomial using Horner’s method so the large powers of $x$ are only computed once.

The result can be either a double float or, or if the argument is zero, infinity. Thus we need to extend the result to be a one-point completion to include infinity.

— package DFSFUN DoubleFloatSpecialFunctions —
t10::R:=36601.25841454446674::R
ti::R:=(th*x+t10)
t11::R:=279913.28608482691646::R
tj::R:=(ti*x-t11)
t12::R:=2060518.7020296525186::R
tk::R:=(tj*x+t12)
t13::R:=13859772.093039815059::R
tl::R:=(tk*x-t13)
t14::R:=81945572.630072918857::R
tm::R:=(tl*x+t14)
t15::R:=413965714.82128317479::R
tn::R:=(tm*x-t15)
t16::R:=1747209536.259547568::R
to::R:=(tn*x+t16)
t17::R:=6036182333.96179427::R
tp::R:=(to*x-t17)
t18::R:=16693683576.106267572::R
tq::R:=(tp*x+t18)
t19::R:=35938625644.58286097::R
tr::R:=(tq*x-t19)
t20::R:=57888657293.609258888::R
ts::R:=(tr*x+t20)
t21::R:=65523779423.11290127::R
tt::R:=(ts*x-t21)
t22::R:=46422751473.201760309::R
tu::R:=(tt*x+t22)
t23::R:=15474250491.067253436::R
tv::R:=(tu*x-t23)
tw::R:=(-1.0::R*x)
tx::R:=exp(tw)
ty::R:=tv*tx
tz::R:=x**22
taz::OPR
x > -4.0::R =>
a1::R:=0.476837158203125E-22::R
a2::R:=0.10967254638671875E-20::R
aa::R:=(-a1*x+a2)
a3::R:=0.20217895507812500001E-19::R
ab::R:=(aa*x-a3)
a4::R:=0.42600631713867187501E-18::R
ac::R:=((ab*x+a4)
a5::R:=0.868625640869140625E-17::R
ad::R:=((ac*x-a5)
a6::R:=0.16553192138671875E-15::R
ae::R:=((ad*x+a6)
a7::R:=0.29870208740234375E-14::R
af::R:=((ae*x-a7)
a8::R:=0.509789077587890625E-13::R
ag::R:=((af*x+a8)
En:(PI,R)→OPR

The $E_n$ function is computed using the recurrence relation:

$$E_{n+1}(z) = \frac{1}{n} \left( e^{-z} - z E_n(z) \right) \quad (n = 1, 2, 3, \ldots)$$

The base case of the recursion depends on $E_1$ above.

The formula is 5.1.14 in Abramowitz and Stegun, 1965, p229
The Ei Function

This function is based on Kin L. Lee’s work.

Abstract

The exponential integral $Ei(x)$ is evaluated via Chebyshev series expansion of its associated functions to achieve high relative accuracy throughout the entire real line. The Chebyshev coefficients for these functions are given to 30 significant digits. Clenshaw’s method is modified to furnish an efficient procedure for the accurate solution of linear systems having near-triangular coefficient matrices.

Introduction

The evaluation of the exponential integral

$$Ei(x) = \int_{-\infty}^{x} \frac{e^{u}}{u} \, du = -E_{1}(-x), \quad x \neq 0$$

is usually based on the value of its associated functions, for example, $xe^{-x}Ei(x)$. High accuracy tabulations of integral (1) by means of Taylor series techniques are given by Harris and Miller and Hurst. The evaluation of $Ei(x)$ for $-4 \leq x \leq \infty$ by means of Chebyshev series is provided by Clenshaw to have the absolute accuracy of 20 decimal places. The evaluation of the same integral (1) by rational approximation of its associated functions is furnished by Cody and Thacher for $-\infty < x < \infty$, and has the relative accuracy of 17 significant figures.

The approximation of Cody and Thacher from the point of view of efficient function evaluation are preferable to those of Clenshaw. However, the accuracy of the latter’s procedure, unlike those of the former, is not limited by the accuracy or the availability of a master function, which is a means of explicitly evaluating the function in question.

In this paper $Ei(x)$ (or equivalently $-E_{1}(-x)$) for the entire real line is evaluated via Chebyshev series expansion of its associated functions that are accurate to 30 significant figures by a modification of Clenshaw’s procedure. To verify the accuracy of the several Chebyshev series, values of the associated functions were checked against those computed by Taylor series and those of Murnaghan and Wrench (see Remarks on Convergence and Accuracy).

Although for most purposes fewer than 30 figures of accuracy are required, such high accuracy is desirable for the following reasons. In order to further reduce the number of arithmetical
operations in the evaluation of a function, the Chebyshev series in question can either be converted into a rational function or rearranged into an ordinary polynomial. Since several figures may be lost in either of these procedures, it is necessary to provide the Chebyshev series with a sufficient number of figures to achieve the desired accuracy. Furthermore, general function approximation routines, such as those used for minimax rational function approximations, require the explicit evaluation of the function to be approximated. To take account of the errors committed by these routines, the function values must have an accuracy higher than the approximation to be determined. Consequently, high-precision results are useful as a master function for finding approximations for (or involving) $Ei(x)$ where prescribed accuracy is less than 30 figures.

Discussion

It is proposed here to provide for the evaluation of $Ei(x)$ by obtaining Chebyshev coefficients for the associated functions given by table 1.

| Table 1: Associated Functions of $Ei(x)$ and their ranges of Chebyshev Series Expansions |
| --- | --- |
| Associated function | Range of expansion |
| $Ei_1$ | $x e^{-x} Ei(x)$ $-\infty < x \leq -10$ |
| $Ei_2$ | $x e^{-x} Ei(x)$ $-10 \leq x \leq -4$ |
| $Ei_3$ | $\frac{Ei(x)-\log|x|-\gamma}{x}$ $-4 \leq x \leq d42$ |
| $Ei_4$ | $x e^{-x} Ei(x)$ $4 \leq x \leq 12$ |
| $Ei_5$ | $x e^{-x} Ei(x)$ $12 \leq x \leq 32$ |
| $Ei_6$ | $x e^{-x} Ei(x)$ $32 \leq x < \infty$ |

($\gamma = 0.5772156649...$ is Euler’s constant.)

— package DFSFUN DoubleFloatSpecialFunctions —

```lisp
Ei(y:OPR):OPR ==
  infinite? y => 1
  x:R:=retract(y)
  x < -10.0::R =>
    ei:R:=retract(Ei1(y))
    (ei/(x*exp(-x)))::OPR
  x < -4.0::R =>
    ei:R:=retract(Ei2(y))
    (ei/(x*exp(-x)))::OPR
  x < 4.0::R =>
    ei:R:=retract(Ei3(y))
    gamma:R:=0.5772156649015328606512090082::R
    (ei3*x+log(abs(x))+gamma)::OPR
  x < 12.0::R =>
    ei:R:=retract(Ei4(y))
    (ei/(x*exp(-x)))::OPR
  x < 32.0::R =>
    ei:R:=retract(Ei5(y))
    (ei/(x*exp(-x)))::OPR
```
Note that the functions \( Ei(x) = \log|x| - \gamma \) and \( xe^{-x} Ei(x) \) have the limiting values of unity at the origin and at infinity, respectively, and that the range of the associated function values is close to unity (see table 4). This makes for the evaluation of the associated functions over the indicated ranges in table 1 (and thus \( Ei(x) \) over the entire real line) with high relative accuracy by means of the Chebyshev series. The reason for this will become apparent later.

Some remarks about the choice of the intervals of expansion for the several Chebyshev series are in order here. The partition of the real line indicated by table 1 is chosen to allow for the approximation of the associated functions with a maximum error of \( 0.5 \times 10^{-30} \) by polynomials of degree \( < 50 \). The real line has also been partitioned with the objective of providing the interval about zero with the lowest degree of polynomial approximation of the six intervals. This should compensate for the computation of \( \log|x| \) required in the evaluation of \( Ei(x) \) over that interval. The ranges \(-\infty < x < -4 \) and \( 4 < x < \infty \) are partitioned into 2 and 3 intervals, respectively, to provide approximations to \( xe^{-x} Ei(x) \) by polynomials of about the same degree.

**Expansions in Chebyshev Series**

Let \( \phi(t) \) be a differentiable function defined on \([-1,1]\). To facilitate discussion, denote its Chebyshev series and that of its derivative by

\[
\phi(t) = \sum_{k=0}^{\infty} A_k^{(0)} T_k(t) \quad \phi'(t) = \sum_{k=0}^{\infty} A_k^{(1)} T_k(t)
\]  

(5.8)

where \( T_k(t) \) are Chebyshev polynomials defined by

\[
T_k(t) = \cos(k \arccos t), \quad -1 \leq t \leq 1
\]  

(5.9)

(A prime over a summation sign indicates that the first term is to be halved.)

If \( \phi(t) \) and \( \phi'(t) \) are continuous, the Chebyshev coefficients \( A_k^{(0)} \) and \( A_k^{(1)} \) can be obtained analytically (if possible) or by numerical quadrature. However, since each function in table 1 satisfies a linear differential equation with polynomial coefficients, the Chebyshev coefficients can be more readily evaluated by the method of Clenshaw.

There are several variations of Clenshaw’s procedure, but for high-precision computation, where multiple precision arithmetic is employed, we find his original procedure easiest to implement. However, straightforward application of it may result in a loss of accuracy if the trial solutions selected are not sufficiently independent. How the difficulty is overcome will be pointed out subsequently.
The function \(xe^{-x}Ei(x)\) on the Finite Interval

We consider first the Chebyshev series expansion of

\[
f(x) = xe^{-x}Ei(x), \quad (a \leq x \leq b)
\]

with \(x \neq 0\). One can easily verify that after the change of variables

\[
x = [(b - a)T + a + b]/2, \quad (-1 \leq t \leq 1)
\]

the function

\[
\phi(t) = f \left[ \frac{(b - a)t + a + b}{2} \right] = f(x)
\]

satisfies the differential equation

\[
2(pt + q)\phi'(t) + p(pt + q - 2)\phi(t) = p(pt + q)
\]

with\(^1\)

\[
\phi(-1) = ae^{-a}Ei(a)
\]

where \(p = b - a\) and \(q = b + a\). Replacing \(\phi(t)\) and \(\phi'(t)\) in equations 7 by their Chebyshev series, we obtain

\[
\sum_{k=0}^{\infty} (-1)^k A_k^{(0)} = \phi(-1)
\]

\[
2 \sum_{k=0}^{\infty} A_k^{(1)}(pt + q)T_k(t) + p \sum_{k=0}^{\infty} A_k^{(0)}(pt + q - 2)T_k(t) = p(pt + q)
\]

It can be demonstrated that if \(B_k\) are the Chebyshev coefficients of a function \(\Psi(t)\), then \(C_k\), the Chebyshev coefficients of \(t^r\Psi(t)\) for positive integers \(r\), are given by

\[
C_k = 2^{-r} \sum_{i=0}^{r} \binom{r}{i} B_{|k-r+2|}
\]

Consequently, the left member of equation 15 can be rearranged into a single series involving \(T_k(t)\). The comparison of the coefficients of \(T_k(t)\) that yields the infinite system of equations

\[
\sum_{k=0}^{\infty} (-1)^k A_k^{(0)} = \phi(-1)
\]

\[
2pA_{|k-1|}^{(1)} + 4qA_k^{(1)} + 2pA_{k+1}^{(1)} + p^2A_{|k-1|}^{(0)} + 2p(q - 2)A_k^{(0)} + p^2A_{k+1}^{(0)}
\]

\[
= \begin{cases} 
4pq & , k = 0 \\
2p^2 & , k = 1 \\
0 & , k = 2, 3, \ldots
\end{cases}
\]

\(^1\)The value of \(Ei(a)\) may be evaluated by means of the Taylor series. In this report \(Ei(a)\) is computed by first finding the Chebyshev series approximation to \(Ei(x) = \log|x| - \gamma/x\) to get \(Ei(a)\). The quantities \(e^a\) and \(\log|a|\) for integral values of \(a\) may be found in existing tables.
The relation
\[ 2kA_k^{(0)} = A_{k-1}^{(1)} - A_{k+1}^{(1)} \]  
(5.19)
can be used to reduce equation 18 to a system of equations involving only \( A_k^{(0)} \). Thus, replacing \( k \) of equations 18 by \( k + 2 \) and subtracting the resulting equation from equations 18, we have, by means of equation 19, the system of equations
\[
\sum_{k=0}^{\infty} (-1)^k A_k^{(0)} = \phi(-1)
\]
\[
\begin{align*}
2p(q-2)A_0 + (8q+p^2)A_1 + 2p(6-q)A_2 - p^2 A_3 &= 4pq \\
p^2 A_{k-1} + 2p(2k + q - 2)A_k + 8q(k + 1)A_{k+1} + 2p(2k - q + 6)A_{k+2} - p^2 A_{k+3} &= 0
\end{align*}
\]
(5.20)
The superscript of \( A_k^{(0)} \) is dropped for simplicity. In order to solve the infinite system 20, Clenshaw essentially considered the required solution as the limiting solution of the sequence of truncated systems consisting of the first \( M + 1 \) equations of the same system, that is, the solution of the system
\[
\sum_{k=0}^{M} (-1)^k A_k = \phi(-1)
\]
(5.21)
\[
\begin{align*}
2p(q-2)A_0 + (8q+p^2)A_1 + 2p(6-q)A_2 - p^2 A_3 &= 4pq \\
p^2 A_{k-1} + 2p(2k + q - 2)A_k + 8q(k + 1)A_{k+1} + 2p(2k - q + 6)A_{k+2} - p^2 A_{k+3} &= 0
\end{align*}
\]
(5.22)
\[
\begin{align*}
p^2 A_{M-3} + 2p(2M + q - 6)A_{M-2} + 8q(M - 1)A_{M-1} + 2p(2M + 4 - q)A_M &= 0 \\
p^2 A_{M-2} + 2p(2M + q - 4)A_{M-1} + 8qMA_M &= 0
\end{align*}
\]
(5.23)
where \( A_k \) is assumed to vanish for \( K > M + 1 \). To solve system (21,22,23) consider first the subsystem 23 consisting of \( M - 2 \) equations in \( M \) unknowns. Here use is made of the fact that the subsystem 23 is satisfied by
\[
A_k = c_1 \alpha_k + c_2 \beta_k + \gamma_k \quad (k = 0, 1, 2, \ldots)
\]
(5.24)
for arbitrary constants \( c_1 \) and \( c_2 \), where \( \gamma_k \) is a particular solution of 23 and where \( \alpha_k \) and \( \beta_k \) are two independent solutions of the homogeneous equations (23 with \( 2p^2 \) deleted) of the same subsystem. Hence, if \( \alpha_k, \beta_k, \) and \( \gamma_k \) are available, the solution of system (21,22,23) reduces to the determinant of \( c_1 \) and \( c_2 \) from equations 21 and 22.

To solve equations (21,22,23), we note that
\[
\gamma_0 = 2, \quad \gamma_k = 0, \quad \text{for} \ k = 1(1)M
\]
(5.25)
is obviously a particular solution of equation 23. The two independent solutions $\gamma_k$ and $\beta_k$ of the homogeneous equations of the same subsystem can be generated in turn by backward recurrence if we set

$$\begin{align*}
\alpha_{M-1} &= 0, \quad \alpha_M = 1 \\
\beta_{M-1} &= 1, \quad \beta_M = 0
\end{align*}$$

or choose any $\alpha M - 1$, $\alpha M$, and $\beta M - 1$, $\beta M$ for which $\alpha_{M-1} \beta M - \alpha M \beta_{M-1} \neq 0$. The arbitrary constants $c_1$ and $c_2$ are determined, and consequently the solution of equations (21, 22, 23) if equation 24 is substituted into equation 21 and 22 and the resulting equations

$$c_1 R(\alpha) + c_2 R(\beta) = \phi(-1) - 1$$

$$c_1 S(\alpha) + c_2 S(\beta) = 8 \rho$$

are solved as two equations in two unknowns. The terms $R(\alpha)$ and $S(\alpha)$ are equal, respectively, to the left members of equations 21 and 22 corresponding to solution $\alpha_k$. (The identical designation holds for $R(\beta)$ and $S(\beta)$.)

The quantities $\alpha_k$ and $\beta_k$ are known as trial solutions in reference. Clenshaw has pointed out that if $\alpha_k$ and $\beta_k$ are not sufficiently independent, loss of significance will occur in the formation of the linear combination 24, with consequent loss of accuracy. Clenshaw suggested the Gauss-Seidel iteration procedure to improve the accuracy of the solution. However, this requires the application of an additional computing procedure and may prove to be extremely slow. A simpler procedure which does not alter the basic computing scheme given above is proposed here. The loss of accuracy can effectively be regained if we first generate a third trial solution $\delta_k (k=0, 1, \ldots, M)$, where $\delta_{M-1}$ and $\delta_{M}$ are equal to $c_1 \alpha_{M-1} + c_2 \beta_{M-1}$ and $c_1 \alpha_M + c_2 \beta_M$, respectively, and where $\delta_k (k=M-2, M-3, \ldots, 0)$ is determined using backward recurrence as before by means of equation 23. Then either $\alpha_k$ or $\beta_k$ is replaced by $\delta_k$ and a new set of $c_1$ and $c_2$ is determined by equations 27 and 28. Such a procedure can be repeated until the required accuracy is reached. However, only one application of it was necessary in the computation of the coefficients of this report.

As an example, consider the case for $4 \leq x \leq 12$ with $M = 15$. The right member of equation 27 and of equation 28 assume, respectively, the values of 0.43820800 and 64. The trial solutions $\alpha_k$ and $\beta_k$ generated with $\alpha_{14} = 8$, $\alpha_{15} = 9$ and $\beta_{14} = 7$, $\beta_{15} = 8$ are certainly independent, since $\alpha_{14} \beta_{15} - \alpha_{15} \beta_{14} = 1 \neq 0$. A check of table 2 shows that equations 27 and 28 have, respectively, the residuals of $-0.137 \times 10^{-4}$ and $-0.976 \times 10^{-3}$. The same table also shows that $c_1 \alpha_k$ is opposite in sign but nearly equal in magnitude to $c_2 \beta_k$. Cancellations in the formation of the linear combination 24 causes a loss of significance of 2 to 6 figures in the computed $A_k$. In the second iteration, where a new set of $\beta_k$ is generated replacing $\beta_{14}$ and $\beta_{15}$, respectively, by $c_1 \alpha_{14} + c_2 \beta_{14}$ and $c_1 \alpha_{15} + c_2 \beta_{15}$ of the first iteration, the new $c_1 \alpha_k$ and $c_2 \beta_k$ differed from 2 to 5 orders of magnitude. Consequently, no cancellation of significant figures in the computation of $A_k$ occurred. Notice that equations 27 and 28 are now satisfied exactly. Further note that the new $c_1$ and $c_2$ are near zero and unity, respectively, for the reason that if equations 21, 22, and 23 are satisfied by equation 24 exactly in the first iteration, the new $c_1$ and $c_2$ should have the precise values zero and 1, respectively. The results of the third iteration show that the $A_k$ of the second iteration are already accurate to...
eight decimal places, since the $A_k$ in the two iterations differ in less than $0.5 \times 10^{-8}$. Notice that for the third iteration, equations 27 and 28 are also satisfied exactly and that $c_1 = 1$ and $c_2 = 0$ (relative to 8 places of accuracy).

Table 2: Computation of Chebyshev Coefficients for $xe^{-x}Ei(x)$

<table>
<thead>
<tr>
<th>$k$</th>
<th>$c_1 \alpha_k$</th>
<th>$c_2 \beta_k$</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.71690285E 03</td>
<td>-0.71644773E 03</td>
<td>0.24551200E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.33302683E 03</td>
<td>0.33286440E 03</td>
<td>-0.16243000E 00</td>
</tr>
<tr>
<td>2</td>
<td>0.13469341E 03</td>
<td>-0.13464845E 03</td>
<td>0.44960000E-01</td>
</tr>
<tr>
<td>3</td>
<td>-0.43211869E 02</td>
<td>0.43205127E 02</td>
<td>-0.67420000E-02</td>
</tr>
<tr>
<td>4</td>
<td>0.99929173E 01</td>
<td>-0.99942238E 01</td>
<td>-0.13065000E-02</td>
</tr>
<tr>
<td>5</td>
<td>-0.11670764E 01</td>
<td>0.11684574E 01</td>
<td>0.13810000E-02</td>
</tr>
<tr>
<td>6</td>
<td>-0.25552137E 00</td>
<td>0.25493635E 00</td>
<td>-0.58502000E-02</td>
</tr>
<tr>
<td>7</td>
<td>0.20617247E 00</td>
<td>-0.20599754E 00</td>
<td>0.17493000E-03</td>
</tr>
<tr>
<td>8</td>
<td>-0.75797238E-01</td>
<td>0.75756767E-01</td>
<td>-0.40471000E-04</td>
</tr>
<tr>
<td>9</td>
<td>0.20506680E-01</td>
<td>-0.20543463E-01</td>
<td>0.72170000E-05</td>
</tr>
<tr>
<td>10</td>
<td>-0.45193333E-02</td>
<td>0.45183721E-02</td>
<td>-0.86120000E-06</td>
</tr>
<tr>
<td>11</td>
<td>0.82656562E-03</td>
<td>-0.82656898E-03</td>
<td>-0.27000000E-09</td>
</tr>
<tr>
<td>12</td>
<td>-0.12333571E-03</td>
<td>0.12337366E-03</td>
<td>0.37950000E-07</td>
</tr>
<tr>
<td>13</td>
<td>0.13309101E-04</td>
<td>-0.13315328E-04</td>
<td>-0.14418000E-07</td>
</tr>
<tr>
<td>14</td>
<td>-0.29699001E-06</td>
<td>0.30091136E-06</td>
<td>0.39213500E-08</td>
</tr>
<tr>
<td>15</td>
<td>-0.33941716E-06</td>
<td>0.33852528E-06</td>
<td>-0.89188000E-09</td>
</tr>
</tbody>
</table>

$c_1 = 0.37613920E - 07$
$c_2 = -0.42427144E - 07$
$c_1 R(\alpha) + c_2 R(\beta) - 0.43820800E 00 = -0.13700000E - 04$
$c_1 S(\alpha) + c_2 S(\beta) - 0.64000000E 00 = -0.97600000E - 03$

Second iteration: $\alpha_{14} = 8, \alpha_{15} = 9 ; \beta_{14} = 7, \beta_{15} = 8$

$\beta_{14} = 0.39213500E - 08, \beta_{15} = -0.89188000E - 09$
<table>
<thead>
<tr>
<th>k</th>
<th>$c_1 \alpha_k$</th>
<th>$c_2 \beta_k$</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.36701576E-05</td>
<td>0.45512986E 00</td>
<td>0.24551335E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.17051695E-05</td>
<td>-0.16243666E 00</td>
<td>-0.16243837E 00</td>
</tr>
<tr>
<td>2</td>
<td>0.69176566E-06</td>
<td>0.44956834E-01</td>
<td>0.44957523E-01</td>
</tr>
<tr>
<td>3</td>
<td>-0.22132756E-06</td>
<td>-0.67413538E-02</td>
<td>-0.67415751E-02</td>
</tr>
<tr>
<td>4</td>
<td>0.5197561E-07</td>
<td>-0.13067496E-02</td>
<td>-0.13066984E-02</td>
</tr>
<tr>
<td>5</td>
<td>-0.5956744E-08</td>
<td>0.13810895E-02</td>
<td>0.13810835E-02</td>
</tr>
<tr>
<td>6</td>
<td>-0.13059663E-08</td>
<td>-0.58502164E-03</td>
<td>-0.58502944E-03</td>
</tr>
<tr>
<td>7</td>
<td>0.10552667E-08</td>
<td>0.17492889E-03</td>
<td>0.17492994E-03</td>
</tr>
<tr>
<td>8</td>
<td>-0.38808033E-09</td>
<td>-0.40472426E-04</td>
<td>-0.40472814E-04</td>
</tr>
<tr>
<td>9</td>
<td>0.10523831E-09</td>
<td>0.72169965E-03</td>
<td>0.72171016E-03</td>
</tr>
<tr>
<td>10</td>
<td>-0.23146333E-10</td>
<td>-0.86125438E-02</td>
<td>-0.86127752E-02</td>
</tr>
<tr>
<td>11</td>
<td>0.42342615E-11</td>
<td>0.37946968E-02</td>
<td>0.37946336E-02</td>
</tr>
<tr>
<td>12</td>
<td>-0.63200810E-12</td>
<td>-0.14417584E-02</td>
<td>-0.14417516E-02</td>
</tr>
<tr>
<td>13</td>
<td>0.15414832E-13</td>
<td>0.39212981E-08</td>
<td>0.39212965E-08</td>
</tr>
<tr>
<td>14</td>
<td>-0.23083059E-07</td>
<td>-0.45513355E-00</td>
<td>-0.45513355E-00</td>
</tr>
<tr>
<td>15</td>
<td>0.99998675E 00</td>
<td>0.43820800E 00</td>
<td>0.43820800E 00</td>
</tr>
</tbody>
</table>

$\alpha_1 = -1.9268540E-15$  
$\alpha_2 = 0.99998675E 00$  
$c_1 R(\alpha) + c_2 R(\beta) - 0.43820800E 00 = 0.0$  
$c_1 S(\alpha) + c_2 S(\beta) - 0.64000000E 00 = 0.0$

**Table 2:** Computation of Chebyshev Coefficients for $xe^{-x}E_1(x)$ - Concluded

$\frac{4 \leq x \leq 12}{with M = 15; \gamma_0 = 2, \gamma_k = 0 for k = 1(1)15}$

Third iteration: $\alpha_{14} = 8, \alpha_{15} = 9;
\beta_{14} = 0.39212965E-08, \beta_{15} = -0.89186991E-09$

<table>
<thead>
<tr>
<th>k</th>
<th>$c_1 \alpha_k$</th>
<th>$c_2 \beta_k$</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-0.23083059E-07</td>
<td>0.45513355E 00</td>
<td>0.24551335E 01</td>
</tr>
<tr>
<td>1</td>
<td>0.10523831E-09</td>
<td>0.72169965E-03</td>
<td>0.72171016E-03</td>
</tr>
<tr>
<td>2</td>
<td>0.39212981E-08</td>
<td>0.37946336E-02</td>
<td>0.37946336E-02</td>
</tr>
<tr>
<td>3</td>
<td>-0.13066984E-02</td>
<td>-0.13066984E-02</td>
<td>-0.13066984E-02</td>
</tr>
<tr>
<td>4</td>
<td>0.13810835E-02</td>
<td>0.13810835E-02</td>
<td>0.13810835E-02</td>
</tr>
<tr>
<td>5</td>
<td>-0.58502296E-03</td>
<td>-0.58502296E-03</td>
<td>-0.58502296E-03</td>
</tr>
<tr>
<td>6</td>
<td>0.17492994E-03</td>
<td>0.17492994E-03</td>
<td>0.17492994E-03</td>
</tr>
<tr>
<td>7</td>
<td>-0.40472814E-04</td>
<td>-0.40472814E-04</td>
<td>-0.40472814E-04</td>
</tr>
<tr>
<td>8</td>
<td>0.72171017E-05</td>
<td>0.72171017E-05</td>
<td>0.72171017E-05</td>
</tr>
<tr>
<td>9</td>
<td>-0.86127752E-02</td>
<td>-0.86127752E-02</td>
<td>-0.86127752E-02</td>
</tr>
<tr>
<td>10</td>
<td>0.37946337E-07</td>
<td>0.37946337E-07</td>
<td>0.37946337E-07</td>
</tr>
<tr>
<td>11</td>
<td>-0.14417516E-02</td>
<td>-0.14417516E-02</td>
<td>-0.14417516E-02</td>
</tr>
<tr>
<td>12</td>
<td>0.39212966E-08</td>
<td>0.39212966E-08</td>
<td>0.39212966E-08</td>
</tr>
<tr>
<td>13</td>
<td>-0.89186990E-09</td>
<td>-0.89186990E-09</td>
<td>-0.89186990E-09</td>
</tr>
<tr>
<td>14</td>
<td>0.12118739E-17</td>
<td>0.12118739E-17</td>
<td>0.12118739E-17</td>
</tr>
<tr>
<td>15</td>
<td>0.10906865E-16</td>
<td>0.10906865E-16</td>
<td>0.10906865E-16</td>
</tr>
</tbody>
</table>

$\alpha_1 = 0.12118739E-17$  
$c_1 = 0.12118739E-17$
\[ c_2 = 0.100000000E \ 01 \]
\[ c_1 R(\alpha) + c_2 R(\beta) - 0.43820800E \ 00 = 0.0 \]
\[ c_1 S(\alpha) + c_2 S(\beta) - 0.64000000E \ 00 = 0.0 \]

It is worth noting that the coefficient matrix of system (21,22,23) yields an upper triangular matrix of order \( M \) after the deletion of the first two rows and the last two columns. Consequently, the procedure of this section is applicable to any linear system having this property. As a matter of fact, the same procedure can be generalized to solve linear systems having coefficient matrices of order \( N \), the deletion of whose first \( r \) \((r < N) \) rows and last \( r \) columns yields upper triangular matrices of order \( N - r \).

**The Function** \((1/x)[Ei(x) - \log|x| - \gamma] \)

Let
\[ f(x) = (1/x)[Ei(x) - \log|x| - \gamma], \quad g(x) = e^x, \quad |x| \leq b \quad (5.29) \]

These functions, with the change of variable \( x = bt \), simultaneously satisfy the differential equations
\[ bt^2 \phi' (t) + bt \phi(t) - \psi(t) = -1 \quad (5.30) \]
\[ \psi'(t) - b \psi(t) = 0, \quad -1 \leq t \leq 1 \quad (5.31) \]

Conversely,\(^2\) any solution of equations 30 and 31 is equal to the functions given by equations 29 for the change of variable \( x = bt \). Therefore, boundary conditions need not be imposed for the solution of the differential equations.

A procedure similar to that of the previous section gives the coupled infinite recurrence relations
\[ b A_1 + b A_3 - B_0 + B_2 = -2 \quad (5.32) \]
\[ \{ \begin{array}{l}
kb A_{k-1} + 2(k+1)b A_{k+1} + (k+2)b A_{k+3} - 2B_{k} + 2B_{k+2} = 0 \\
b B_{k-1} - 2kB_k - b B_{k+1} = 0, \quad k = 1, 2, \ldots 
\end{array} \quad (5.33) \]

where \( A_k \) and \( B_k \) are the Chebyshev coefficients of \( \phi(t) \) and \( \psi(t) \), respectively.

Consider first the subsystem 33. If \( A_k = \alpha_k \) and \( B_k = \beta_k \) are a simultaneous solution of the system, which is homogeneous, then
\[ \begin{align*}
A_k &= c \alpha_k \\
B_k &= c \beta_k
\end{align*} \quad (5.34) \]

\(^2\)The general solution of the differential equations has the form
\[ \phi(t) = (c_1/t) + [Ei(bt) - \log|bt| - \gamma]/bt \]
\[ \psi(t) = c_2 e^{bt} \]

where the first and second terms of \( \phi(t) \) are, respectively, the complementary solution and a particular integral of equation 30. The requirement that \( \phi(t) \) is bounded makes the constant \( c_1 = 0 \). The fact that \( \psi(0) = 1 \) is implicit in equation 30.
are also a solution for an arbitrary constant \( c \). Thus based on considerations analogous to the solution of equations 21, 22, and 23, one can initiate an approximate solution of equations 32 and 33 by setting

\[
\begin{align*}
\alpha_M &= 0, & \alpha_k &= 0 & \text{for } k \geq M + 1 \\
\beta_M &= 1, & \beta_k &= 0 & \text{for } k \geq M + 1
\end{align*}
\]  

(5.35)

and then determining \( \alpha_k \) and \( \beta_k \) \((k = M - 1, M - 2, \ldots, 0)\) by backward recurrence by means of equation 33. The arbitrary constant \( c \) is determined by substituting 34 into 32.

**The Function** \( x e^{-x} Ei(x) \) **on the Infinite Interval**

Let

\[
f(x) = x e^{-x} Ei(x), \quad -\infty < x \leq b < 0, \quad \text{or } 0 < b \leq x < \infty
\]  

(5.36)

By making the change of variables,

\[
x = 2b/(t + 1)
\]  

(5.37)

we can easily demonstrate that

\[
f(x) = f[2b/(t + 1)] = \phi(t)
\]  

(5.38)

satisfies the differential equation

\[
(t + 1)^2 \phi'(t) + (t + 1 - 2b)\phi(t) = -2b
\]  

(5.39)

with

\[
\phi(1) = be^{-b} Ei(b)
\]  

(5.40)

An infinite system of equations involving the Chebyshev coefficients \( A_k \) of \( \phi(t) \) is deducible from equations 39 and 40 by the same procedure as applied to equations 13 and 14 to obtain the infinite system 20; it is given as follows.

\[
\sum_{k=0}^{\infty} A_k = \phi(1) = be^{-b} Ei(b)
\]  

(5.41)

\[
(1 - 2b)A_0 + 3A_1 + (3 + 2b)A_2 + A_3 = -4b
\]  

(5.42)

\[
kA_{k-1} + 2[(2k + 1) - 2b]A_k + 6(k + 1)A_{k+1} + 2(2k + 3 + 2b)A_{k+2}
\]

\[
+ (k + 2)A_{k+3} = 0, \quad k = 1, 2, \ldots
\]  

(5.43)

As in the case of equations 21, 22 and 23, the solution of 41, 42 and 43 can be assumed to be

\[
A_k = c_1 \alpha_k + c_2 \beta_k
\]  

(5.44)

with \( A_k \) vanishing for a \( k \geq M \). Thus, we can set, say

\[
\begin{align*}
\alpha_{M-1} &= 0, & \alpha_M &= 1 \\
\beta_{M-1} &= 1, & \beta_M &= 0
\end{align*}
\]  

(5.45)
and determine the trial solutions $\alpha_k$ and $\beta_k$ ($k=\text{M-1, M-2, \ldots, 0}$) by means of equation 43 by backward recurrence. The required solution of equations 41, 42, and 43 is then determined by substituting equation 44 in equations 41 and 42 and solving the resulting equations for $c_1$ and $c_2$.

Loss of accuracy in the computation of $A_k$ can also occur here, as in the solution of equations 21, 22, and 23, if the trial solutions are not sufficiently independent. The process used to improve the accuracy of $A_k$ of the system 21, 22, and 23 can also be applied here.

For efficiency in computation, it is worth noting that for $b < 0$ ($-\infty < x \leq b < 0$) the boundary condition 40 is not required for the solution of equation 39 and 40. This follows from the fact that any solution\(^3\) of the differential equation 39 is equal to $xe^{-x}Ei(x) \ (x = 2b/(t + 1))$. Hence the $A_k$ of $xe^{-x}Ei(x)$ for $-\infty < x \leq b < 0$ can be obtained without the use of equation 39 and can be assumed to have the form

$$A_k = c\alpha_k, \quad (k = 0, 1, \ldots, M)$$  \hspace{1cm} (5.46)

The M+1 values of $\alpha_k$ can be generated by setting $\alpha_M = 1$ and computing $\alpha_k$ ($k=0,1,\ldots,\text{M-1}$) by means of equation 43 by backward recurrence. The substitution of equation 46 into 42 then enables one to determine $c$ from the resulting equation.

**Remarks on Convergence and Accuracy**

The Chebyshev coefficients of table 3 were computed on the IBM 7094 with 50-digit normalized floating-point arithmetic. In order to assure that the sequence of approximate solutions (see Discussion) converged to the limiting solution of the differential equation in question, a trial M was incremented by 4 until the approximate Chebyshev coefficients showed no change greater than or equal to $0.5 \times 10^{-35}$. Hence the maximum error is bounded by

$$0.5(M + 1) \times 10^{-35} + \sum_{M+1}^{\infty} |A_k|$$  \hspace{1cm} (5.47)

where the first term is the maximum error of the M+1 approximate Chebyshev coefficients, and the sum is the maximum error of the truncated Chebyshev series of M+1 terms. If the Chebyshev series is rapidly convergent, the maximum error of the approximate Chebyshev series should be of the order of $10^{-30}$. The coefficients of table 3 have been rounded to 30 digits, and higher terms for $k > N$ giving the maximum residual

$$\sum_{k=N+1}^{M} |A_k| < 0.5 \times 10^{-30}$$  \hspace{1cm} (5.48)

have been dropped. This should allow for evaluation of the relevant function that is accurate to 30 decimal places. Since the range of values of each function is bounded between $2/5$ and

\[^3\]The general solution of the differential equation 39. Since equation 39 has no bounded complementary solution for $-\infty < x \leq b < 0$, every solution of it is equal to the particular integral $xe^{-x}Ei(x)$. On the other hand, a solution of equation 39 for $0 < x \leq b < \infty$ would, in general, involve the complementary function. Hence, boundary condition 40 is required to guarantee that the solution of equation 39 is equal to $xe^{-x}Ei(x)$.\]
The evaluated function should be good to 30 significant digits. Taylor series evaluation also checks with that of the function values of table 4 (computed with 30-digit floating-point arithmetic using the coefficients of table 3) for at least 28-1/2 significant digits. Evaluation of $E_i(x)$ using the coefficients of table 3 also checked with Murnaghan and Wrench for 28-1/2 significant figures.

**Table 3: Chebyshev Coefficients**

$$xe^{-x}Ei(x) = \sum_{k=0}^{40} A_k T_k(t), \quad t = (-20/x) - 1, \quad (-\infty < x \leq -10)$$

<table>
<thead>
<tr>
<th>$k$</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.1912173225 8605534539 1519326510E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.4208355052 868483755 0974986689E 01</td>
</tr>
<tr>
<td>2</td>
<td>0.1722819627 284326783 7118158735E 02</td>
</tr>
<tr>
<td>3</td>
<td>-0.9915782173 4445636455 9842322973E 04</td>
</tr>
<tr>
<td>4</td>
<td>0.717093168 0227750526 590665592E 05</td>
</tr>
<tr>
<td>5</td>
<td>-0.6152733145 0951269968 284232973E 06</td>
</tr>
<tr>
<td>6</td>
<td>0.6024857106 527583129 399701610E 07</td>
</tr>
<tr>
<td>7</td>
<td>-0.6573848845 2883048229 5894189637E 08</td>
</tr>
<tr>
<td>8</td>
<td>0.7853167541 8323998199 4810079871E 09</td>
</tr>
<tr>
<td>9</td>
<td>-0.1013730288 038789855 4202774257E 09</td>
</tr>
<tr>
<td>10</td>
<td>0.1399770413 2267686027 872348623E 10</td>
</tr>
<tr>
<td>11</td>
<td>-0.2051008376 7838189961 8962318711E 11</td>
</tr>
<tr>
<td>12</td>
<td>0.3168388726 0024778181 4907958518E 12</td>
</tr>
<tr>
<td>13</td>
<td>-0.5132760082 8391806541 5984751899E 13</td>
</tr>
<tr>
<td>14</td>
<td>0.8680933040 7665493418 7433687383E 14</td>
</tr>
<tr>
<td>15</td>
<td>-0.1527015040 9510389949 8572355351E 14</td>
</tr>
<tr>
<td>16</td>
<td>0.2784686251 6493573965 0105251435E 15</td>
</tr>
<tr>
<td>17</td>
<td>-0.5249890437 4217696960 8472933696E 16</td>
</tr>
<tr>
<td>18</td>
<td>0.1020717991 2485612924 7455787226E 16</td>
</tr>
<tr>
<td>19</td>
<td>-0.2042264679 8997184130 8462421876E 17</td>
</tr>
<tr>
<td>20</td>
<td>0.4197064172 7264847440 8827228562E 18</td>
</tr>
<tr>
<td>21</td>
<td>-0.8844508176 1728105081 6483737536E 19</td>
</tr>
<tr>
<td>22</td>
<td>0.1908272629 5947174199 5060168262E 19</td>
</tr>
<tr>
<td>23</td>
<td>-0.4290746222 9351995033 6450865676E 20</td>
</tr>
<tr>
<td>24</td>
<td>0.9483904058 1983732674 1500214512E 21</td>
</tr>
<tr>
<td>25</td>
<td>-0.2179467860 1366743199 4032574014E 21</td>
</tr>
<tr>
<td>26</td>
<td>0.5103936869 0714599499 3452562741E 22</td>
</tr>
<tr>
<td>27</td>
<td>-0.1216883113 3344150908 9746779693E 22</td>
</tr>
<tr>
<td>28</td>
<td>0.2951289166 4447851929 4773751144E 23</td>
</tr>
<tr>
<td>29</td>
<td>-0.7275353763 7728468971 4438950920E 24</td>
</tr>
<tr>
<td>30</td>
<td>0.1821639048 6230739612 1667115976E 24</td>
</tr>
<tr>
<td>31</td>
<td>-0.4629629963 1633171661 2753482604E 25</td>
</tr>
<tr>
<td>32</td>
<td>0.1193539970 9715779152 3052371292E 25</td>
</tr>
<tr>
<td>33</td>
<td>-0.3119493285 2201424493 1062147473E 26</td>
</tr>
<tr>
<td>34</td>
<td>0.8261419743 5334664228 4170028518E 27</td>
</tr>
<tr>
<td>35</td>
<td>-0.2215803733 6609829830 2591176797E 27</td>
</tr>
<tr>
<td>36</td>
<td>0.6016031671 6542638904 5303124429E 28</td>
</tr>
<tr>
<td>37</td>
<td>-0.1652725098 3821265964 9744302314E 28</td>
</tr>
<tr>
<td>38</td>
<td>0.4592230558 7730270279 5906371766E 29</td>
</tr>
<tr>
<td>39</td>
<td>-0.1290062767 2132638473 7453212670E 29</td>
</tr>
<tr>
<td>40</td>
<td>0.3662718481 0320025908 1177078922E 30</td>
</tr>
</tbody>
</table>
\begin{align*}
t_{06} &= t_{05} - 0.615273145096269627956791331E-06 \cdot R \cos(5.0 \cdot R t) \\
t_{07} &= t_{06} + 0.60248571065627583129399701610E-07 \cdot R \cos(6.0 \cdot R t) \\
t_{08} &= t_{07} - 0.6573848845288000838798554202774257E-08 \cdot R \cos(7.0 \cdot R t) \\
t_{09} &= t_{08} + 0.785316754132267660277823488623E-09 \cdot R \cos(8.0 \cdot R t) \\
t_{10} &= t_{09} - 0.10137028000387898554202774257E-09 \cdot R \cos(9.0 \cdot R t) \\
t_{11} &= t_{10} + 0.13997704132267660277823488623E-10 \cdot R \cos(10.0 \cdot R t) \\
t_{12} &= t_{11} - 0.2051008376783818966231871E-11 \cdot R \cos(11.0 \cdot R t) \\
t_{13} &= t_{12} + 0.3168388726000247781814907985818E-12 \cdot R \cos(12.0 \cdot R t) \\
t_{14} &= t_{13} - 0.51376008239180654159847518999E-13 \cdot R \cos(13.0 \cdot R t) \\
t_{15} &= t_{14} + 0.868093304076654934187433687383E-14 \cdot R \cos(14.0 \cdot R t) \\
t_{16} &= t_{15} - 0.152701504090308497198572355351E-14 \cdot R \cos(15.0 \cdot R t) \\
t_{17} &= t_{16} + 0.278468625164935739650105251453E-15 \cdot R \cos(16.0 \cdot R t) \\
t_{18} &= t_{17} - 0.524989043742176696808472933696E-16 \cdot R \cos(17.0 \cdot R t) \\
t_{19} &= t_{18} + 0.1020719912485612924745787226E-16 \cdot R \cos(18.0 \cdot R t) \\
t_{20} &= t_{19} - 0.2042246647989971841308462421876E-17 \cdot R \cos(19.0 \cdot R t) \\
t_{21} &= t_{20} + 0.419704622293519950336450865676E-20 \cdot R \cos(20.0 \cdot R t) \\
t_{22} &= t_{21} - 0.84450817617281050816483737536E-19 \cdot R \cos(21.0 \cdot R t) \\
t_{23} &= t_{22} + 0.190827265947174199950601682621E-19 \cdot R \cos(22.0 \cdot R t) \\
t_{24} &= t_{23} - 0.420974622293519950336450865676E-20 \cdot R \cos(23.0 \cdot R t) \\
t_{25} &= t_{24} + 0.9483904058198373276741500214512E-21 \cdot R \cos(24.0 \cdot R t) \\
t_{26} &= t_{25} - 0.217946786013667431994032574014E-21 \cdot R \cos(25.0 \cdot R t) \\
t_{27} &= t_{26} + 0.51039368907145094993452562741E-22 \cdot R \cos(26.0 \cdot R t) \\
t_{28} &= t_{27} - 0.1216863113333441509089746779693E-22 \cdot R \cos(27.0 \cdot R t) \\
t_{29} &= t_{28} + 0.29512891664477851929477375144E-23 \cdot R \cos(28.0 \cdot R t) \\
t_{30} &= t_{29} - 0.7275353767728468914438950920E-24 \cdot R \cos(29.0 \cdot R t) \\
t_{31} &= t_{30} + 0.182163304862037396121667115976E-24 \cdot R \cos(30.0 \cdot R t) \\
t_{32} &= t_{31} - 0.462962896316331711612753482064E-25 \cdot R \cos(31.0 \cdot R t) \\
t_{33} &= t_{32} + 0.119353979097157791523052371292E-25 \cdot R \cos(32.0 \cdot R t) \\
t_{34} &= t_{33} - 0.311940328522014244931062147473E-26 \cdot R \cos(33.0 \cdot R t) \\
t_{35} &= t_{34} + 0.82641973453436642284170028518E-27 \cdot R \cos(34.0 \cdot R t) \\
t_{36} &= t_{35} - 0.22158037366098298302591177697E-27 \cdot R \cos(35.0 \cdot R t) \\
t_{37} &= t_{36} + 0.60163167154263849045333124429E-28 \cdot R \cos(36.0 \cdot R t) \\
t_{38} &= t_{37} - 0.16527508382126596947443023143E-28 \cdot R \cos(37.0 \cdot R t) \\
t_{39} &= t_{38} + 0.45922303887730272975636377166E-29 \cdot R \cos(38.0 \cdot R t) \\
t_{40} &= t_{39} - 0.1290062767123638473453212670E-29 \cdot R \cos(39.0 \cdot R t) \\
t_{41} &= t_{40} + 0.366271848103202059681177078922E-30 \cdot R \cos(40.0 \cdot R t) \\
\end{align*}

Table 3: Chebyshev Coefficients - Continued (b)

\[ xe^{-x} E_i(x) = \sum_{k=0}^{40} \,'A_k T_k(t), \quad t = (x+7)/3, \quad (-10 \leq x \leq -4) \]
\[
\begin{array}{cccc}
\text{k} & A_k & \text{Ei2}(y:OPR):OPR == \\
0 & 0.175756496 061293738 8762834691E 011 & 21 & -0.432277683 383850564 5764394579E-15 \\
1 & -0.435541517 7361661170 5001876964E-01 & 22 & -0.906301479 660172551 4905603536E-16 \\
2 & -0.7979507139 5584254013 3217027942E-02 & 23 & -0.190466997 5816613974 4019633425E-16 \\
3 & -0.1484372372 3037121385 9070210001E-02 & 24 & -0.4011792336 3502786634 6744272570E-17 \\
4 & -0.2800301984 3775145748 6203954948E-03 & 25 & -0.8467772130 0168322313 4166334685E-18 \\
5 & -0.534864512 8657932203 9177361553E-04 & 26 & -0.1790842733 6589666555 5826492204E-18 \\
6 & -0.103267243 573548661 0233266460E-04 & 27 & -0.3794490638 1714782440 1106171666E-19 \\
7 & -0.2014083313 0055368773 2226198639E-05 & 28 & -0.8053999236 7982798526 0999654058E-20 \\
8 & -0.3961758434 2738664582 2338443500E-06 & 29 & -0.1712339011 2362012974 3228671244E-20 \\
9 & -0.7853872767 966313606 7607656096E-07 & 30 & -0.3646274058 7749686208 6576568216E-21 \\
10 & -0.1567295981 074698262 4616270279E-07 & 31 & -0.7775969638 8939479435 3098157647E-22 \\
11 & -0.3150059985 736998825 0007328515E-08 & 32 & -0.1660628498 4484020566 5231950966E-22 \\
12 & -0.6365096822 524203704 0380263972E-09 & 33 & -0.3551178625 7882509300 5927145352E-23 \\
13 & -0.1298881133 2805631835 6593121259E-09 & 34 & -0.7603722685 9413580929 5734653294E-24 \\
14 & -0.2638690999 6592557613 2149942808E-10 & 35 & -0.1630007413 2584900288 9638374755E-24 \\
15 & -0.5408958287 405687349 1922207896E-11 & 36 & -0.3498575202 7268223250 7538497255E-25 \\
16 & -0.1113222784 6010898999 7676692708E-11 & 37 & -0.7517197627 8900198246 0645141543E-26 \\
17 & -0.2299624726 0744624618 4338864145E-12 & 38 & -0.1616877480 0527227629 8777137918E-26 \\
18 & -0.4766832389 4951902622 3913482091E-13 & 39 & -0.3481270085 7247596174 8202271565E-27 \\
19 & -0.9911756747 3352709450 6246643717E-14 & 40 & -0.7502707769 5024657041 0642233720E-28 \\
20 & -0.2067103580 4957072400 090080521E-14 & 41 & -0.1618454364 4959102680 7612330206E-28 \\
21 & & 42 & -0.3493667717 7051616774 9482836452E-29 \\
& & 43 & -0.7551036906 1261678855 6037026797E-30 \\
\end{array}
\]
\[ t_{20} := t_{19} - 0.9917567433527094506246643371E-14::R \cos(19.0::R \cdot t) \]
\[ t_{21} := t_{20} - 0.20671038049570724000900805021E-14::R \cos(20.0::R \cdot t) \]
\[ t_{22} := t_{21} - 0.43227767338338505645764394579E-15::R \cos(21.0::R \cdot t) \]
\[ t_{23} := t_{22} - 0.9630147966501725514905603356E-16::R \cos(22.0::R \cdot t) \]
\[ t_{24} := t_{23} - 0.19046699758166139744015963342E-16::R \cos(23.0::R \cdot t) \]
\[ t_{25} := t_{24} - 0.401179232635027866346744227520E-17::R \cos(24.0::R \cdot t) \]
\[ t_{26} := t_{25} - 0.846777213001683223134166334685E-18::R \cos(25.0::R \cdot t) \]
\[ t_{27} := t_{26} - 0.179084273365869665555826492204E-18::R \cos(26.0::R \cdot t) \]
\[ t_{28} := t_{27} - 0.379449063817147824401106175166E-19::R \cos(27.0::R \cdot t) \]
\[ t_{29} := t_{28} - 0.80539992367982798526099654058E-20::R \cos(28.0::R \cdot t) \]
\[ t_{30} := t_{29} - 0.17123390112362012974328671244E-20::R \cos(29.0::R \cdot t) \]
\[ t_{31} := t_{30} - 0.346274058774968620865766562816E-21::R \cos(30.0::R \cdot t) \]
\[ t_{32} := t_{31} - 0.777596938893947935909157647E-22::R \cos(31.0::R \cdot t) \]
\[ t_{33} := t_{32} - 0.16062849844840205662531950966E-22::R \cos(32.0::R \cdot t) \]
\[ t_{34} := t_{33} - 0.35117862578825093005927145352E-23::R \cos(33.0::R \cdot t) \]
\[ t_{35} := t_{34} - 0.760372268594139509295734653294E-24::R \cos(34.0::R \cdot t) \]
\[ t_{36} := t_{35} - 0.16300741372584900289683874755E-24::R \cos(35.0::R \cdot t) \]
\[ t_{37} := t_{36} - 0.349857520272863232507538497255E-25::R \cos(36.0::R \cdot t) \]
\[ t_{38} := t_{37} - 0.7517179627890098246064514514E-26::R \cos(37.0::R \cdot t) \]
\[ t_{39} := t_{38} - 0.1616877440065272276298777317918E-26::R \cos(38.0::R \cdot t) \]
\[ t_{40} := t_{39} - 0.348127008572475691748202271565E-27::R \cos(39.0::R \cdot t) \]
\[ t_{41} := t_{40} - 0.75027077755024654701064223720E-28::R \cos(40.0::R \cdot t) \]
\[ t_{42} := t_{41} - 0.1618453649499102680761230206E-28::R \cos(41.0::R \cdot t) \]
\[ t_{43} := t_{42} - 0.34943667717051666749482836452E-29::R \cos(42.0::R \cdot t) \]
\[ t_{44} := t_{43} - 0.755103690612616785856037026797E-30::R \cos(43.0::R \cdot t) \]
\[ t_{44} :: OPR \]

---

**Table 3: Chebyshev Coefficients - Continued (c)**

\[
[E_i - \log |x| - \gamma]/x = \sum_{k=0}^{33} \hat{A}_k T_k(t), \quad t = x/4, \quad (-4 \leq x \leq 4)
\]
<table>
<thead>
<tr>
<th>k</th>
<th>$A_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.329370103 767391293 93905231421E 01</td>
</tr>
<tr>
<td>1</td>
<td>0.1679835052 3713092156 5505796064E 01</td>
</tr>
<tr>
<td>2</td>
<td>0.7220436105 6187543596 2999679644E 00</td>
</tr>
<tr>
<td>3</td>
<td>0.2600312360 5480896171 3740181192E 00</td>
</tr>
<tr>
<td>4</td>
<td>0.8010494308 1735902239 4742889237E-01</td>
</tr>
<tr>
<td>5</td>
<td>0.2154036636 976337548 055243005E-01</td>
</tr>
<tr>
<td>6</td>
<td>0.5116207789 303312062 1968910894E-02</td>
</tr>
<tr>
<td>7</td>
<td>0.1090932861 073913560 5066199014E-02</td>
</tr>
<tr>
<td>8</td>
<td>0.2107415320 239891631 8349875226E-03</td>
</tr>
<tr>
<td>9</td>
<td>0.719904516 6518885709 5949085964E-04</td>
</tr>
<tr>
<td>10</td>
<td>0.6043491637 238785705 4767032866E-05</td>
</tr>
<tr>
<td>11</td>
<td>0.9092954274 629529754 6295416819E-06</td>
</tr>
<tr>
<td>12</td>
<td>0.1273805160 6592475886 5567194969E-06</td>
</tr>
<tr>
<td>13</td>
<td>0.1669157848 1909990739 0986138141E-07</td>
</tr>
<tr>
<td>14</td>
<td>0.2054417026 4010479254 7612484511E-08</td>
</tr>
<tr>
<td>15</td>
<td>0.2385844444 4668176591 4052321417E-09</td>
</tr>
<tr>
<td>16</td>
<td>$0.2615386738$ 8854429666 9068664148E-10</td>
</tr>
<tr>
<td>17</td>
<td>$0.2721508622$ 5416706444 6550268995E-11</td>
</tr>
</tbody>
</table>

$E_{i3}(y: OPR): OPR ==$

$x: R := retract(y)$

$x = 0.0 :: R => 1$

$t: R := acos(x/4.0 :: R) :: R$

t01 := 0.3293700103767391293905231421E1 :: R * cos(0.0 :: R) / 2.0 :: R$

t02 := t01 + 0.167983505237130921565505796064E1 :: R * cos(t)$

t03 := t02 + 0.722043610561875435202299679644E0 :: R * cos(2.0 :: R * t)$

t04 := t03 + 0.26003123605480956713740181192E0 :: R * cos(3.0 :: R * t)$

t05 := t04 + 0.80104943081735902239472889237E-01 :: R * cos(4.0 :: R * t)$

t06 := t05 + 0.21074153202398916318348675226E-03 :: R * cos(5.0 :: R * t)$

t07 := t06 + 0.1090932861073913560506066199014E-02 :: R * cos(6.0 :: R * t)$

t08 := t07 + 0.518967683404351272078008019E-02 :: R * cos(7.0 :: R * t)$

t09 := t08 + 0.3560409454099706811804316222E-02 :: R * cos(8.0 :: R * t)$

t10 := t09 + 0.2361979432579384237018720394E-02 :: R * cos(9.0 :: R * t)$

t11 := t10 + 0.163402455192893174065567184969E-02 :: R * cos(10.0 :: R * t)$

t12 := t11 + 0.127380516065924758865567194969E-02 :: R * cos(11.0 :: R * t)$

t13 := t12 + 0.99225879250737105964632581302E-02 :: R * cos(12.0 :: R * t)$

t14 := t13 + 0.7306252806722103294723088018E-02 :: R * cos(13.0 :: R * t)$

t15 := t14 + 0.518967683404351272078008019E-02 :: R * cos(14.0 :: R * t)$

t16 := t15 + 0.3560409454099706811804316222E-02 :: R * cos(15.0 :: R * t)$

t17 := t16 + 0.2361979432579384237018720394E-02 :: R * cos(16.0 :: R * t)$

t18 := t17 + 0.163402455192893174065567184969E-02 :: R * cos(17.0 :: R * t)$

t19 := t18 + 0.127380516065924758865567194969E-02 :: R * cos(18.0 :: R * t)$

t20 := t19 + 0.99225879250737105964632581302E-02 :: R * cos(19.0 :: R * t)$

t21 := t20 + 0.7306252806722103294723088018E-02 :: R * cos(20.0 :: R * t)$
Table 3: Chebyshev Coefficients - Continued (d)

\[ xe^{-x}Ei(x) = \sum_{k=0}^{49} A_k T_k(t), \quad t = (x - 8)/4, \quad (4 \leq x \leq 12) \]
<table>
<thead>
<tr>
<th>k</th>
<th>( A_k )</th>
<th>( k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2455133538 7812952867 3420457043E 01</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>-0.1624383791 3037652439 6002276856E 00</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>0.4495753080 9357264148 0785417193E-01</td>
<td>26</td>
</tr>
<tr>
<td>3</td>
<td>-0.6741578679 9892998884 8718835050E-02</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>-0.1306697142 8032948205 1599348937E-02</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>0.1381083146 0007257602 0202089820E-02</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>-0.5850228790 1596759796 8718835050E-03</td>
<td>30</td>
</tr>
<tr>
<td>7</td>
<td>0.1749299341 0789197003 8718835050E-04</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>-0.4047281499 0529303552 2869333800E-05</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>0.7217012412 1709975003 5752600049E-06</td>
<td>33</td>
</tr>
<tr>
<td>10</td>
<td>-0.8612767970 1986775241 4815450193E-07</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>-0.2514475296 5322559777 9084739054E-08</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>0.3794747138 2041951081 4074505574E-09</td>
<td>36</td>
</tr>
<tr>
<td>13</td>
<td>-0.1421179699 5211980616 265640172E-09</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>0.3935049295 9761013108 190848042E-09</td>
<td>38</td>
</tr>
<tr>
<td>15</td>
<td>-0.9284689401 0633175304 7289201303E-10</td>
<td>39</td>
</tr>
<tr>
<td>16</td>
<td>0.2031789568 0065461336 6090995698E-10</td>
<td>40</td>
</tr>
<tr>
<td>17</td>
<td>-0.4292498504 9928698041 265640172E-11</td>
<td>41</td>
</tr>
<tr>
<td>18</td>
<td>0.899253437 2931901982 5435824585E-12</td>
<td>42</td>
</tr>
<tr>
<td>19</td>
<td>-0.2019654670 8242638335 4948543451E-13</td>
<td>43</td>
</tr>
<tr>
<td>20</td>
<td>0.4663128735 2730486582 6001239238E-14</td>
<td>44</td>
</tr>
<tr>
<td>21</td>
<td>-0.1185959588 9190288794 6724005478E-15</td>
<td>45</td>
</tr>
<tr>
<td>22</td>
<td>0.302090590 5567131073 1137614875E-16</td>
<td>46</td>
</tr>
<tr>
<td>23</td>
<td>-0.7701650548 1663660609 8827057102E-30</td>
<td>47</td>
</tr>
</tbody>
</table>

---

**package DFSFUN DoubleFloatSpecialFunctions**

```lisp
| package DFSFUN DoubleFloatSpecialFunctions |
|---|---|
| Ei4(y:OPR):OPR == | x:R:=retract(y)
| x:R:=acos((x-8.0::R)/4.0::R)::R |
| t01:= 0.245513353878129528673420457043E1::R*cos(0.0::R)/2.0::R |
| t02::t01-0.162438379130376524396002278856E0::R*cos(t) |
| t03::t02-0.44957530809357264148078541793E0::R*cos( 2.0::R*t) |
| t04::t03-0.67415786799822998848718335050E-02::R*cos( 3.0::R*t) |
| t05::t04-0.130669714280329428051599341387E0::R*cos( 4.0::R*t) |
| t06::t05+0.379474713820419510814074505574E-01::R*cos( 5.0::R*t) |
| t07::t06-0.86127679701986775241481540193E-02::R*cos( 6.0::R*t) |
| t08::t07+0.203178956800654613366090995698E-03::R*cos( 7.0::R*t) |
| t09::t08-0.42924985049928698041265640172E-03::R*cos( 8.0::R*t) |
| t10::t09+0.89925343729319019825435824585E-13::R*cos( 9.0::R*t) |
| t11::t10-0.174929934107891970038740976432E0::R*cos( 10.0::R*t) |
| t12::t11+0.721701241217099750035752600049E-05::R*cos( 11.0::R*t) |
| t13::t12-0.39474713820419510814074505574E-07::R*cos( 12.0::R*t) |
| t14::t13+0.142211796952119806160265604172E-07::R*cos( 13.0::R*t) |
| t15::t14-0.393504929597610131087190848042E-08::R*cos( 14.0::R*t) |
| t16::t15-0.928469840106331753047289210353E-09::R*cos( 15.0::R*t) |
```

---
t_{17}:=t_{16}+0.2031789680065461336609095698E-09::R*cos(16.0::R*t)
\n\nt_{18}:=t_{17}-0.429249850499236831427918026902E-10::R*cos(17.0::R*t)
\n\nt_{19}:=t_{18}+0.899264717778123935268001544182E-11::R*cos(18.0::R*t)
\n\nt_{20}:=t_{19}-0.19008691184121097524396635722E-11::R*cos(19.0::R*t)
\n\nt_{21}:=t_{20}+0.40921989122232373834526121178338E-12::R*cos(20.0::R*t)
\n\nt_{22}:=t_{21}-0.89925343729319019825435824585E-13::R*cos(21.0::R*t)
\n\nt_{23}:=t_{22}+0.2019654670824263833549484543451E-13::R*cos(22.0::R*t)
\n\nt_{24}:=t_{23}-0.461293026138308207194950531726E-14::R*cos(23.0::R*t)
\n\nt_{25}:=t_{24}+0.10690230729386956885875640919E-14::R*cos(24.0::R*t)
\n\nt_{26}:=t_{25}-0.250703007050707295692572254042E-15::R*cos(25.0::R*t)
\n\nt_{27}:=t_{26}+0.59373225037915516070673736509E-16::R*cos(26.0::R*t)
\n\nt_{28}:=t_{27}-0.141773458243766252344732005648E-16::R*cos(27.0::R*t)
\n\nt_{29}:=t_{28}+0.34092037543608093426806402093E-17::R*cos(28.0::R*t)
\n\nt_{30}:=t_{29}-0.824829026950549379288702529656E-18::R*cos(29.0::R*t)
\n\nt_{31}:=t_{30}+0.200636971262144231398824095937E-18::R*cos(30.0::R*t)
\n\nt_{32}:=t_{31}-0.490385166796742224403498152027E-19::R*cos(31.0::R*t)
\n\nt_{33}:=t_{32}+0.1203734482348332171666409324E-19::R*cos(32.0::R*t)
\n\nt_{34}:=t_{33}-0.29662824471436825381453572575E-20::R*cos(33.0::R*t)
\n\nt_{35}:=t_{34}+0.73355123842880759924212328436E-21::R*cos(34.0::R*t)
\n\nt_{36}:=t_{35}-0.1819924129065112734426385604E-21::R*cos(35.0::R*t)
\n\nt_{37}:=t_{36}+0.45282937429576060217359526404E-22::R*cos(36.0::R*t)
\n\nt_{38}:=t_{37}-0.1129880043750506091338906717853E-22::R*cos(37.0::R*t)
\n\nt_{39}:=t_{38}+0.282668125129011666923764408445E-23::R*cos(38.0::R*t)
\n\nt_{40}:=t_{39}-0.70871797716904961666732640699E-24::R*cos(39.0::R*t)
\n\nt_{41}:=t_{40}+0.178110452401870951534401503034E-24::R*cos(40.0::R*t)
\n\nt_{42}:=t_{41}-0.44850407661896357312006142358E-25::R*cos(41.0::R*t)
\n\nt_{43}:=t_{42}+0.11315402925754766224505309840E-25::R*cos(42.0::R*t)
\n\nt_{44}:=t_{43}-0.28599578997793216379041326136E-26::R*cos(43.0::R*t)
\n\nt_{45}:=t_{44}+0.724077580692267361758172726753E-27::R*cos(44.0::R*t)
\n\nt_{46}:=t_{45}-0.183613223412577898050666710105E-27::R*cos(45.0::R*t)
\n\nt_{47}:=t_{46}+0.466312873522730486582600122073E-28::R*cos(46.0::R*t)
\n\nt_{48}:=t_{47}-0.118595985891902887946724005478E-28::R*cos(47.0::R*t)
\n\nt_{49}:=t_{48}+0.30202905095671310731137614875E-29::R*cos(48.0::R*t)
\n\nt_{50}:=t_{49}-0.770165054816636606098827057102E-30::R*cos(49.0::R*t)
\n\nt_{50}::OPR

Table 3: Chebyshev Coefficients - Continued (e)

\[ xe^{-x}Ei(x) = \sum_{k=0}^{47} A_k T_k(t), \quad t = (x - 22)/10, \quad (12 \leq x \leq 32) \]
<table>
<thead>
<tr>
<th>( k )</th>
<th>( A_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2117028640 4369866832 9789991614E 01</td>
</tr>
<tr>
<td>1</td>
<td>-0.3204237273 7548579499 0618303177E-01</td>
</tr>
<tr>
<td>2</td>
<td>0.8891732077 3531683589 0182400335E-02</td>
</tr>
<tr>
<td>3</td>
<td>-0.2507952805 1892993708 835242063E-02</td>
</tr>
<tr>
<td>4</td>
<td>0.7202789465 9598754887 5760002487E-03</td>
</tr>
<tr>
<td>5</td>
<td>0.2103490058 5011305432 351441256E-03</td>
</tr>
<tr>
<td>6</td>
<td>0.6205732318 2769321658 8857730842E-04</td>
</tr>
<tr>
<td>7</td>
<td>-0.1825667649 8167026544 9155689733E-04</td>
</tr>
<tr>
<td>8</td>
<td>0.5270515755 8293637580 778296811E-05</td>
</tr>
<tr>
<td>9</td>
<td>-0.1459665474 6199457532 3066719367E-05</td>
</tr>
<tr>
<td>10</td>
<td>0.3781719973 5896376198 048193981E-06</td>
</tr>
<tr>
<td>11</td>
<td>-0.8842581280 4807192007 7971589012E-07</td>
</tr>
<tr>
<td>12</td>
<td>0.1741749198 5383963137 7350309156E-07</td>
</tr>
<tr>
<td>13</td>
<td>-0.2313517747 0436906350 6474480152E-08</td>
</tr>
<tr>
<td>14</td>
<td>0.5898114347 0713196171 563478524E-09</td>
</tr>
<tr>
<td>15</td>
<td>-0.9099707635 9564920464 3554720718E-15</td>
</tr>
<tr>
<td>16</td>
<td>0.1040732032 8669538658 5405697541E-15</td>
</tr>
<tr>
<td>17</td>
<td>-0.3977098842 5639477336 5935494417E-17</td>
</tr>
<tr>
<td>18</td>
<td>0.1580329091 284795713 6759888420E-17</td>
</tr>
<tr>
<td>19</td>
<td>-0.4684291758 808273064 8439752957E-18</td>
</tr>
<tr>
<td>20</td>
<td>0.119516854 5919089370 7533478542E-18</td>
</tr>
<tr>
<td>21</td>
<td>-0.2832594749 8418651767 9349743711E-19</td>
</tr>
<tr>
<td>22</td>
<td>0.6205732318 2769321658 8857730842E-04</td>
</tr>
<tr>
<td>23</td>
<td>-0.1825667649 8167026544 9155689733E-04</td>
</tr>
<tr>
<td>24</td>
<td>0.5270515755 8293637580 778296811E-05</td>
</tr>
<tr>
<td>25</td>
<td>-0.1459665474 6199457532 3066719367E-05</td>
</tr>
<tr>
<td>26</td>
<td>0.3781719973 5896376198 048193981E-06</td>
</tr>
<tr>
<td>27</td>
<td>-0.8842581280 4807192007 7971589012E-07</td>
</tr>
<tr>
<td>28</td>
<td>0.1741749198 5383963137 7350309156E-07</td>
</tr>
<tr>
<td>29</td>
<td>-0.2313517747 0436906350 6474480152E-08</td>
</tr>
<tr>
<td>30</td>
<td>0.5898114347 0713196171 563478524E-09</td>
</tr>
</tbody>
</table>

---

package DFSFUN DoubleFloatSpecialFunctions

\[
\text{Ei5}(y:\text{OPR}):\text{OPR} == \\
x:\text{R}:=\text{retract}(y) \\
t:\text{R}:=\text{acos}((x-22.0::\text{R})/10.0::\text{R})::\text{R} \\
t01::= 0.211702864043698668329789991614E1::\text{R} \times \text{cos}(0.0::\text{R})::\text{R} / 2.0::\text{R} \\
t02::= \text{t01}-0.320423727375485794999618303177E-01::\text{R} \times \text{cos}(t) \\
t03::= \text{t02}+0.889173207735316835890182400335E-02::\text{R} \times \text{cos}(2.0::\text{R} \times t) \\
t04::= \text{t03}-0.2507952805189299370835242063E-02::\text{R} \times \text{cos}(3.0::\text{R} \times t) \\
t05::= \text{t04}+0.720278946595987548875760902987E-03::\text{R} \times \text{cos}(4.0::\text{R} \times t) \\
t06::= \text{t05}-0.210349005850113054323531441256E-03::\text{R} \times \text{cos}(5.0::\text{R} \times t) \\
t07::= \text{t06}+0.6205732318276932165885730842E-04::\text{R} \times \text{cos}(6.0::\text{R} \times t) \\
t08::= \text{t07}-0.182566749167026549155689733E-04::\text{R} \times \text{cos}(7.0::\text{R} \times t) \\
t09::= \text{t08}+0.5270515758193657387788296811E-05::\text{R} \times \text{cos}(8.0::\text{R} \times t) \\
t10::= \text{t09}-0.1459665476199457532066719367E-05::\text{R} \times \text{cos}(9.0::\text{R} \times t) \\
t11::= \text{t10}+0.378171997358963617980448193981E-06::\text{R} \times \text{cos}(10.0::\text{R} \times t) \\
t12::= \text{t11}-0.884258128284071920077971589012E-07::\text{R} \times \text{cos}(11.0::\text{R} \times t) \\
t13::= \text{t12}+0.17417491985839361377350309165E-07::\text{R} \times \text{cos}(12.0::\text{R} \times t) \\
t14::= \text{t13}-0.32315774704369063506474480152E-08::\text{R} \times \text{cos}(13.0::\text{R} \times t) \\
t15::= \text{t14}+0.12860981910806238832104835230E-09::\text{R} \times \text{cos}(14.0::\text{R} \times t) \\
t16::= \text{t15}+0.2349966236322863707487311318926E-09::\text{R} \times \text{cos}(15.0::\text{R} \times t) \\
t17::= \text{t16}-0.110071940102762876890738963049E-09::\text{R} \times \text{cos}(16.0::\text{R} \times t) \\
\]
Table 3: Chebyshev Coefficients - Continued (f)

\[ xe^{-x} E_i(x) = \sum_{k=0}^{46} 'A_k T_k(t), \quad t = (64/x) - 1, \quad (32 \leq x < \infty) \]
<table>
<thead>
<tr>
<th>k</th>
<th>( A_k )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.2032843945 7961669908 7873844202E 01</td>
</tr>
<tr>
<td>1</td>
<td>0.1669920452 0313628514 7618434339E-01</td>
</tr>
<tr>
<td>2</td>
<td>0.2845284724 3613468074 2489985325E-03</td>
</tr>
<tr>
<td>3</td>
<td>0.7563944358 5162064894 8786693854E-05</td>
</tr>
<tr>
<td>4</td>
<td>0.2798971289 4508591575 0484318090E-06</td>
</tr>
<tr>
<td>5</td>
<td>0.1357901828 5345310695 256392593E-07</td>
</tr>
<tr>
<td>6</td>
<td>0.8343596202 0404692558 5610289412E-09</td>
</tr>
<tr>
<td>7</td>
<td>0.6709717267 6402484382 7524337306E-10</td>
</tr>
<tr>
<td>8</td>
<td>0.6007247608 811621357 6083084850E-11</td>
</tr>
<tr>
<td>9</td>
<td>0.7022876174 6797735907 5059216588E-12</td>
</tr>
<tr>
<td>10</td>
<td>0.1018302673 706736930 9667322152E-12</td>
</tr>
<tr>
<td>11</td>
<td>0.1761812903 4308800404 656741545E-13</td>
</tr>
<tr>
<td>12</td>
<td>0.3250828614 235306942 4072007647E-14</td>
</tr>
<tr>
<td>13</td>
<td>0.5071770025 5059216588 1479300685E-15</td>
</tr>
<tr>
<td>14</td>
<td>0.1665177387 0432942985 3520036957E-16</td>
</tr>
<tr>
<td>15</td>
<td>-0.3165403769 7975144007 2410018963E-16</td>
</tr>
<tr>
<td>16</td>
<td>-0.1588403763 6641415154 824313407E-16</td>
</tr>
<tr>
<td>17</td>
<td>-0.4175513256 1380195830 962455063E-17</td>
</tr>
<tr>
<td>18</td>
<td>-0.2892347749 7071418820 2868862358E-18</td>
</tr>
<tr>
<td>19</td>
<td>0.2800625903 3966080728 9978777339E-18</td>
</tr>
<tr>
<td>20</td>
<td>0.1322938639 5345310695 256392593E-07</td>
</tr>
<tr>
<td>21</td>
<td>0.1804447444 1773019598 534311191E-19</td>
</tr>
<tr>
<td>22</td>
<td>-0.7905384086 5226165620 2021080364E-20</td>
</tr>
<tr>
<td>23</td>
<td>-0.4435711366 3695734471 8167314045E-20</td>
</tr>
</tbody>
</table>

---

package DFSFUN DoubleFloatSpecialFunctions ---

\[ \text{Ei6}(y:\text{OPR}) :\text{OPR} == \]
\[ \text{infinite? } y \Rightarrow 1 \]
\[ x:R := \text{retract}(y) \]
\[ m:R := 64.0::R/x-1.0::R \]
\[ t:R := \text{acos}(m::R)::R \]
\[ t01 := 0.203284394579616699087873844202E1::R*\cos(0.0::R)::R/2.0::R \]
\[ t02 := t01*0.166992045203136285147618434339E-01::R*\cos(t) \]
\[ t03 := t02*0.284528472436134680742489985325E-03::R*\cos( 2.0::R*t) \]
\[ t04 := t03*0.756394435851620648498786693854E-05::R*\cos( 3.0::R*t) \]
\[ t05 := t04*0.279897128945085591750484318090E-06::R*\cos( 4.0::R*t) \]
\[ t06 := t05*0.135790182853453106952556392593E-07::R*\cos( 5.0::R*t) \]
\[ t07 := t06*0.834359620204046925585610289412E-09::R*\cos( 6.0::R*t) \]
\[ t08 := t07*0.63709717276402484382752437306E-10::R*\cos( 7.0::R*t) \]
\[ t09 := t08*0.6007247608186112357603804850E-11::R*\cos( 8.0::R*t) \]
\[ t10 := t09*0.7022876174679735970559216588E-12::R*\cos( 9.0::R*t) \]
\[ t11 := t10*0.10183026737036386796939667322152E-12::R*\cos(10.0::R*t) \]
\[ t12 := t11*0.17618129034308800404656741554E-13::R*\cos(11.0::R*t) \]
\[ t13 := t12*0.32508286142353606942470207647E-14::R*\cos(12.0::R*t) \]
\[ t14 := t13*0.507177002550581867881479300685E-15::R*\cos(13.0::R*t) \]
\[
t_{15} = t_{14} + 0.1665177387043292895320036957E-16 \cdot R \cos (14.0 \cdot R \cdot t)
\]
\[
t_{16} = t_{15} - 0.316675389079751440072410018963E-16 \cdot R \cos (15.0 \cdot R \cdot t)
\]
\[
t_{17} = t_{16} - 0.158840376664141515482314047E-16 \cdot R \cos (16.0 \cdot R \cdot t)
\]
\[
t_{18} = t_{17} + 0.417551325613801883089626455063E-17 \cdot R \cos (17.0 \cdot R \cdot t)
\]
\[
t_{19} = t_{18} - 0.289234774970714188202886862358E-18 \cdot R \cos (18.0 \cdot R \cdot t)
\]
\[
t_{20} = t_{19} + 0.280062590339660807289978777339E-18 \cdot R \cos (19.0 \cdot R \cdot t)
\]
\[
t_{21} = t_{20} + 0.132293863953927089140653056364E-18 \cdot R \cos (20.0 \cdot R \cdot t)
\]
\[
t_{22} = t_{21} + 0.180444444417730199583348119119E-19 \cdot R \cos (21.0 \cdot R \cdot t)
\]
\[
t_{23} = t_{22} - 0.79053840865226166562020108364E-20 \cdot R \cos (22.0 \cdot R \cdot t)
\]
\[
t_{24} = t_{23} - 0.44357136636957344718163140459E-20 \cdot R \cos (23.0 \cdot R \cdot t)
\]
\[
t_{25} = t_{24} - 0.426410399497810261760579779746E-21 \cdot R \cos (24.0 \cdot R \cdot t)
\]
\[
t_{26} = t_{25} + 0.392010176693714390725625388636E-21 \cdot R \cos (25.0 \cdot R \cdot t)
\]
\[
t_{27} = t_{26} - 0.152737805134396364472804486402E-22 \cdot R \cos (26.0 \cdot R \cdot t)
\]
\[
t_{28} = t_{27} + 0.102484952704949060786953149788E-22 \cdot R \cos (27.0 \cdot R \cdot t)
\]
\[
t_{29} = t_{28} - 0.213490787477108937948904287231E-22 \cdot R \cos (28.0 \cdot R \cdot t)
\]
\[
t_{30} = t_{29} - 0.323913947516023687614279789345E-23 \cdot R \cos (29.0 \cdot R \cdot t)
\]
\[
t_{31} = t_{30} + 0.214218376229645970296249355934E-23 \cdot R \cos (30.0 \cdot R \cdot t)
\]
\[
t_{32} = t_{31} + 0.823460941961899553169207838151E-24 \cdot R \cos (31.0 \cdot R \cdot t)
\]
\[
t_{33} = t_{32} - 0.152465282962067210811495038147E-24 \cdot R \cos (32.0 \cdot R \cdot t)
\]
\[
t_{34} = t_{33} + 0.137820828248824401290438126477E-24 \cdot R \cos (33.0 \cdot R \cdot t)
\]
\[
t_{35} = t_{34} + 0.21311120142873706791513005998E-26 \cdot R \cos (34.0 \cdot R \cdot t)
\]
\[
t_{36} = t_{35} + 0.201264965187132665859213006507E-25 \cdot R \cos (35.0 \cdot R \cdot t)
\]
\[
t_{37} = t_{36} + 0.199553566205637402320607178286E-26 \cdot R \cos (36.0 \cdot R \cdot t)
\]
\[
t_{38} = t_{37} - 0.279899581220179711426020884464E-26 \cdot R \cos (37.0 \cdot R \cdot t)
\]
\[
t_{39} = t_{38} - 0.553451183050700259947849425660E-27 \cdot R \cos (38.0 \cdot R \cdot t)
\]
\[
t_{40} = t_{39} - 0.388499542268455253129749000696E-27 \cdot R \cos (39.0 \cdot R \cdot t)
\]
\[
t_{41} = t_{40} - 0.112103404072307015240043264712E-27 \cdot R \cos (40.0 \cdot R \cdot t)
\]
\[
t_{42} = t_{41} - 0.5566582867449488057823816866E-28 \cdot R \cos (41.0 \cdot R \cdot t)
\]
\[
t_{43} = t_{42} - 0.204548261246513576288686878722E-28 \cdot R \cos (42.0 \cdot R \cdot t)
\]
\[
t_{44} = t_{43} + 0.845381404648938089437361193598E-29 \cdot R \cos (43.0 \cdot R \cdot t)
\]
\[
t_{45} = t_{44} + 0.356575515120151526590791715785E-29 \cdot R \cos (44.0 \cdot R \cdot t)
\]
\[
t_{46} = t_{45} + 0.13836524237797751810195772006E-29 \cdot R \cos (45.0 \cdot R \cdot t)
\]
\[
t_{47} = t_{46} - 0.60621426532093450576786528306E-30 \cdot R \cos (46.0 \cdot R \cdot t)
\]
\[
t_{47} \cdot \text{OPR}
\]

Table 4: Function Values of the Associated Functions
\[ x \quad t = -(20/x) - 1 \quad xe^{-x}Ei(x) \]

<table>
<thead>
<tr>
<th>x</th>
<th>( t = -(20/x) - 1 )</th>
<th>( xe^{-x}Ei(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-\infty)</td>
<td>1.000</td>
<td>0.1000000000 0000000000 0000000000 E 01</td>
</tr>
<tr>
<td>-160</td>
<td>-0.875</td>
<td>0.9938266956 7406127387 8797850088 E 00</td>
</tr>
<tr>
<td>-80</td>
<td>-0.750</td>
<td>0.9878013330 9428877356 4522608410 E 00</td>
</tr>
<tr>
<td>-53 1/3</td>
<td>-0.625</td>
<td>0.9819162901 3419439617 7735426105 E 00</td>
</tr>
<tr>
<td>-40</td>
<td>-0.500</td>
<td>0.9761646031 8514305080 8000604060 E 00</td>
</tr>
<tr>
<td>-32</td>
<td>-0.375</td>
<td>0.9705398840 7466392046 258464361 E 00</td>
</tr>
<tr>
<td>-26 2/3</td>
<td>-0.250</td>
<td>0.9650362511 2337703576 3536593528 E 00</td>
</tr>
<tr>
<td>-22 6/7</td>
<td>-0.125</td>
<td>0.9596482710 7936727616 5478970820 E 00</td>
</tr>
<tr>
<td>-20</td>
<td>-0.000</td>
<td>0.9543709099 1921683397 5195829433 E 00</td>
</tr>
<tr>
<td>-17 7/9</td>
<td>0.125</td>
<td>0.9491994907 7974574460 6445346803 E 00</td>
</tr>
<tr>
<td>-16</td>
<td>0.250</td>
<td>0.9441296577 3690297898 414971583 E 00</td>
</tr>
<tr>
<td>-14 6/11</td>
<td>0.375</td>
<td>0.9391573444 1928421424 0422409988 E 00</td>
</tr>
<tr>
<td>-13 1/3</td>
<td>0.500</td>
<td>0.9342787466 5341046480 9375801650 E 00</td>
</tr>
<tr>
<td>-12 4/13</td>
<td>0.625</td>
<td>0.9294902984 9721403772 5319679042 E 00</td>
</tr>
<tr>
<td>-11 3/7</td>
<td>0.750</td>
<td>0.9247886511 4084169605 5993585492 E 00</td>
</tr>
<tr>
<td>-10 2/3</td>
<td>0.875</td>
<td>0.9201706542 4944567620 2148012149 E 00</td>
</tr>
<tr>
<td>-10</td>
<td>1.000</td>
<td>0.9156333939 9788081876 0698157666 E 00</td>
</tr>
</tbody>
</table>

\[ x \quad t = -(x + 7)/3 \quad xe^{-x}Ei(x) \]

<table>
<thead>
<tr>
<th>x</th>
<th>(-20/x)</th>
<th>(xe^{-x}Ei(x))</th>
</tr>
</thead>
<tbody>
<tr>
<td>-10.00</td>
<td>-1.000</td>
<td>0.9156333939 9788081876 0698157666 E 01</td>
</tr>
<tr>
<td>-9.625</td>
<td>-0.875</td>
<td>0.9128444614 6799341885 6575662217 E 00</td>
</tr>
<tr>
<td>-9.250</td>
<td>-0.750</td>
<td>0.9098627515 2542413937 8954274597 E 00</td>
</tr>
<tr>
<td>-8.875</td>
<td>-0.625</td>
<td>0.9066672706 5475388033 4995756418 E 00</td>
</tr>
<tr>
<td>-8.500</td>
<td>-0.500</td>
<td>0.9032390197 7320784414 4682926135 E 00</td>
</tr>
<tr>
<td>-8.125</td>
<td>-0.375</td>
<td>0.8995347176 8847383630 1415777697 E 00</td>
</tr>
<tr>
<td>-7.750</td>
<td>-0.250</td>
<td>0.8955371870 8753915717 9475513219 E 00</td>
</tr>
<tr>
<td>-7.375</td>
<td>-0.125</td>
<td>0.8912031763 2125431626 708746258 E 00</td>
</tr>
<tr>
<td>-7.000</td>
<td>-0.000</td>
<td>0.8864876725 3642932889 3993846569 E 00</td>
</tr>
<tr>
<td>-6.625</td>
<td>0.125</td>
<td>0.8813371384 6821020039 4305076270 E 00</td>
</tr>
<tr>
<td>-6.250</td>
<td>0.250</td>
<td>0.8756873647 8846593227 646215532 E 00</td>
</tr>
<tr>
<td>-5.875</td>
<td>0.375</td>
<td>0.8694062945 5113401030 204713364 E 00</td>
</tr>
<tr>
<td>-5.500</td>
<td>0.500</td>
<td>0.8625618846 9070142209 0918986586 E 00</td>
</tr>
<tr>
<td>-5.125</td>
<td>0.625</td>
<td>0.8548755389 9019954239 2425567234 E 00</td>
</tr>
<tr>
<td>-4.750</td>
<td>0.750</td>
<td>0.8462482991 0358736117 1665798810 E 00</td>
</tr>
<tr>
<td>-4.375</td>
<td>0.875</td>
<td>0.8364987545 5629874174 2152267582 E 00</td>
</tr>
<tr>
<td>-4.000</td>
<td>1.000</td>
<td>0.8253825996 0422333240 8183035504 E 00</td>
</tr>
</tbody>
</table>
### Chapter 5

#### Chapter D

| $x$ | $t = x/4$ | $[Ei(x) - \log|x| - \gamma]/x$ |
|-----|----------|-------------------------------|
| -4.0 | -1.000 | 0.491823446 0781809647 9962798267 E 00 |
| -3.5 | -0.875 | 0.5248425066 4412835691 8258753311 E 00 |
| -3.0 | -0.750 | 0.5629587782 2127986313 8086024270 E 00 |
| -2.5 | -0.625 | 0.6073685258 5838306451 4266925640 E 00 |
| -2.0 | -0.500 | 0.6596316780 8476964479 5492023380 E 00 |
| -1.5 | -0.375 | 0.7218002369 4421992965 7623030310 E 00 |
| -1.0 | -0.250 | 0.7965995992 9705313428 3675865540 E 00 |
| -0.5 | -0.125 | 0.8876841582 3549672587 2151815870 E 00 |
| 0.0 | -0.000 | 0.1000000000 0000000000 0000000000 E 01 |
| 0.5 | 0.125 | 0.1140302841 0431720574 6248768807 E 01 |
| 1.0 | 0.250 | 0.1319021511 4544038948 6000844242 E 01 |
| 1.5 | 0.375 | 0.154536450 7467337302 4859074039 E 01 |
| 2.0 | 0.500 | 0.1841935755 2702059966 7788045934 E 01 |
| 2.5 | 0.625 | 0.2232103799 1211651144 5340506423 E 01 |
| 3.0 | 0.750 | 0.2752668205 6852580020 0219289740 E 01 |
| 3.5 | 0.875 | 0.3455821531 9301241243 7300898811 E 01 |
| 4.0 | 1.000 | 0.4416841111 0088991358 0118598668 E 01 |

<table>
<thead>
<tr>
<th>$x$</th>
<th>$t = (x - 8)/4$</th>
<th>$xe^{-x}Ei(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>-1.000</td>
<td>0.1438208031 4544827847 0968670330 E 01</td>
</tr>
<tr>
<td>4.5</td>
<td>-0.875</td>
<td>0.1396419029 6297460710 0674523183 E 01</td>
</tr>
<tr>
<td>5.0</td>
<td>-0.750</td>
<td>0.1353812777 4552859779 0189174047 E 01</td>
</tr>
<tr>
<td>5.5</td>
<td>-0.625</td>
<td>0.1314143565 7421192454 1219881699 E 01</td>
</tr>
<tr>
<td>6.0</td>
<td>-0.500</td>
<td>0.1278838600 4895616189 2314909578 E 01</td>
</tr>
<tr>
<td>6.5</td>
<td>-0.375</td>
<td>0.1248391155 0017048647 0471941387 E 01</td>
</tr>
<tr>
<td>7.0</td>
<td>-0.250</td>
<td>0.1222080528 3605310590 365846622 E 01</td>
</tr>
<tr>
<td>7.5</td>
<td>-0.125</td>
<td>0.1204124999 5996307864 3879158950 E 01</td>
</tr>
<tr>
<td>8.0</td>
<td>0.000</td>
<td>0.1181847986 9872079731 733962644 E 01</td>
</tr>
<tr>
<td>8.5</td>
<td>0.125</td>
<td>0.1161265251 8117484943 9918142965 E 01</td>
</tr>
<tr>
<td>9.0</td>
<td>0.250</td>
<td>0.1152759132 7089424132 2309981452 E 01</td>
</tr>
<tr>
<td>9.5</td>
<td>0.375</td>
<td>0.1141323475 9526242015 5338560641 E 01</td>
</tr>
<tr>
<td>10.0</td>
<td>0.500</td>
<td>0.1131470204 7341077803 4051681355 E 01</td>
</tr>
<tr>
<td>10.5</td>
<td>0.625</td>
<td>0.1122915570 0177600604 2888630755 E 01</td>
</tr>
<tr>
<td>11.0</td>
<td>0.750</td>
<td>0.1115430938 9980384416 4779434229 E 01</td>
</tr>
<tr>
<td>11.5</td>
<td>0.875</td>
<td>0.1108832926 3050773058 6855234934 E 01</td>
</tr>
<tr>
<td>12.0</td>
<td>1.000</td>
<td>0.1102974544 9067590726 7241234953 E 01</td>
</tr>
</tbody>
</table>
\[
\begin{array}{|c|c|c|}
\hline
x & t = (x - 22)/10 & xe^{-x}Ei(x) \\
\hline
12.00 & -1.000 & 0.1102974544 9067590726 7241234952 E 01 \\
13.25 & -0.875 & 0.1090844898 2154756926 6468614954 E 01 \\
14.50 & -0.750 & 0.1081351395 7351912850 6346643795 E 01 \\
15.75 & -0.625 & 0.1073701384 1997572371 2157900374 E 01 \\
17.00 & -0.500 & 0.1067393691 9585378312 957216197 E 01 \\
18.25 & -0.375 & 0.106206608 6221502426 837264556 E 01 \\
19.50 & -0.250 & 0.1057581342 1587250319 5393949410 E 01 \\
20.75 & -0.125 & 0.1053684451 2894094408 2103414964 E 01 \\
22.00 & -0.000 & 0.1050285719 6851897941 1780661532 E 01 \\
23.25 & 0.125 & 0.1047294557 0705348581 492365591 E 01 \\
24.50 & 0.250 & 0.1044614276 9046436368 9761075289 E 01 \\
25.75 & 0.375 & 0.1042271337 2023202388 5710928048 E 01 \\
27.00 & 0.500 & 0.1040144383 2301043813 713899754 E 01 \\
28.25 & 0.625 & 0.1038216700 3601458768 005648394 E 01 \\
29.50 & 0.750 & 0.1036487262 929418457 5154685419 E 01 \\
30.75 & 0.875 & 0.1034874149 896479647 2990938990 E 01 \\
32.00 & 1.000 & 0.1033413564 2162410494 3493552567 E 01 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|c|}
\hline
x & t = (64/x) - 1 & xe^{-x}Ei(x) \\
\hline
\infty & -1.000 & 0.100000000 000000000 0000000001 E 01 \\
512 & -0.875 & 0.1001960794 9450711925 313348473 E 01 \\
256 & -0.750 & 0.1003937130 9056986278 8809078297 E 01 \\
170 2/3 & -0.625 & 0.1005929275 6922929112 9466603932 E 01 \\
128 & -0.500 & 0.1007937524 4081401828 1776821694 E 01 \\
102 2/5 & -0.375 & 0.1009962174 7406449755 7436755570 E 01 \\
85 1/3 & -0.250 & 0.1012003545 3329884820 1864466702 E 01 \\
73 1/7 & -0.125 & 0.1014061949 9696913314 5942329335 E 01 \\
64 & -0.000 & 0.1016137726 4943252321 7035710831 E 01 \\
56 8/9 & 0.125 & 0.1018231211 88432696 8233017143 E 01 \\
51 1/5 & 0.250 & 0.1020342776 9307837747 8721782908 E 01 \\
46 6/11 & 0.375 & 0.1022472777 8405420595 9127536479 E 01 \\
42 2/3 & 0.500 & 0.1024621616 4681078391 0118780424 E 01 \\
39 5/13 & 0.625 & 0.1026789683 7290928524 5098450823 E 01 \\
36 4/7 & 0.750 & 0.1028977401 4015808008 6337835059 E 01 \\
34 2/15 & 0.875 & 0.1031185212 3645926355 875874663 E 01 \\
32 & 1.000 & 0.1033413564 2162410494 3493552567 E 01 \\
\hline
\end{array}
\]
The Fresnel Integral\cite{Pear56,Losc60}

The Fresnel function is
\[ C(x) - iS(x) = \int_{0}^{x} i^{-t^2} dt = \int_{0}^{x} \exp(-i\pi t^2/2) dt \]

We compare Axiom’s results to Pearcey’s tables which show the fresnel results to 6 decimal places. Computation of these values requires floats as the range quickly exceeds DoubleFloat. In each decade of the range we increase the number of terms by a factor of 10. So we compute with 10 terms in the range 0.0-10.0, 100 terms in 10.0-20.0, etc.

\texttt{fresnelC}

The \texttt{fresnelC} is the real portion of the Fresnel integral, \( C(u) \), is defined as:
\[ C(\sqrt{2x/\pi}) = \frac{1}{2} \int_{0}^{x} J_{-\frac{1}{2}}(t) dt = \frac{1}{\sqrt{(2\pi)}} \int_{0}^{x} \cos(t) \sqrt{t} dt \]

where \( J_{-\frac{1}{2}}(t) \) is the Bessel function of the first kind of order \( -\frac{1}{2} \).

This is related to the better known definition of \( C(u) \), namely:
\[ C(u) = \int_{0}^{u} \cos \frac{\pi t^2}{2} dt \]

where \( x = \pi u^2/2 \), or \( u = (2x/\pi)^{1/2} \)

\texttt{fresnelC} is an analytic function of \( z \) with \( z=0 \) as a two-sheeted branch point. Along the positive real axis the real definition gives:
\[ C(0) = 0 \]
\[ \lim_{x \to +\infty} C(x) = \frac{1}{2} \]

The asymptotic behavior of the function in the corner \( |\text{arc } z| \leq \pi - \epsilon, (\epsilon > 0) \), for \( |z| \gg 1 \) is given by
\[ C(z) \approx \frac{1}{2} + \frac{\sin z}{\sqrt{2\pi z}} \left( 1 - \frac{1 \cdot 3}{(2z)^2} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{(2z)^4} - \cdots \right) - \frac{\cos z}{\sqrt{2\pi z}} \left( \frac{1}{(2z)} - \frac{1 \cdot 3 \cdot 5}{(2z)^3} + \cdots \right) \]

(Note: Pearcey has a sign error for the second term \cite[p7]{Pear56})

The first approximation is
\[ C(z) \approx \frac{1}{2} + \frac{\sin z}{\sqrt{2\pi z}} \]

Axiom uses the power series at the zero point:
\[ C(z) = \sqrt{\frac{2z}{\pi}} \sum_{k=0}^{n} (-1)^k \frac{z^{2k}}{(4k + 1)(2k)!} \]
fresnelC(z:F):F ==
z < 0 => error "fresnelC not defined for negative argument"
z = 0 => 0
n:PI:= 100
sqrt((2.0/pi()$F)*z)*_
reduce(_+,[(-1)**k*z**(2*k)/(factorial(2*k)*(4*k+1))_
for k in 0..n]$F

|||-

fresnelS

The fresnelS is the complex portion of the Fresnel integral, $S(u)$, is defined as:

$$S(\sqrt{2x/\pi}) = \frac{1}{2} \int_0^x J_{1/2}(t) \, dt = \frac{1}{\sqrt{(2\pi)}} \int_0^x \frac{\sin(t)}{\sqrt{t}} \, dt$$

where $J_{1/2}(t)$ is the Bessel function of the first kind of order $1/2$.

This is related to the better known definition of $S(u)$, namely:

$$S(u) = \int_0^u \frac{\pi t^2}{2} \, dt$$

where $x = \pi u^2 / 2$, or $u = (2x/\pi)^{1/2}$.

fresnelS is an analytic function of $z$ with $z=0$ as a two-sheeted branch point. Along the positive real axis the real definition gives:

$$S(0) = 0$$

$$\lim_{x \to +\infty} S(x) = \frac{1}{2}$$

The asymptotic behavior of the function in the corner $|\text{arc } z| \leq \pi - \epsilon$, ($\epsilon > 0$), for $|z| \gg 1$ is given by

$$S(z) \approx \frac{1}{2} - \frac{\cos z}{\sqrt{2\pi z}} \left(1 - \frac{1 \cdot 3}{(2z)^2} + \frac{1 \cdot 3 \cdot 5 \cdot 7}{(2z)^4} - \cdots\right) - \frac{\sin z}{\sqrt{2\pi z}} \left(\frac{1}{(2z)} - \frac{1 \cdot 3 \cdot 5}{(2z)^3} + \cdots\right)$$

The first approximation is

$$S(z) \approx \frac{1}{2} - \frac{\cos z}{\sqrt{2\pi z}}$$
Axiom uses the power series at the zero point:

\[
S(z) = \sqrt{\frac{2z}{\pi}} \sum_{k=0}^{n} (-1)^k \frac{z^{2k+1}}{(4k+3)(2k+1)!}
\]

— package DFSFUN DoubleFloatSpecialFunctions —

```lisp
fresnelS(z:F):F ==
  z < 0 => error "fresnelS not defined for negative argument"
  z = 0 => 0
  n:PI:= 100
  sqrt((2.0/pi()$F)*z)*_
    reduce(_+,
      [(-1)**k*(z**(2*k+1))/(factorial(2*k+1)*(4*k+3))
        for k in 0..n])$LF
```

———

— package DFSFUN DoubleFloatSpecialFunctions —

```lisp
polygamma(k,z) == CPSI(k, z)$Lisp
polygamma(k,x) == RPSI(k, x)$Lisp
logGamma z == CLNGAMMA(z)$Lisp
logGamma x == RLNGAMMA(x)$Lisp
besselJ(v,z) == CBESSELJ(v,z)$Lisp
besselJ(n,x) == RBESSELJ(n,x)$Lisp
besselI(v,z) == CBESSELI(v,z)$Lisp
besselI(n,x) == RBESSELI(n,x)$Lisp
hypergeometric0F1(a,z) == CHYPER0F1(a, z)$Lisp
hypergeometric0F1(n,x) == retract hypergeometric0F1(n::C, x::C)
```

-- All others are defined in terms of these.
```lisp
digamma x == polygamma(0, x)
digamma z == polygamma(0, z)
Beta(x,y) == Gamma(x)*Gamma(y)/Gamma(x+y)
Beta(w,z) == Gamma(w)*Gamma(z)/Gamma(w+z)
fuzz := (10::R)**(-7)
```

import IntegerRetractions(R)
import IntegerRetractions(C)
besselY(n,x) ==
  if integer? n then n := n + fuzz
  vp := n * pi()$R
  (cos(vp) * besselJ(n,x) - besselJ(-n,x) )/sin(vp)

besselY(v,z) ==
  if integer? v then v := v + fuzz::C
  vp := v * pi()$C
  (cos(vp) * besselJ(v,z) - besselJ(-v,z) )/sin(vp)

besselK(n,x) ==
  if integer? n then n := n + fuzz
  p := pi()$R
  vp := n*p
  ahalf:= 1/(2::R)
  p * ahalf * ( besselI(-n,x) - besselI(n,x) )/sin(vp)

besselK(v,z) ==
  if integer? v then v := v + fuzz::C
  p := pi()$C
  vp := v*p
  ahalf:= 1/(2::C)
  p * ahalf * ( besselI(-v,z) - besselI(v,z) )/sin(vp)

airyAi x ==
  ahalf := recip(2::R)::R
  athird := recip(3::R)::R
  eta := 2 * athird * (-x) ** (3*ahalf)
  (-x)**ahalf * athird * (besselJ(-athird,eta) + besselJ(athird,eta))

airyAi z ==
  ahalf := recip(2::C)::C
  athird := recip(3::C)::C
  eta := 2 * athird * (-z) ** (3*ahalf)
  (-z)**ahalf * athird * (besselJ(-athird,eta) + besselJ(athird,eta))

airyBi x ==
  ahalf := recip(2::R)::R
  athird := recip(3::R)::R
  eta := 2 * athird * (-x) ** (3*ahalf)
  (-x*athird)**ahalf * ( besselJ(-athird,eta) - besselJ(athird,eta) )

airyBi z ==
  ahalf := recip(2::C)::C
  athird := recip(3::C)::C
  eta := 2 * athird * (-z) ** (3*ahalf)
  (-z*athird)**ahalf * ( besselJ(-athird,eta) - besselJ(athird,eta) )
package DBLRESP DoubleResultantPackage

-- DoubleResultantPackage.input --

)set break resume
/sys rm -f DoubleResultantPackage.output
/spool DoubleResultantPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DoubleResultantPackage
--R
--R DoubleResultantPackage(F: Field,UP: UnivariatePolynomialCategory(F),UPUP: UnivariatePolynomialCategory(Fraction(UP)),R: FunctionFieldCategory(F,UP,UPUP)) is a package constructor
--R Abbreviation for DoubleResultantPackage is DBLRESP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DBLRESP
--R
--R----------------------------------- Operations -------------------------------
--R doubleResultant : (R,(UP -> UP)) -> UP
--R
--E 1

)spool
)lisp (bye)

---

-- DoubleResultantPackage.help --

====================================================================
DoubleResultantPackage examples
====================================================================

This package provides functions for computing the residues of a function on an algebraic curve.
See Also:
- )show DoubleResultantPackage

DoubleResultantPackage (DBLRESP)

Exports:
doubleResultant

-- package DBLRESP DoubleResultantPackage --

)abbrev package DBLRESP DoubleResultantPackage
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 July 1990
++ Description:
++ This package provides functions for computing the residues
++ of a function on an algebraic curve.

DoubleResultantPackage(F, UP, UPUP, R): Exports == Implementation where
  F : Field
  UP : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R : FunctionFieldCategory(F, UP, UPUP)

  RF ===> Fraction UP
  UP2 ===> SparseUnivariatePolynomial UP
  UP3 ===> SparseUnivariatePolynomial UP2

Exports === with
  doubleResultant: (R, UP -> UP) -> UP
++ doubleResultant(f, ') returns p(x) whose roots are 
++ rational multiples of the residues of f at all its 
++ finite poles. Argument ' is the derivation to use.

Implementation ==> add
import CommuteUnivariatePolynomialCategory(F, UP, UP2)
import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)

UP2 : UP -> UP2
UP23 : UPUP -> UP3
remove0: UP -> UP -- removes the power of x dividing p

remove0 p ==
primitivePart((p exquo monomial(1, minimumDegree p))::UP)

UP22 p ==
map(x+->x::UP, p)$UnivariatePolynomialCategoryFunctions2(F,UP,UP,UP2)

UP23 p ==
map(x+->UP22(retract(x)@UP),p)_
   $UnivariatePolynomialCategoryFunctions2(RF, UPUP, UP2, UP3)

doubleResultant(h, derivation) ==
cd := splitDenominator lift h
d := (cd.den exquo (g := gcd(cd.den, derivation(cd.den))))::UP
r := swap primitivePart swap resultant(UP23(cd.num)- 
   ((monomial(1, 1)$UP :: UP2) * UP22(g * derivation d))::UP3,
      UP23 definingPolynomial())
remove0 resultant(r, UP22 d)

——
— DBLRESP.dotabb —

"DBLRESP" [color="#FF4488",href=bookvol10.4.pdf#nameddest=DBLRESP"]
"FFCAT" [color="#4488FF",href=bookvol10.2.pdf#nameddest=FFCAT"]
"DBLRESP" -> "FFCAT"

——

package DRAWCX DrawComplex

— DrawComplex.input —

)set break resume
--S 1 of 1
)--spool DrawComplex.output
)clear all

--R DrawComplex is a package constructor
--R Abbreviation for DrawComplex is DRAWCX
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DRAWCX
--R
--R----------------------------------- Operations -----------------------------
--R setImagSteps : Integer -> Integer  setRealSteps : Integer -> Integer
--R drawComplex : ((Complex(DoubleFloat) -> Complex(DoubleFloat)),Segment(DoubleFloat),Segment(DoubleFloat)) -> ThreeDimensionalViewport
--R drawComplexVectorField : ((Complex(DoubleFloat) -> Complex(DoubleFloat)),Segment(DoubleFloat),Segment(DoubleFloat)) -> ThreeDimensionalViewport
--R setClipValue : DoubleFloat -> DoubleFloat
--R
--E 1

)--spool
)--lisp (bye)

---

— DrawComplex.help —

====================================================================
DrawComplex examples
====================================================================

DrawComplex provides some facilities for drawing complex functions.

See Also:
 o )show DrawComplex

---
DrawComplex (DRAWCX)

Exports:
setImagSteps setRealSteps drawComplex drawComplexVectorField setClipValue

— package DRAWCX DrawComplex —

)abbrev package DRAWCX DrawComplex
++ Description:
++ \axiomType{DrawComplex} provides some facilities
++ for drawing complex functions.

DrawComplex(): Exports == Implementation where
C ==> Complex DoubleFloat
S ==> Segment DoubleFloat
PC ==> Record(rr:SF, th:SF)
INT ==> Integer
SF ==> DoubleFloat
NNI ==> NonNegativeInteger
VIEW3D ==> ThreeDimensionalViewport
ARRAY2 ==> TwoDimensionalArray
Exports == with
drawComplex: (C -> C,S,S,Boolean) -> VIEW3D
  ++ drawComplex(f,rRange,iRange,arrows?)
  ++ draws a complex function as a height field.
  ++ It uses the complex norm as the height and the complex
  ++ argument as the color.
  ++ It will optionally draw arrows on the surface indicating the direction
  ++ of the complex value.
  ++ Sample call:
  ++ \spad{\textcolor{blue}{\texttt{f z == exp(1/z)}}}
  ++ \spad{\textcolor{blue}{\texttt{drawComplex(f, 0.3..3, 0..2*\%pi, false)}}}
  ++ Parameter descriptions:
  ++ f: the function to draw
  ++ rRange : the range of the real values
  ++ iRange : the range of imaginary values
  ++ arrows? : a flag indicating whether to draw the phase arrows for f
++ Call the functions \texttt{setRealSteps}\{DrawComplex\} and ++ \texttt{setImagSteps}\{DrawComplex\} to change the ++ number of steps used in each direction.

drawComplexVectorField: (C -> C,S,S) -> VIEW3D ++ drawComplexVectorField(f,\text{rRange},\text{iRange}) ++ draws a complex vector field using arrows on the \texttt{x--y} plane. ++ These vector fields should be viewed from the top by pressing the ++ "XY" translate button on the 3-d viewport control panel. ++ Sample call: ++ \texttt{f \text{z} == \text{sin \text{z}}} ++ \texttt{drawComplexVectorField(f, -2..2, -2..2)} ++ Parameter descriptions: ++ \texttt{f} : the function to draw ++ \texttt{rRange} : the range of the real values ++ \texttt{iRange} : the range of the imaginary values ++ Call the functions \texttt{setRealSteps}\{DrawComplex\} and ++ \texttt{setImagSteps}\{DrawComplex\} to change the ++ number of steps used in each direction.

setRealSteps: INT -> INT ++ setRealSteps(i) ++ sets to \texttt{i} the number of steps to use in the real direction ++ when drawing complex functions. Returns \texttt{i}.

setImagSteps: INT -> INT ++ setImagSteps(i) ++ sets to \texttt{i} the number of steps to use in the imaginary direction ++ when drawing complex functions. Returns \texttt{i}.

setClipValue: SF-> SF ++ setClipValue(x) ++ sets to \texttt{x} the maximum value to plot when drawing complex functions. Returns \texttt{x}.

Implementation == add

-- relative size of the arrow head compared to the length of the arrow arrowScale : SF := (0.125)::SF
arrowAngle: SF := pi()-pi()/\texttt{20::SF} -- angle of the arrow head
realSteps: INT := 11 -- the number of steps in the real direction
imagSteps: INT := 11 -- the number of steps in the imaginary direction
clipValue: SF := 10::SF -- the maximum length of a vector to draw

-- Add an arrow head to a line segment, which starts at 'p1', ends at 'p2', -- has length 'len', and angle 'arg'. We pass 'len' and 'arg' as -- arguments since they were already computed by the calling program
makeArrow(p1:Point SF, p2:Point SF, len: SF, arg:SF):List List Point SF ==
c1 := cos(arg + arrowAngle)
s1 := sin(arg + arrowAngle)
c2 := cos(arg - arrowAngle)
s2 := sin(arg - arrowAngle)
p3 := point \{p2.1 + c1*arrowScale*len, p2.2 + s1*arrowScale*len, p2.3, p2.4\}
p4 := point \{p2.1 + c2*arrowScale*len, p2.2 + s2*arrowScale*len, p2.3, p2.4\}
-- clip a value in the interval (-clip...clip)
clipFun(x:SF):SF ==
min(max(x, -clipValue), clipValue)

drawComplex(f, realRange, imagRange, arrows?) ==
delReal := (hi(realRange) - lo(realRange))/realSteps::SF
delImag := (hi(imagRange) - lo(imagRange))/imagSteps::SF
funTable: ARRAY2(PC) :=
new((realSteps::NNI)+1, (imagSteps::NNI)+1, [0,0]$PC)
real := lo(realRange)
for i in 1..realSteps+1 repeat
  imag := lo(imagRange)
  for j in 1..imagSteps+1 repeat
    z := f complex(real, imag)
    funTable(i,j) := [clipFun(sqrt norm z), argument(z)]$PC
    imag := imag + delImag
    real := real + delReal
  llp := empty()$(List List Point SF)
  real := lo(realRange)
  for i in 1..realSteps+1 repeat
    imag := lo(imagRange)
    lp := empty()$(List Point SF)
    for j in 1..imagSteps+1 repeat
      p := point [real, imag, funTable(i,j).rr, funTable(i,j).th]
      lp := cons(p, lp)
    imag := imag + delImag
    real := real + delReal
    llp := cons(lp, llp)
  llp := empty()$(List List Point SF)
  if arrows? then
    real := lo(realRange)
    for i in 1..realSteps+1 repeat
      imag := lo(imagRange)
      for j in 1..imagSteps+1 repeat
        arg := funTable(i,j).th
        p1 := point [real, imag, funTable(i,j).rr, arg]
        len := delReal*2.0::SF
        p2 := point [p1.1 + len*cos(arg), p1.2 + len*sin(arg),
                     p1.3, p1.4]
        arrow := makeArrow(p1, p2, len, arg)
        for a in arrow repeat curve(space, a)$(ThreeSpace SF)
        imag := imag + delImag
        real := real + delReal
    space := mesh(llp)$ThreeSpace
    makeViewport3D(space, "Complex Function")$VIEW3D

drawComplexVectorField(f, realRange, imagRange): VIEW3D ==
-- compute the steps size of the grid
  delReal := (hi(realRange) - lo(realRange))/realSteps::SF
\begin{verbatim}

delImag := (hi(imagRange) - lo(imagRange))/imagSteps::SF
-- create the space to hold the arrows
space := create3Space()$(ThreeSpace SF)
real := lo(realRange)
for i in 1..realSteps+1 repeat
  imag := lo(imagRange)
  for j in 1..imagSteps+1 repeat
    -- compute the function
    z := f complex(real, imag)
    -- get the direction of the arrow
    arg := argument z
    -- get the length of the arrow
    len := clipFun(sqrt norm z)
    -- create point at the base of the arrow
    p1 := point [real, imag, 0::SF, arg]
    -- scale the arrow length so it isn't too long
    scaleLen := delReal * len
    -- create the point at the top of the arrow
    p2 := point [p1.1 + scaleLen*cos(arg), p1.2 + scaleLen*sin(arg),
                 0::SF, arg]
    -- make the pointer at the top of the arrow
    arrow := makeArrow(p1, p2, scaleLen, arg)
    -- add the line segments in the arrow to the space
    for a in arrow repeat curve(space, a)$(ThreeSpace SF)
  imag := imag + delImag
  real := real + delReal
-- draw the vector field
makeViewport3D(space, "Complex Vector Field")$VIEW3D

-- set the number of steps to use in the real direction
setRealSteps(n) ==
  realSteps := n

-- set the number of steps to use in the imaginary direction
setImagSteps(n) ==
  imagSteps := n

-- set the maximum value to plot
setClipValue clip ==
  clipValue := clip

-----

"DRAWCX" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWCX"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
\end{verbatim}
"DRAWCX" -> "FIELD"
"DRAWCX" -> "RADCAT"

package DRAWHACK DrawNumericHack

--- DrawNumericHack.input ---

)set break resume
)sys rm -f DrawNumericHack.output
)spool DrawNumericHack.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DrawNumericHack
  --R
  --R DrawNumericHack(R: Join(OrderedSet,IntegralDomain,ConvertibleTo(Float))) is a package constructor
  --R Abbreviation for DrawNumericHack is DRAWHACK
  --R This constructor is exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for DRAWHACK
  --R
  --R------------------------------------------ Operations ------------------------------------------
  --R coerce : SegmentBinding(Expression(R)) -> SegmentBinding(Float)
  --R
  --E 1

)spool
)lisp (bye)

--- DrawNumericHack.help ---

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
DrawNumericHack examples
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Hack for the draw interface. DrawNumericHack provides a "coercion" from something of the form \( x = a..b \) where \( a \) and \( b \) are formal expressions to a binding of the form \( x = c..d \) where \( c \) and \( d \) are the numerical values of \( a \) and \( b \). This "coercion" fails if \( a \) and \( b \) contains symbolic variables, but is meant for expressions involving \( \%\pi \).
Note that this package is meant for internal use only.

See Also:
- )show DrawNumericHack

---

**DrawNumericHack (DRAWHACK)**

Exports:
- coerce

---

```spad
)abbrev package DRAWHACK DrawNumericHack
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 21 Feb 1990
++ Description:
++ Hack for the draw interface. DrawNumericHack provides
++ a "coercion" from something of the form \( x = a..b \) where \( a \)
++ and \( b \) are
++ formal expressions to a binding of the form \( x = c..d \) where \( c \) and \( d \)
++ are the numerical values of \( a \) and \( b \). This "coercion" fails if
++ \( a \) and \( b \) contains symbolic variables, but is meant for expressions
++ involving \%pi.
++ Note that this package is meant for internal use only.

DrawNumericHack(R:Join(OrderedSet,IntegralDomain,ConvertibleTo Float)):
with coerce: SegmentBinding Expression R -> SegmentBinding Float
  ++ coerce(x = a..b) returns \( x = c..d \) where \( c \) and \( d \) are the
  ++ numerical values of \( a \) and \( b \).
```
== add coerce s ==
  map(numeric$Numeric(R),s)$SegmentBindingFunctions2(Expression R, Float)

---
---
---

 package DROPT0 DrawOptionFunctions0

--- DrawOptionFunctions0.input ---

)set break resume
)sys rm -f DrawOptionFunctions0.output
)spool DrawOptionFunctions0.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DrawOptionFunctions0

--R
--R DrawOptionFunctions0 is a package constructor
--R Abbreviation for DrawOptionFunctions0 is DROPT0
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DROPT0
--R
--R-------------------------------------- Operations -----------------------------------
--R adaptive : (List(DrawOption),Boolean) -> Boolean
--R clipBoolean : (List(DrawOption),Boolean) -> Boolean
--R coord : (List(DrawOption),(Point(DoubleFloat) -> Point(DoubleFloat))) -> (Point(DoubleFloat))
--R curveColorPalette : (List(DrawOption),Palette) -> Palette
--R pointColorPalette : (List(DrawOption),Palette) -> Palette
--R ranges : (List(DrawOption),List(Segment(Float))) -> List(Segment(Float))
--R space : List(DrawOption) -> ThreeSpace(DoubleFloat)
--R style : (List(DrawOption),String) -> String
--R title : (List(DrawOption),String) -> String
--R toScale : (List(DrawOption),Boolean) -> Boolean

--- DRAWHACK.dotabb ---

"DRAWHACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAWHACK"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"DRAWHACK" -> "ALGEBRA"
PACKAGE DROPT0 DRAWOPTIONFUNCTIONS0

--R tubePoints : (List(DrawOption),PositiveInteger) -> PositiveInteger
--R tubeRadius : (List(DrawOption),Float) -> Float
--R units : (List(DrawOption),List(Float)) -> List(Float)
--R var1Steps : (List(DrawOption),PositiveInteger) -> PositiveInteger
--R var2Steps : (List(DrawOption),PositiveInteger) -> PositiveInteger
--R viewpoint : (List(DrawOption),Record(theta: DoubleFloat,phi: DoubleFloat,scale: DoubleFloat,scaleX: DoubleFloat,scaleY: DoubleFloat,scaleZ: DoubleFloat,deltaX: DoubleFloat,deltaY: DoubleFloat)) -> E

)spool
)lisp (bye)

DrawOptionFunctions0 (DROPT0)

Exports:
  adaptive    clipBoolean      coord    curveColorPalette    pointColorPalette
  ranges      space            style    title                toScale
  tubePoints  tubeRadius       units    var1Steps            var2Steps
  viewpoint

This package has no description

See Also:
  o )show DrawOptionFunctions0
package DROPT0 DrawOptionFunctions0

abbrev package DROPT0 DrawOptionFunctions0
++ Description:
++ This package has no description
-- The functions here are not in DrawOptions since they are not
-- visible to the interpreter.

DrawOptionFunctions0(): Exports == Implementation where
RANGE ==> List Segment Float
UNIT ==> List Float
PAL ==> Palette
POINT ==> Point(DoubleFloat)
SEG ==> Segment Float
SF ==> DoubleFloat
SPACE3 ==> ThreeSpace(DoubleFloat)
VIEWPT ==> Record( theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, deltaY:SF )

Exports ==> with
adaptive: (List DrawOption, Boolean) -> Boolean
  ++ adaptive(l,b) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{adaptive}.
  ++ If the option does not exist the value, b is returned.
clipBoolean: (List DrawOption, Boolean) -> Boolean
  ++ clipBoolean(l,b) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{clipBoolean}.
  ++ If the option does not exist the value, b is returned.
viewpoint: (List DrawOption, VIEWPT) -> VIEWPT
  ++ viewpoint(l,ls) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{viewpoint}.
  ++ IF the option does not exist, the value ls is returned.
title: (List DrawOption, String) -> String
  ++ title(l,s) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{title}.
  ++ If the option does not exist the value, s is returned.
sty le: (List DrawOption, String) -> String
  ++ style(l,s) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{style}.
  ++ If the option does not exist the value, s is returned.
toScale: (List DrawOption, Boolean) -> Boolean
  ++ toScale(l,b) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{toScale}.
  ++ If the option does not exist the value, b is returned.
pointColorPalette: (List DrawOption,PAL) -> PAL
  ++ pointColorPalette(l,p) takes the list of draw options, l, and checks
  ++ the list to see if it contains the option \spad{pointColorPalette}.
  ++ If the option does not exist the value, p is returned.
curveColorPalette: (List DrawOption,PAL) -> PAL
++ curveColorPalette(l,p) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{curveColorPalette}.
++ If the option does not exist the value, p is returned.

ranges: (List DrawOption, RANGE) -> RANGE
++ ranges(l,r) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{ranges}.
++ If the option does not exist the value, r is returned.

var1Steps: (List DrawOption, PositiveInteger) -> PositiveInteger
++ var1Steps(l,n) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{var1Steps}.
++ If the option does not exist the value, n is returned.

var2Steps: (List DrawOption, PositiveInteger) -> PositiveInteger
++ var2Steps(l,n) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{var2Steps}.
++ If the option does not exist the value, n is returned.

space: (List DrawOption) -> SPACE3
++ space(l) takes a list of draw options, l, and checks to see
++ if it contains the option \spad{space}. If the the option
++ doesn’t exist, then an empty space is returned.

tubePoints : (List DrawOption, PositiveInteger) -> PositiveInteger
++ tubePoints(l,n) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{tubePoints}.
++ If the option does not exist the value, n is returned.

tubeRadius : (List DrawOption, Float) -> Float
++ tubeRadius(l,n) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{tubeRadius}.
++ If the option does not exist the value, n is returned.

coord: (List DrawOption, (POINT->POINT)) -> (POINT->POINT)
++ coord(l,p) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{coord}.
++ If the option does not exist the value, p is returned.

units: (List DrawOption, UNIT) -> UNIT
++ units(l,u) takes the list of draw options, l, and checks
++ the list to see if it contains the option \spad{unit}.
++ If the option does not exist the value, u is returned.

Implementation ==> add

adaptive(l,s) ==
  (u := option(l, "adaptive"::Symbol)$DrawOptionFunctions1(Boolean))
  case "failed" => s
  u::Boolean

clipBoolean(l,s) ==
  (u := option(l, "clipBoolean"::Symbol)$DrawOptionFunctions1(Boolean))
  case "failed" => s
  u::Boolean

title(l, s) ==
  (u := option(l, "title"::Symbol)$DrawOptionFunctions1(String))
case "failed" => s
u::String

viewpoint(l, vp) ==
  (u := option(l, "viewpoint"::Symbol)$DrawOptionFunctions1(VIEWPT))
  case "failed" => vp
  u::VIEWPT

style(l, s) ==
  (u := option(l, "style"::Symbol)$DrawOptionFunctions1(String))
  case "failed" => s
  u::String

toScale(l,s) ==
  (u := option(l, "toScale"::Symbol)$DrawOptionFunctions1(Boolean))
  case "failed" => s
  u::Boolean

pointColorPalette(l,s) ==
  (u := option(l, "pointColorPalette"::Symbol)$DrawOptionFunctions1(PAL))
  case "failed" => s
  u::PAL

curveColorPalette(l,s) ==
  (u := option(l, "curveColorPalette"::Symbol)$DrawOptionFunctions1(PAL))
  case "failed" => s
  u::PAL

ranges(l, s) ==
  (u := option(l, "ranges"::Symbol)$DrawOptionFunctions1(RANGE))
  case "failed" => s
  u::RANGE

space(l) ==
  (u := option(l, "space"::Symbol)$DrawOptionFunctions1(SPACE3))
  case "failed" => create3Space()$SPACE3
  u::SPACE3

var1Steps(l,s) ==
  (u := option(l, "var1Steps"::Symbol)$DrawOptionFunctions1(PositiveInteger))
  case "failed" => s
  u::PositiveInteger

var2Steps(l,s) ==
  (u := option(l, "var2Steps"::Symbol)$DrawOptionFunctions1(PositiveInteger))
  case "failed" => s
  u::PositiveInteger
PACKAGE DROPT1 DRAWOPTIONFUNCTIONS1

tubePoints(l,s) ==
  (u := option(l, "tubePoints"::Symbol)$DrawOptionFunctions1(PositiveInteger))
  case "failed" => s
  u::PositiveInteger

tubeRadius(l,s) ==
  (u := option(l, "tubeRadius"::Symbol)$DrawOptionFunctions1(Float))
  case "failed" => s
  u::Float

coord(l,s) ==
  (u := option(l, "coord"::Symbol)$DrawOptionFunctions1(POINT->POINT))
  case "failed" => s
  u::(POINT->POINT)

units(l,s) ==
  (u := option(l, "unit"::Symbol)$DrawOptionFunctions1(UNIT))
  case "failed" => s
  u::UNIT

— DROPT0.dotabb —

"DROPT0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DROPT0"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DROPT0" -> "ALIST"

package DROPT1 DrawOptionFunctions1

— DrawOptionFunctions1.input —

)set break resume
)sys rm -f DrawOptionFunctions1.output
)spool DrawOptionFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show DrawOptionFunctions1
--R
CHAPTER 5. CHAPTER D

---R DrawOptionFunctions1(S: Type) is a package constructor
---R Abbreviation for DrawOptionFunctions1 is DROPT1
---R This constructor is not exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for DROPT1
---R
---R------------------------------- Operations --------------------------------
---R option : (List(DrawOption),Symbol) -> Union(S,"failed")
---R
---E 1

)spool
)lisp (bye)

——

— DrawOptionFunctions1.help —

====================================================================
DrawOptionFunctions1 examples
====================================================================

This package has no description

See Also:
- )show DrawOptionFunctions1

——

DrawOptionFunctions1 (DROPT1)

Exports:
option
— package DROPT1 DrawOptionFunctions1 —

)abbrev package DROPT1 DrawOptionFunctions1
++ Description:
++ This package has no description

DrawOptionFunctions1(S:Type): Exports == Implementation where
RANGE ==> List Segment Float
UNIT ==> List Float
PAL ==> Palette
POINT ==> Point(DoubleFloat)
SEG ==> Segment Float
SF ==> DoubleFloat
SPACE3 ==> ThreeSpace(DoubleFloat)
VIEWPT ==> Record( theta:SF, phi:SF, scale:SF, scaleX:SF, scaleY:SF, scaleZ:SF, deltaX:SF, deltaY:SF )

Exports ==> with
option: (List DrawOption, Symbol) -> Union(S, "failed")
++ option(l,s) determines whether the indicated drawing option, s,
++ is contained in the list of drawing options, l, which is defined
++ by the draw command.
Implementation ==> add
option(l, s) ==
(u := option(l, s)@Union(Any, "failed")) case "failed" => "failed"
retract(u::Any)$AnyFunctions1(S)

——

— DROPT1.dotabb —

"DROPT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DROPT1"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"DROPT1" -> "TYPE"

——

package D01AGNT d01AgentsPackage

— d01AgentsPackage.input —

)set break resume
)sys rm -f d01AgentsPackage.output
)spool d01AgentsPackage.output
\texttt{)}set message test on
\texttt{)}set message auto off

\texttt{)}clear all

\texttt{--S 1 of 136}
\texttt{)}show d01AgentsPackage
\texttt{--R}
\texttt{--R d01AgentsPackage is a package constructor}
\texttt{--R Abbreviation for d01AgentsPackage is D01AGNT}
\texttt{--R This constructor is exposed in this frame.}
\texttt{--R Issue \texttt{)}edit\texttt{ bookvol10.4.pamphlet\ to see algebra source code for D01AGNT}
\texttt{--R}
\texttt{--R------------------------------- Operations -------------------------------}
\texttt{--R df2st : DoubleFloat \rightarrow String}
\texttt{--R changeName \rightarrow Result}
\texttt{--R commaSeparate : List(String) \rightarrow String}
\texttt{--R functionIsContinuousAtEndPoints : Record(var: Symbol, fn: Expression(DoubleFloat), range: Segment(OrderedCompletion(DoubleFloat))) \rightarrow Boolean}
\texttt{--R gethi : Segment(OrderedCompletion(DoubleFloat)) \rightarrow DoubleFloat}
\texttt{--R getlo : Segment(OrderedCompletion(DoubleFloat)) \rightarrow DoubleFloat}
\texttt{--R ldf2lst : List(DoubleFloat) \rightarrow List(String)}
\texttt{--R problemPoints : (Expression(DoubleFloat), Symbol, Segment(OrderedCompletion(DoubleFloat))) \rightarrow List(DoubleFloat)}
\texttt{--R rangeIsFinite : Record(var: Symbol, fn: Expression(DoubleFloat), range: Segment(OrderedCompletion(DoubleFloat)), abserr: DoubleFloat, relerr: DoubleFloat) \rightarrow Boolean}
\texttt{--R sdf2lst : Stream(DoubleFloat) \rightarrow List(String)}
\texttt{--R singularitiesOf : Record(var: Symbol, fn: Expression(DoubleFloat), range: Segment(OrderedCompletion(DoubleFloat)), abserr: DoubleFloat, relerr: DoubleFloat) \rightarrow Stream(DoubleFloat)}

\texttt{--E 1}

\texttt{)}clear all

\texttt{--S 2 of 136}
\texttt{showArrayValues true}
\texttt{--R}
\texttt{--R (1) true}
\texttt{--E 2}

\texttt{--S 3 of 136}
\texttt{showScalarValues true}
\texttt{--R}
\texttt{--R (2) true}
\texttt{--E 3}

\texttt{--S 4 of 136}
\texttt{e:EXPR FLOAT:=(X*sin(30*X)/(sqrt(1-(X/(2*%pi))\^2)))}
\texttt{--R}
--R
--R \ X \sin(30.0 \ X)
--R \ (3) \---------------------------------------
--R \ + \------------------------------------+
--R \ | \ 2
--R \ \ \ \ \ \ \ \ 0.0253302959 1058444286 1 X + 1.0
--R
--R Type: Expression(Float)
--E 4

--S 5 of 136
f:ASP1(F):=retract e
--R
--R \ (4) F
--R Type: Asp1(F)
--E 5

--S 6 of 136
a:SF:=0.0
--R
--R \ (5) 0.
--R Type: DoubleFloat
--E 6

--S 7 of 136
b:SF:=\pi*2
--R
--R \ (6) 6.2831853071795862
--R Type: DoubleFloat
--E 7

--S 8 of 136
epsabs:SF:=0.0
--R
--R \ (7) 0.
--R Type: DoubleFloat
--E 8

--S 9 of 136
epsrel:SF:=1.0e-4
--R
--R \ (8) 9.9999999999999991E-5
--R Type: DoubleFloat
--E 9

--S 10 of 136
liw:=200
--R
--R
--R (9) 200
--R
--E 10

lw:=4*liw
--R
--R
--R (10) 800
--R
--E 11

result:=d01ajf(a,b,epsabs,epsrel,1w,liw,-1,f)
--E 12

)clear all

showArrayValues true
--R
--R
--R (1) true
--R
--E 13

showScalarValues true
--R
--R
--R (2) true
--R
--E 14

e:EXPR FLOAT:=X*sin(30.0*X)*cos(X)
--R
--R
--R (3) X cos(X)sin(30.0 X)
--R
--E 15

f:ASP1(F):=retract e
--R
--R
--R (4) F
a:SF:=0.0
--R
--R
--R (5) 0.
--R
--E 17

b:SF:=\pi*2
--R
--R
--R (6) 6.2831853071795862
--R
--E 18

epsabs:SF:=0.0
--R
--R
--R (7) 0.
--R
--E 19

epsrel:SF:=1.0e-4
--R
--R
--R (8) 9.9999999999999991E-5
--R
--E 20

liw:=200
--R
--R
--R (9) 200
--R
--E 21

lw:=4*liw
--R
--R
--R (10) 800
--R
--E 22
chapter 5. chapter d

result := d01akf(a, b, epsabs, epsrel, lw, liw, -1, f)

)clear all

showArrayValues true

--R

(1) true

Type: Boolean

showScalarValues true

--R

(2) true

Type: Boolean

e := 1/sqrt(abs(X - 1/7))

1.0

|abs(X - 0.1428571428 5714285714)

Type: Expression(Float)

f := retract e

(4) F

Type: Asp1(F)

a := 0.0

(5) 0.

Type: DoubleFloat
--S 29 of 136
b:SF:=1.0
--R
--R
--R (6) 1.
--R Type: DoubleFloat
--E 29

--S 30 of 136
points:Matrix SF:=[[1/7]]
--R
--R
--R (7) [0.14285714285714285]
--R Type: Matrix(DoubleFloat)
--E 30

--S 31 of 136
npts:=ncols points
--R
--R
--R (8) 1
--R Type: PositiveInteger
--E 31

--S 32 of 136
epsabs:SF:=0.0
--R
--R
--R (9) 0.
--R Type: DoubleFloat
--E 32

--S 33 of 136
epsrel:SF:=1.0e-4
--R
--R
--R (10) 9.9999999999999991E-5
--R Type: DoubleFloat
--E 33

--S 34 of 136
liw:=max(npts*4,400)
--R
--R
--R (11) 400
--R Type: PositiveInteger
--E 34

--S 35 of 136
lw:=max(2*npts*8,2*liw)
--R
--R
--R (12) 800
--R Type: PositiveInteger
--E 35

--S 36 of 136
-- result:=d01alf(a,b,npts,points,epsabs,epsrel,1w,liw,-1,f)
--E 36

)clear all

--S 37 of 136
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 37

--S 38 of 136
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 38

--S 39 of 136
e:EXPR FLOAT:=1/((X+1)*sqrt(X))
--R
--R
--R 1.0
--R (3) -----------
--R Type: Expression(Float)
--E 39

--S 40 of 136
f:ASP1(F):=retract e
--R
--R
--R (4) F
--R Type: Asp1(F)
--E 40

--S 41 of 136
a:SF:=0.0
--R
--R
--R  (5)  0.
--R
--E 41

--S 42 of 136
epsabs:SF:=0.0
--R
--R  (6)  0.
--R
--E 42

--S 43 of 136
epsrel:SF:=1.0e-4
--R
--R  (7)  9.9999999999999991E-5
--R
--E 43

--S 44 of 136
liw:=200
--R
--R  (8)  200
--R
--E 44

--S 45 of 136
lw:=4*liw
--R
--R  (9)  800
--R
--E 45

--S 46 of 136
result:=d01amf(a,1,epsabs,epsrel,liw,liw,-1,f)
--E 46

)clear all

--S 47 of 136
showArrayValues true
--R
--R  (1)  true
--R
--E 47
showScalarValues true
(2) true
Type: Boolean

---
e:EXPR FLOAT:=log(X)
(3) log(X)
Type: Expression(Float)

---
f:ASP1(G):=retract e
(4) G
Type: Asp1(G)

---
a:SF:=1.0e-6
(5) 9.9999999999999995E-7
Type: DoubleFloat

---
b:SF:=1.0
(6) 1.
Type: DoubleFloat

---
epsabs:SF:=0.0
(7) 0.
Type: DoubleFloat

---
epsrel:SF:=1.0e-4
--R
--R
--R (8) 9.9999999999999991E-5
--R
--E 54

--S 55 of 136
omega:SF:=10*%pi
--R
--R
--R (9) 31.415926535897931
--R
--E 55

--S 56 of 136
liw:=max(400,2)
--R
--R
--R (10) 400
--R
--E 56

--S 57 of 136
lw:=max(2*liw,4)
--R
--R
--R (11) 800
--R
--E 57

--S 58 of 136
result:=d01anf(a,b,omega,2,epsabs,epsrel,liw,liw,-1,f)
--E 58

)clear all

--S 59 of 136
showArrayValues true
--R
--R
--R (1) true
--R
--E 59

--S 60 of 136
showScalarValues true
--R
--R
--R (2) true
--R
--E 60

--S 61 of 136
e:Expression Float:=sin(10*X)
--R
--R
--R (3) sin(10.0 X)
--R

Type: Expression(Float)
--E 61

--S 62 of 136
f:Asp1(G):=retract e
--R
--R
--R (4) G
--R

Type: Asp1(G)
--E 62

--S 63 of 136
a:SF:=1.0e-6
--R
--R
--R (5) 9.9999999999999995E-7
--R

Type: DoubleFloat
--E 63

--S 64 of 136
b:SF:=1.0
--R
--R
--R (6) 1.
--R

Type: DoubleFloat
--E 64

--S 65 of 136
alpha:SF:=-0.5
--R
--R
--R (7) - 0.5
--R

Type: DoubleFloat
--E 65

--S 66 of 136
beta:SF:=-0.5
--R
--R
--R (8) - 0.5
--R

Type: DoubleFloat
--E 66
--S 67 of 136  
epsabs:SF:=0.0  
--R  
--R  
--R  (9) 0.  
--R  
--E 67  

--S 68 of 136  
epsrel:SF:=1.0e-4  
--R  
--R  
--R  (10) 9.9999999999999991E-5  
--R  
--E 68  

--S 69 of 136  
key:=1  
--R  
--R  
--R  (11) 1  
--R  
--E 69  

--S 70 of 136  
liw:=200  
--R  
--R  
--R  (12) 200  
--R  
--E 70  

--S 71 of 136  
lw:=4*liw  
--R  
--R  
--R  (13) 800  
--R  
--E 71  

--S 72 of 136  
result:=d01apf(a,b,\alpha,\beta,key,epsabs,epsrel,lw,liw,-1,f)  
--E 72  

)clear all  

--S 73 of 136  
showArrayValues true  
--R
--R
--R (1) true
--R Type: Boolean
--E 73

--S 74 of 136
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 74

--S 75 of 136
e:Expression Float:=(X^2+0.01^2)^-1
--R
--R
--R 1.0
--R (3) ----------
--R 2
--R X + 0.0001
--R Type: Expression(Float)
--E 75

--S 76 of 136
f:ASP1(G):=retract e
--R
--R
--R (4) G
--R Type: Asp1(G)
--E 76

--S 77 of 136
a:SF:=-1.0
--R
--R
--R (5) - 1.
--R Type: DoubleFloat
--E 77

--S 78 of 136
b:SF:=1.0
--R
--R
--R (6) 1.
--R Type: DoubleFloat
--E 78

--S 79 of 136
c:SF:=0.5
--R
--R
--R (7) 0.5
--R
--E 79

--S 80 of 136
epsabs:SF:=0.0
--R
--R
--R (8) 0.
--R
--E 80

--S 81 of 136
epsrel:SF:=1.0e-4
--R
--R
--R (9) 9.9999999999999991E-5
--R
--E 81

--S 82 of 136
liw:=200
--R
--R
--R (10) 200
--R
--E 82

--S 83 of 136
lw:=4*200
--R
--R
--R (11) 800
--R
--E 83

--S 84 of 136
-- result:=d01aqf(a,b,c,epsabs,epsrel,lw,liw,-1,f)
--E 84

)clear all

--S 85 of 136
showArrayValues true
--R
--R
--R (1) true
--R

Type: Boolean
--E 85

--S 86 of 136
showScalarValues true
--R
--R
--R  (2) true
--R
--E 86

--S 87 of 136
e:Expression Float:=1/sqrt(X)
--R
--R
--R  1.0
--R  (3) ----
--R  +++
--R  |\X
--R
--E 87

--S 88 of 136
f:ASP1(G):=retract e
--R
--R
--R  (4) G
--R
--E 88

--S 89 of 136
a:SF:=1.0e-12
--R
--R
--R  (5) 9.9999999999999998E-13
--R
--E 89

--S 90 of 136
omega:SF:=%pi/2
--R
--R
--R  (6) 1.5707963267948966
--R
--E 90

--S 91 of 136
key:1
--R
--R
--R  (7) 1
epsabs:SF:=1.0e-3

limlst:=50

liw:=400

lw:=2*liw

result:=d01asf(a,omega,key,epsabs,limlst,lw,liw,-1,f)

)clear all

showArrayValues true
showScalarValues true
--R
--R
--R (2) true
--R
--E 98

--S 99 of 136
a:SF:=0.0
--R
--R
--R (3) 0.
--R
--E 99

--S 100 of 136
b:SF:=1.0
--R
--R
--R (4) 1.
--R
--E 100

--S 101 of 136
itype:=1
--R
--R
--R (5) 1
--R
--E 101

--S 102 of 136
n:=6
--R
--R
--R (6) 6
--R
--E 102

--S 103 of 136
kind:=0
--R
--R
--R (7) 0
--R
--E 103

--S 104 of 136
result:=d01bbf(a,b,itype,n,kind,-1)
--E 104
clear all

showArrayValues true

(1) true
Type: Boolean

showScalarValues true

(2) true
Type: Boolean

e := (4.0*X[1]*X[3]*X[3]*exp(2.0*X[1]*X[3])/((1.0+X[2]+X[4])^2))::EXPR FLOAT

(3) -------------------------------------------
Type: Expression(Float)

f:ASP4(FUNCTN):=retract e

(4) FUNCTN
Type: Asp4(FUNCTN)

ndim:=4

(5) 4
Type: PositiveInteger
a: Matrix SF := [[0.0, 0.0, 0.0, 0.0]]

b: Matrix SF := [[1.0, 1.0, 1.0, 1.0]]

alpha := 2^ndim + 2*ndim^2 + 2*ndim + 1

minpts := 1000

maxpts := 5700

d := 0.0001

lenwrk := 606
result:=d01fcf(ndim,a,b,maxpts,eps,lenwrk,minpts,-1,f)

showArrayValues true

showScalarValues true

n:=21

x:Matrix SF:=

[[0.00 ,0.04 ,0.08 ,0.12 ,0.22 ,0.26 ,0.30 ,
  0.38 ,0.39 ,0.42 ,0.45 ,0.46 ,0.60 ,0.68 ,
  0.72 ,0.73 ,0.83 ,0.85 ,0.88 ,0.90 ,1.00 ]]

[0., 3.9999999999999994E-2, 7.9999999999999998E-2, 0.12,
  0.2199999999999997, 0.26000000000000001, 0.2999999999999999, 0.38,
  0.39000000000000001, 0.41999999999999998, 0.44999999999999996, 
  0.45999999999999996, 0.59999999999999998, 0.67999999999999994,
  0.71999999999999997, 0.72999999999999998, 0.82999999999999996,
```
--R 0.84999999999999998, 0.87999999999999998, 0.89999999999999991, 1.]
--R ]
--R Type: Matrix(DoubleFloat)
--E 121

--S 122 of 136
y:Matrix SF:=
[[4.0000, 3.9936, 3.9746, 3.9432, 3.8153, 3.7467, 3.6697, _
  3.4943, 3.4719, 3.4002, 3.3264, 3.3014, 2.9412, 2.7352, _
  2.6344, 2.6094, 2.3684, 2.3222, 2.2543, 2.2099, 2.0000 ]]  
--R  
--R (5) 
--R [  
--R [4., 3.9935999999999998, 3.9745999999999997, 3.9432, 3.8152999999999997,  
  3.7466999999999997, 3.6696999999999996, 2.9411999999999998, 2.7351999999999999,  
  2.6343999999999999, 2.6093999999999999, 2.3683999999999998, 2.3221999999999996,  
  2.2542999999999997, 2.2099000000000002, 2. ]  
--R ]  
--R Type: Matrix(DoubleFloat)  
--E 122

--S 123 of 136
-- result:=d01gaf(x,y,n,-1) 
--E 123

)clear all

--S 124 of 136
showArrayValues true
--R  
--R  
--R (1) true  
--R  
--E 124

--S 125 of 136
showScalarValues true
--R  
--R  
--R (2) true  
--R  
--E 125

--S 126 of 136
ndim:=4  
--R  
--R
```
--R (3) 4
--R Type: PositiveInteger
--E 126

--S 127 of 136
e:=(4.0*X[1]*X[3]*X[3]*exp(2.0*X[1]*X[3])/((1.0+X[2]+X[4])^2))
--R
--R
--R 2.0 X X
--R    2    1 3
--R 4.0 X X %e
--R    1 3
--R (4) -------------------------------------------
--R    2    2
--R  X + (2.0 X + 2.0)X + X + 2.0 X + 1.0
--R    4    2    4    2    2
--R Type: Expression(Float)
--E 127

--S 128 of 136
f:ASP4(FUNCTN):=retract e
--R
--R
--R (5) FUNCTN
--R Type: Asp4(FUNCTN)
--E 128

--S 129 of 136
a:Matrix SF:=[[0.0,0.0,0.0,0.0,0.0]]
--R
--R
--R (6) [0. 0. 0. 0.]
--R Type: Matrix(DoubleFloat)
--E 129

--S 130 of 136
b:Matrix SF:=[[1.0,1.0,1.0,1.0,1.0]]
--R
--R
--R (7) [1. 1. 1. 1.]
--R Type: Matrix(DoubleFloat)
--E 130

--S 131 of 136
maxcls:=20000
--R
--R
--R (8) 20000
--R Type: PositiveInteger
--E 131
eps:=0.01
Type: Float

lenwrk:=500
Type: PositiveInteger

wrk:Matrix SF:=new(1,lenwrk,0.0);
Type: Matrix(DoubleFloat)

mincls:=1000
Type: PositiveInteger

result:=d01gbf(ndim,a,b,maxcls,eps,lenwrk,mincls,wrk,-1,f)

)spool

)lisp (bye)
d01AgentsPackage is a package of numerical agents to be used
to investigate attributes of an input function so as to decide the
measure of an appropriate numerical integration routine.

It contains functions rangeIsFinite to test the input range and
functionIsContinuousAtEndPoints to check for continuity at
the end points of the range.

See Also:
o )show d01AgentsPackage

---

d01AgentsPackage (D01AGNT)

Exports:

- changeName
- commaSeparate
- df2st
- functionIsContinuousAtEndPoints
- functionIsOscillatory
- gethi
- getlo
- ldf2lst
- problemPoints
- rangeIsFinite
- sdf2lst
- singularitiesOf

--- package D01AGNT d01AgentsPackage ---

)abbrev package D01AGNT d01AgentsPackage
++ Author: Brian Dupee
++ Date Created: March 1994
++ Date Last Updated: December 1997
++ Description:
++ \axiomType{d01AgentsPackage} is a package of numerical agents to be used
++ to investigate attributes of an input function so as to decide the
++ \axiomFun{measure} of an appropriate numerical integration routine.
++ It contains functions \axiomFun{rangeIsFinite} to test the input range and
++ \axiomFun{functionIsContinuousAtEndPoints} to check for continuity at
++ the end points of the range.
d01AgentsPackage(): E == I where

  EF2 ==> ExpressionFunctions2
  EFI ==> Expression Fraction Integer
  FI ==> Fraction Integer
  LEDF ==> List Expression DoubleFloat
  KEDF ==> Kernel Expression DoubleFloat
  EEDF ==> Equation Expression DoubleFloat
  EDF ==> Expression DoubleFloat
  PDF ==> Polynomial DoubleFloat
  LDF ==> List DoubleFloat
  SDF ==> Stream DoubleFloat
  DF ==> DoubleFloat
  F ==> Float
  ST ==> String
  LST ==> List String
  SI ==> SingleInteger
  SOCDF ==> Segment OrderedCompletion DoubleFloat
  OCDF ==> OrderedCompletion DoubleFloat
  OCEDF ==> OrderedCompletion Expression DoubleFloat
  EOCEFI ==> Equation OrderedCompletion Expression Fraction Integer
  OCEFI ==> OrderedCompletion Expression Fraction Integer
  OCFI ==> OrderedCompletion Fraction Integer
  NIA ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
  INT ==> Integer
  CTYPE ==> Union(continuous: "Continuous at the end points",
      lowerSingular: "There is a singularity at the lower end point",
      upperSingular: "There is a singularity at the upper end point",
      bothSingular: "There are singularities at both end points",
      notEvaluated: "End point continuity not yet evaluated")
  RTYPE ==> Union(finite: "The range is finite",
      lowerInfinite: "The bottom of range is infinite",
      upperInfinite: "The top of range is infinite",
      bothInfinite: "Both top and bottom points are infinite",
      notEvaluated: "Range not yet evaluated")
  STYPE ==> Union(str:SDF,
      notEvaluated: "Internal singularities not yet evaluated")
  ATT ==> Record(endPointContinuity:CTYPE,
      singularitiesStream:STYPE,range:RTYPE)
  ROA ==> Record(key:NIA,entry:ATT)

  E ==> with

  rangeIsFinite : NIA -> RTYPE
      ++ rangeIsFinite(args) tests the endpoints of \spad{args.range} for
      ++ infinite end points.
  functionIsContinuousAtEndPoints: NIA -> CTYPE
      ++ functionIsContinuousAtEndPoints(args) uses power series limits
      ++ to check for problems at the end points of the range of \spad{args}.
  getlo : SOCDF -> DF
      ++ getlo(x) gets the \axiomType{DoubleFloat} equivalent of
++ the first endpoint of the range \axiom{x}
gethi : SOCDF \to DF
++ gethi(x) gets the \axiomType{DoubleFloat} equivalent of
++ the second endpoint of the range \axiom{x}
functionIsOscillatory:NIA \to F
++ functionIsOscillatory(a) tests whether the function \spad{a.fn}
++ has many zeros of its derivative.
problemPoints: (EDF, Symbol, SOCDF) \to \text{List DF}
++ problemPoints(f,var,range) returns a list of possible problem points
++ by looking at the zeros of the denominator of the function if it
++ can be retracted to \axiomType{Polynomial DoubleFloat}.
singularitiesOf:NIA \to SDF
++ singularitiesOf(args) returns a list of potential
++ singularities of the function within the given range
df2st:DF \to \text{String}
++ df2st(n) coerces a \axiomType{DoubleFloat} to \axiomType{String}
ldf2lst:LDF \to \text{LST}
++ ldf2lst(ln) coerces a \text{List of \axiomType{DoubleFloat}} to
++ \axiomType{List String}
sdf2lst:SDF \to \text{LST}
++ sdf2lst(ln) coerces a \text{Stream of \axiomType{DoubleFloat}} to
++ \axiomType{List String}
commaSeparate:LST \to \text{ST}
++ commaSeparate(l) produces a comma separated string from a
++ list of strings.
changeName:(Symbol,Symbol,Result) \to Result
++ changeName(s,t,r) changes the name of item \axiom{s} in \axiom{r}
++ to \axiom{t}.

I ==> ExpertSystemContinuityPackage add
import ExpertSystemToolsPackage
import ExpertSystemContinuityPackage

-- local functions
ocdf2ocefi : OCDF \to OCEFI
rangeOfArgument : (KEDF, NIA) \to DF
continuousAtPoint? : (EFI,EOCEFI) \to Boolean
rand:(SOCDF,INT) \to LDF
eval:(EDF.Symbol,LDF) \to LDF
numberOfSignChanges:LDF \to INT
rangeIsFiniteFunction:NIA \to RTYPE
functionIsContinuousAtEndPointsFunction:NIA \to CTYPE

changeName(s:Symbol,t:Symbol,r:Result):Result ==
a := remove!(s,r)$Result
a case Any =>
  insert!([t,a],r)$Result
  r
  r
CHAPTER 5. CHAPTER D

\begin{verbatim}
commaSeparate(l:LST):ST ==
  empty?(l$LST) => ""
  one?(#(l)) => concat(l)$ST
  (#(l) = 1) => concat(l)$ST
  f := first(l)$LST
  t := [concat([", ",l.i]$ST for i in 2..#(l)]
  concat(f,concat(t)$ST)$ST

rand(seg:SGCDF,n:INT):LDF ==
  -- produced a sorted list of random numbers in the given range
  l:DF := getlo seg
  s:DF := (gethi seg) - l
  seed:INT := random()$INT
  dseed:DF := seed :: DF
  r:LDF := [(((random(seed)$INT) :: DF)*s/dseed + l) for i in 1..n]
  sort(r)$LDF

eval(f:EDF,var:Symbol,l:LDF):LDF ==
  empty?(l$LDF) => [0$DF]
  ve := var::EDF
  [retract(eval(f,equation(ve,u::EDF)$EEDF)$EDF)@DF for u in l]

numberOfSignChanges(l:LDF):INT ==
  -- calculates the number of sign changes in a list
  a := 0$INT
  empty?(l$LDF) => 0
  for i in 2..# l repeat
    if negative?(l.i*l.(i-1)) then
      a := a + 1
  a

rangeOfArgument(k: KEDF, args:NIA): DF ==
  Args := copy args
  Args.fn := arg := first(argument(k)$KEDF)$LEDF
  functionIsContinuousAtEndPoints(Args) case continuous =>
    r:SGCDF := args.range
    low:EDF := (getlo r) :: EDF
    high:EDF := (gethi r) :: EDF
    eql := equation(a := args.var :: EDF, low)$EEDF
    eqh := equation(a, high)$EEDF
    e1 := (numeric(eval(arg,eql)$EDF)$Numeric(DF)) :: DF
    e2 := (numeric(eval(arg,eqh)$EDF)$Numeric(DF)) :: DF
    e2-e1
  0$DF

ocdf2ocefi(r:OCDF):OCEFI ==
  finite?(r$OCDF) => (edf2efi(((retract(r)@DF)$OCDF)::EDF))::OCEFI
  r pretend OCEFI
\end{verbatim}
continuousAtPoint?(f: EFI,e: EOCI): Boolean ==
  l := limit(f,e)$PowerSeriesLimitPackage(FI,EFI) case OCEFI =>
  finite?(l :: OCEFI)
-- if the left hand limit equals the right hand limit, or if neither
-- side has a limit at this point, the return type of limit() is
-- Union(Ordered Completion Expression Fraction Integer,"failed")
false

-- exported functions

rangeIsFiniteFunction(args:NIA): RTYPE ==
  -- rangeIsFinite(x) tests the endpoints of x.range for infinite
  -- end points.
  --
  --   [ -inf, inf ] => 4
  --   [ x , inf ] => 3
  --   [ -inf, x ] => 1
  --   [ x , y ] => 0
  fr: SI := (3::SI * whatInfinity(hi(args.range))$OCDF
               - whatInfinity(lo(args.range))$OCDF)
  fr = 0 => "The range is finite"
  fr = 1 => "The bottom of range is infinite"
  fr = 3 => "The top of range is infinite"
  fr = 4 => "Both top and bottom points are infinite"
  error("rangeIsFinite",["this is not a valid range"])$ErrorFunctions

rangeIsFinite(args:NIA): RTYPE ==
  nia := copy args
  (t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
    s := coerce(t)$ATT
    s.range case notEvaluated =>
      s.range := rangeIsFiniteFunction(nia)
      r: ROA := [nia,s]
      insert!(r)$IntegrationFunctionsTable
      s.range
      s.range
    a: ATT := [["End point continuity not yet evaluated"],
               ["Internal singularities not yet evaluated"],
               e:=rangeIsFiniteFunction(nia)]
    r: ROA := [nia,a]
    insert!(r)$IntegrationFunctionsTable
    e

functionIsContinuousAtEndPointsFunction(args:NIA): CTYPE ==
  v := args.var :: EFI :: OCEFI
  high: OCEFI := ocdf2ocefi(hi(args.range))
  low: OCEFI := ocdf2ocefi(lo(args.range))
  f := edf2efi(args.fn)
l: Boolean := continuousAtPoint?(f, equation(v, low)$EOCI) 
  h: Boolean := continuousAtPoint?(f, equation(v, high)$EOCI)
l and h => ["Continuous at the end points"]
l => ["There is a singularity at the upper end point"]
h => ["There is a singularity at the lower end point"]
["There are singularities at both end points"]

functionIsContinuousAtEndPoints(args:NIA): CTYPE ==
nia := copy args
(t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
s := coerce(t)$ATT
s.endPointContinuity case notEvaluated =>
s.endPointContinuity := functionIsContinuousAtEndPointsFunction(nia)
r:ROA := [nia,s]
insert!(r)$IntegrationFunctionsTable
s.endPointContinuity

a:ATT := [e:=functionIsContinuousAtEndPointsFunction(nia),
            ["Internal singularities not yet evaluated"],
            ["Range not yet evaluated"]]
r:ROA := [nia,a]
insert!(r)$IntegrationFunctionsTable
e

functionIsOscillatory(a:NIA):F ==
args := copy a
k := tower(numerator args.fn)$EDF
p:F := pi()$F
for i in 1..# k repeat
    is?(ker := k.i, sin :: Symbol) =>
        ra := convert(rangeOfArgument(ker,args))@F
        ra > 2*p => return (ra/p)
    is?(ker, cos :: Symbol) =>
        ra := convert(rangeOfArgument(ker,args))@F
        ra > 2*p => return (ra/p)

l:LDF := rand(args.range,30)
l := eval(args.fn,args.var,l)
numberOfSignChanges(l) :: F

singularitiesOf(args:NIA):SDF ==
nia := copy args
(t := showAttributes(nia)$IntegrationFunctionsTable) case ATT =>
s:ATT := coerce(t)$ATT
p:STYPE := s.singularitiesStream
p case str => p.str
e:SDF := singularitiesOf(nia.fn,[nia.var],nia.range)
if not empty?(e) then
    if less?(e,10)$SDF then extend(e,10)$SDF
s.singularitiesStream := [e]
r:ROA := [nia,s]
insert!(r)$IntegrationFunctionsTable
e

\(e:=\text{singularitiesOf}(\text{nia.fn},[\text{nia.var}],\text{nia.range})\)
if not empty?(e) then
  if less?(e,10)\$\text{SDF} then extend(e,10)\$\text{SDF}

\(\text{a:ATT} := \left[\text{"End point continuity not yet evaluated"},[e],\right.\text{["Range not yet evaluated"]}\right]\)

\(\text{r:ROA} := \text{[nia,a]}\)
insert!(r)\$\text{IntegrationFunctionsTable}
e

---

— D01AGNT.dotabb —

"\text{D01AGNT}" \[\text{color="\text{FF4488}",href=bookvol10.4.pdf\#nameddest=D01AGNT"]\]
"\text{FS}" \[\text{color="\text{4488FF}",href=bookvol10.2.pdf\#nameddest=FS"]\]
"\text{D01AGNT}" \rightarrow "\text{FS}"

---

package D01WGTS d01WeightsPackage

--- d01WeightsPackage.input ---

)set break resume
)sys rm -f d01WeightsPackage.output
)spool d01WeightsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show d01WeightsPackage
--R
--R d01WeightsPackage is a package constructor
--R Abbreviation for d01WeightsPackage is D01WGTS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for D01WGTS
--R
--R-------------------- Operations ------------------------------
--R exprHasAlgebraicWeight : Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat)),abserr: DoubleFloat,relerr: DoubleFloat) -> Union(List(DoubleFloat),"failed")
--R exprHasLogarithmicWeights : Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat)),abserr: DoubleFloat,relerr: DoubleFloat) -> Integer
--R exprHasWeightCosWXorSinWX : Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat)),abserr: DoubleFloat,relerr: DoubleFloat) -> Union(Record(op: BasicOperator,w: DoubleFloat),"failed")
--- E 1

)spool
)lisp (bye)

---

— d01WeightsPackage.help —

====================================================================
d01WeightsPackage examples
====================================================================

d01WeightsPackage is a package for functions used to investigate whether
a function can be divided into a simpler function and a weight function.
The types of weights investigated are those giving rise to end-point
singularities of the algebraico-logarithmic type, and trigonometric weights.

See Also:
o )show d01WeightsPackage

---

d01WeightsPackage (D01WGTS)

Exports:
exprHasAlgebraicWeight exprHasLogarithmicWeights exprHasWeightCosWXorSinWX

— package D01WGTS d01WeightsPackage —

)abbrev package D01WGTS d01WeightsPackage
++ Author: Brian Dupee
++ Date Created: July 1994
++ Date Last Updated: January 1998
++ Description:
++ \texttt{d01WeightsPackage} is a package for functions used to investigate
++ whether a function can be divided into a simpler function and a weight
++ function. The types of weights investigated are those giving rise to
++ end-point singularities of the algebraico-logarithmic type, and
++ trigonometric weights.

\texttt{d01WeightsPackage()}: \texttt{E == I} where

\texttt{LEDF} ==> \texttt{List Expression DoubleFloat}
\texttt{KEDF} ==> \texttt{Kernel Expression DoubleFloat}
\texttt{LKEDF} ==> \texttt{List Kernel Expression DoubleFloat}
\texttt{EDF} ==> \texttt{Expression DoubleFloat}
\texttt{PDF} ==> \texttt{Polynomial DoubleFloat}
\texttt{FI} ==> \texttt{Fraction Integer}
\texttt{LDF} ==> \texttt{List DoubleFloat}
\texttt{DF} ==> \texttt{DoubleFloat}
\texttt{SOCDF} ==> \texttt{Segment OrderedCompletion DoubleFloat}
\texttt{OCDF} ==> \texttt{OrderedCompletion DoubleFloat}
\texttt{NIA} ==> \texttt{Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)}
\texttt{INT} ==> \texttt{Integer}
\texttt{BOP} ==> \texttt{BasicOperator}
\texttt{URBODF} ==> \texttt{Union(Record(op:BasicOperator,w:DF),"failed")}
\texttt{LURBODF} ==> \texttt{List(Union(Record(op:BasicOperator,w:DF), "failed"))}

\texttt{E} ==> with

\texttt{exprHasWeightCosWXorSinWX:NIA -> URBODF}
++ \texttt{exprHasWeightCosWXorSinWX} looks for trigonometric
++ weights in an expression of the form \texttt{\cos \omega x} or
++ \texttt{\sin \omega x}, returning the value of \texttt{\omega}
++ (\texttt{\not= 1}) and the operator.
\texttt{exprHasAlgebraicWeight:NIA -> Union(LDF,\texttt{"failed")}
++ \texttt{exprHasAlgebraicWeight} looks for algebraic weights
++ giving rise to singularities of the function at the end-points.
\texttt{exprHasLogarithmicWeights:NIA -> INT}
++ \texttt{exprHasLogarithmicWeights} looks for logarithmic weights
++ giving rise to singularities of the function at the end-points.

\texttt{I} ==> add
\texttt{score:(EDF,EDF) -> FI}
\texttt{kernelIsLog:KEDF -> Boolean}
\texttt{functionIsPolynomial?:EDF -> Boolean}
\texttt{functionIsNthRoot?:(EDF,EDF) -> Boolean}
\texttt{functionIsQuotient:EDF -> Union(EDF,\texttt{"failed")}
\texttt{findCommonFactor:LEDF -> Union(LEDF,\texttt{"failed")}
\texttt{findAlgebraicWeight:(NIA,EDF) -> Union(DF,\texttt{"failed")}

exprHasListOfWeightsCosXorSinX:(EDF,Symbol) -> LURBODF
exprOfFormCosXorSinX:(EDF,Symbol) -> URBODF
bestWeight:LURBODF -> URBODF
weightIn?:(UBORDF, LURBODF) -> Boolean
inRest?:(EDF,LEDF)->Boolean
factorIn?:(EDF,LEDF)->Boolean
voo?:(EDF,EDF)->Boolean

kernelIsLog(k:KEDF):Boolean ==
  (name k = (log :: Symbol))@Boolean

factorIn?(a:EDF,l:LEDF):Boolean ==
  for i in 1..# l repeat
    (a = l.i)@Boolean => return true
  false

voo?(b:EDF,a:EDF):Boolean ==
  (voo:=isTimes(b)) case LEDF and factorIn?(a,voo)

inRest?(a:EDF,l:LEDF):Boolean ==
  every?(x+->voo?(x,a) ,l)

findCommonFactor(l:LEDF):Union(LEDF,"failed") ==
  empty?(l)$LEDF => "failed"
  f := first(l)$LEDF
  r := rest(l)$LEDF
  (t := isTimes(f)$EDF) case LEDF =>
    pos:=select(x+->inRest?(x,r),t)
    empty?(pos) => "failed"
    pos
  "failed"

exprIsLogarithmicWeight(f:EDF,Var:EDF,a:EDF,b:EDF):INT ==
  ans := 0$INT
  k := tower(f)$EDF
  if := select(kernelIsLog,k)$LKEDF
  empty?(if)$LKEDF => ans
  for i in 1..# if repeat
    arg := argument if.i
    if (arg.1 = (Var - a)) then
      ans := ans + 1
    else if (arg.1 = (b - Var)) then
      ans := ans + 2
    ans

exprHasLogarithmicWeights(args:NIA):INT ==
  ans := 0$INT
  a := getlo(args.range)$d01AgentsPackage :: EDF
  b := gethi(args.range)$d01AgentsPackage :: EDF
  Var := args.var :: EDF
(l := isPlus numerator args.fn) case LEDF =>
(cf := findCommonFactor l) case LEDF =>
  for j in 1..# cf repeat
    ans := ans + exprIsLogarithmicWeight(cf.j, Var, a, b)
  ans
  ans := ans + exprIsLogarithmicWeight(args.fn, Var, a, b)

functionIsQuotient(expr: EDF): Union(EDF,"failed") ==
  (k := mainKernel expr) case KEDF =>
    expr = inv(f := k :: KEDF :: EDF)$EDF => f
  -- one?(numerator expr) => denominator expr
  (numerator expr = 1) => denominator expr
  "failed"
  "failed"

functionIsPolynomial?(f: EDF): Boolean ==
  (retractIfCan(f)@Union(PDF,"failed"))$EDF case PDF

functionIsNthRoot?(f: EDF, e: EDF): Boolean ==
  (m := mainKernel f) case "failed" => false
  --
  (
    (one?(# (kernels f)))
    
    (# (kernels f)) = 1
    and (name operator m = (nthRoot :: Symbol))@Boolean
    and (((argument m).1 = e)@Boolean)
  )

score(f: EDF, e: EDF): FI ==
  ans := 0$FI
  (t := isTimes f) case LEDF =>
    for i in 1..# t repeat
      ans := ans + score(t.i, e)
  ans
  (q := functionIsQuotient f) case EDF =>
    ans := ans - score(q, e)
  functionIsPolynomial? f =>
    g: EDF := f/e
    if functionIsPolynomial? g then
      ans := 1+score(g, e)
    else
      ans
  (l := isPlus f) case LEDF =>
    (cf := findCommonFactor l) case LEDF =>
      factor := 1$EDF
      for i in 1..# cf repeat
        factor := factor*cf.i
      ans := ans + score(f/factor, e) + score(factor, e)
  ans
  functionIsNthRoot?(f, e) =>
    (p := isPower f) case "failed" => ans
    exp := p.exponent
m := mainKernel f
m case KEDF =>
  arg := argument m
a:INT := (retract(arg.2)@INT)$EDF
exp / a
ans
ans

findAlgebraicWeight(args:NIA,e:EDF):Union(DF,"failed") ==
  zero?(s := score(args.fn,e)) => "failed"
s :: DF

exprHasAlgebraicWeight(args:NIA):Union(LDF,"failed") ==
  (f := functionIsContinuousAtEndPoints(args)$d01AgentsPackage)
case continuous =>"failed"
Var := args.var :: EDF
a := getlo(args.range)$d01AgentsPackage :: EDF
b := gethi(args.range)$d01AgentsPackage :: EDF
A := Var - a
B := b - Var
f case lowerSingular =>
  (h := findAlgebraicWeight(args,A)) case "failed" => "failed"
  [h,0]
f case upperSingular =>
  (g := findAlgebraicWeight(args,B)) case "failed" => "failed"
  [0,g]
h := findAlgebraicWeight(args,A)
g := findAlgebraicWeight(args,B)
r := (h case "failed")
s := (g case "failed")
(r) and (s) => "failed"
r => [0,coerce(g)@DF]
s => [coerce(h)@DF,0]
[coerce(h)@DF,coerce(g)@DF]

exprOfFormCosWXorSinWX(f:EDF,var:Symbol): URBODF ==
  l:LKEDF := kernels(f)$EDF
-- one?((# l)$LKEDF)$INT =>
  # l = 1 =>
    a:LEDF := argument(e:KEDF := first(l)$LKEDF)$KEDF
    empty?(a) => "failed"
m:Union(LEDF,"failed") := isTimes(first(a)$LEDF)$EDF
m case LEDF => -- if it is a list, it will have at least two elements
  is?(second(m)$LEDF,var)$EDF =>
    omega:DF := retract(first(m)$LEDF)@DF
    o:BOP := operator(n:Symbol:=name(e)$KEDF)$BOP
    (n = cos$Symbol)@Boolean => [o,omega]
    (n = sin$Symbol)@Boolean => [o,omega]
    "failed"
  "failed"
"failed"
"failed"

exprHasListOfWeightsCosWXorSinWX(f:EDF,var:Symbol): LURBODF ==
  (e := isTimes(f)$EDF) case LEDF =>
    [exprOfFormCosWXorSinWX(u,var) for u in e]
empty?(k := kernels f) => ["failed"]
((first(k)::EDF) = f) =>
    [exprOfFormCosWXorSinWX(f,var)]
["failed"]

bestWeight(l:LURBODF): URBODF ==
  empty?(l)$LURBODF => "failed"
  best := first(l)$LURBODF -- best is first in list
  empty?(rest(l)$LURBODF) => best
  for i in 2..# l repeat -- unless next is better
    r:URBODF := l.i
    if r case "failed" then leave
    else if best case "failed" then
      best := r
    else if r.w > best.w then
      best := r
  best

weightIn?(weight:URBODF,listOfWeights:LURBODF):Boolean ==
  n := # listOfWeights
  for i in 1..n repeat -- cycle through list
    (weight = listOfWeights.i)$Boolean => return true -- return when found
  false

exprHasWeightCosWXorSinWX(args:NIA):URBODF ==
  ans := empty()$LURBODF
  f:EDF := numerator(args.fn)$EDF
  (t:Union(LEDF,"failed") := isPlus(f)) case "failed" =>
    bestWeight(exprHasListOfWeightsCosWXorSinWX(f,args.var))
  if t case LEDF then
    e1 := first(t)$LEDf
    le1:LURBODF := exprHasListOfWeightsCosWXorSinWX(e1,args.var)
    le1 := [u for u in le1 | (not (u case "failed"))]
    empty?(le1)$LURBODF => "failed"
    test := true
    for i in 1..# le1 repeat
      le1i:URBODF := le1.i
      for j in 2..# t repeat
        if test then
          tj:URBODF := exprHasListOfWeightsCosWXorSinWX(t.j,args.var)
          test := weightIn?(le1i,tj)
        if test then
          ans := concat([le1i],ans)
    bestWeight ans
else "failed"

---

--- D01WGTS.dotabb ---

"D01WGTS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D01WGTS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D01WGTS" -> "FS"

---

package D02AGNT d02AgentsPackage

--- d02AgentsPackage.input ---

)set break resume
)sys rm -f d02AgentsPackage.output
)spool d02AgentsPackage.output
)set message test on
)set message auto off
)
clear all

--S 1 of 137
)show d02AgentsPackage
--)R
--)R d02AgentsPackage is a package constructor
--)R Abbreviation for d02AgentsPackage is D02AGNT
--)R This constructor is exposed in this frame.
--)R Issue )edit bookvol10.4.pamphlet to see algebra source code for D02AGNT
--)R
)--R--------------------------------------------- Operations ---------------------------------------------
--)R accuracyIF : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(DoubleFloat),intvals: List(DoubleFloat),g: Expression(DoubleFloat),abserr: DoubleFloat,relerr: DoubleFloat) -> Float
--)R combineFeatureCompatibility : (Float,List(Float)) -> Float
--)R eval : (Matrix(Expression(DoubleFloat)),List(Symbol),Vector(Expression(DoubleFloat))) -> Matrix(Expression(DoubleFloat))
--)R expenseOfEvaluationIF : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(DoubleFloat),intvals: List(DoubleFloat),g: Expression(DoubleFloat),abserr: DoubleFloat,relerr: DoubleFloat) -> Float
--)R intermediateResultsIF : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(DoubleFloat),intvals: List(DoubleFloat),g: Expression(DoubleFloat),abserr: DoubleFloat,relerr: DoubleFloat) -> Float
--)R jacobian : (Vector(Expression(DoubleFloat)),List(Symbol)) -> Matrix(Expression(DoubleFloat))
--)R sparsityIF : Matrix(Expression(DoubleFloat)) -> Float
--)R stiffnessAndStabilityFactor : Matrix(Expression(DoubleFloat)) -> Record(stiffnessFactor: Float)
--)R stiffnessAndStabilityOfODEIF : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(DoubleFloat),intvals: List(DoubleFloat),g: Expression(DoubleFloat),abserr: DoubleFloat,relerr: DoubleFloat) -> Record(stiffnessFactor: Float)
--)R systemSizeIF : Record(xinit: DoubleFloat,xend: DoubleFloat,fn: Vector(Expression(DoubleFloat)),yinit: List(DoubleFloat),intvals: List(DoubleFloat),g: Expression(DoubleFloat),abserr: DoubleFloat,relerr: DoubleFloat) -> Float
clear all

showArrayValues true

(1) true
Type: Boolean

showScalarValues true

(2) true
Type: Boolean

xend:SF:=8.0

(3) 8.
Type: DoubleFloat

n:=3

(4) 3
Type: PositiveInteger

irelab:=0

(5) 0
Type: NonNegativeInteger

x:SF:=0.0

(6) 0.
```
CHAPTER 5. CHAPTER D

--R
--E 7

--S 8 of 137
y:Matrix SF:= [[0.0, 0.5, \pi*0.2]]
--R
--R (7) [0. 0.5 0.62831853071795862]
--R Type: Matrix(DoubleFloat)
--E 8

--S 9 of 137
tol:SF:=0.0001
--R
--R (8) 9.9999999999999991E-5
--R Type: DoubleFloat
--E 9

--S 10 of 137
vef:Vector Expression Float:=
[\tan(Y[3]), -0.032tan(Y[3])/Y[2], -0.02*Y[2]/cos(Y[3]), -0.032/(Y[2]^2)]
--R
--R (9) [\tan(Y), -0.032 \cos(Y) \tan(Y) - 0.02 Y, -0.032
--R 3 3 2 0.032
--R (9) [\tan(Y), -\tan(Y) \cos(Y) Y, 2]
--R 3 2 Y
--R Type: Vector(Expression(Float))
--E 10

--S 11 of 137
fcn:Asp7(FCN):= retract vef
--R
--R (10) FCN
--R Type: Asp7(FCN)
--E 11

--S 12 of 137
vm:Vector MachineFloat:= [1,2,3,4,5,6,7,8]
--R
--R (11) [1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0]
--R Type: Vector(MachineFloat)
--E 12
```
--S 13 of 137
output:Asp8(OUTPUT):= coerce vm
--R
--R
--R (12) OUTPUT
--R Type: Asp8(OUTPUT)
--E 13

--S 14 of 137
-- result:=d02bbf(xend,# vm,n,irelab,x,y,tol,-1,fcn,output)
--E 14

clear all

--S 15 of 137
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 15

--S 16 of 137
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 16

--S 17 of 137
xend:SF:=10.0
--R
--R
--R (3) 10.
--R Type: DoubleFloat
--E 17

--S 18 of 137
n:=3
--R
--R
--R (4) 3
--R Type: PositiveInteger
--E 18

--S 19 of 137
irelab:=0
--R
--R
hmax:SF:=0.0

x:SF:=0.0

y:Matrix SF:=[[0.5, 0.5, %pi*0.2]]

tol:SF:=0.0001

ef:Expression Float:=1.0*Y[1]:EXPR FLOAT

g:Asp9(G):=retract ef
--R Type: Asp9(G)
--E 25

--S 26 of 137

vef:Vector Expression Float:=
    [tan(Y[3]), -0.032*tan(Y[3])/Y[2] - 0.02*Y[2]/cos(Y[3]), -0.032/(Y[2]^2) ]
--R
--R
--R 2
--R - 0.032 cos(Y) tan(Y) - 0.02 Y
--R 3 3 2 0.032
--R (12) [tan(Y),---------------------------------,- -----
--R 3 Y cos(Y) 2
--R 2 3 Y
--R 2
--R Type: Vector(Expression(Float))
--E 26

--S 27 of 137

fcn:Asp7(FCN):= retract vef
--R
--R
--R (13) FCN
--R
--E 27

--S 28 of 137

-- result:=d02bhf(xend,n,irelab,hmax,x,y,tol,-1,g,fcn)
--E 28
)
clear all

--S 29 of 137

showArrayValues true
--R
--R
--R (1) true
--R
--E 29

--S 30 of 137

showScalarValues true
--R
--R
--R (2) true
--R
--E 30

--S 31 of 137

xend:SF:=10.0
--R
--R
--R  (3) 10.
--E 31

--S 32 of 137
n:=3
--R
--R
--R  (4) 3
--R
--E 32

--S 33 of 137
tol:SF:=0.0001
--R
--R
--R  (5) 9.9999999999999991E-5
--R
--E 33

--S 34 of 137
relabs:="D"
--R
--R
--R  (6) "D"
--R
--E 34

--S 35 of 137
x:SF:=0.0
--R
--R
--R  (7) 0.
--R
--E 35

--S 36 of 137
y:Matrix SF:=[[0.5 ,0.5 ,%pi*0.2 ]]
--R
--R
--R  (8) [0.5 0.5 0.62831853071795862]
--R
--E 36

--S 37 of 137
ef:Expression Float:=Y[1]: EXPR FLOAT
--R
--R
--R  (9) Y
--R      1
--R                         Type: Expression(Float)
--E 37

--S 38 of 137
g:Asp9(G):=retract ef
--R
--R    (10) G
--R                         Type: Asp9(G)
--E 38

--S 39 of 137
vef:Vector Expression Float:=
[\tan(Y[3]), -0.032*\tan(Y[3])/Y[2], -0.02*Y[2]/\cos(Y[3]), -0.032/(Y[2]^2) ]
--R
--R
--R     2
--R     - 0.032 \cos(Y) \tan(Y) - 0.02 Y
--R     3 \quad 3 \quad 2 \quad 0.032
--R (11) [\tan(Y ),-----------------------------,-- ----]
--R     3 \quad Y \cos(Y ) \quad 2
--R     2 \quad 3 \quad Y
--R)
--R                         Type: Vector(Expression(Float))
--E 39

--S 40 of 137
fcn:Asp7(FCN):= retract vef
--R
--R
--R    (12) FCN
--R                         Type: Asp7(FCN)
--E 40

--S 41 of 137
vm:Vector MachineFloat:= [2,4,6,8]
--R
--R
--R    (13) [2.0,4.0,6.0,8.0]
--R                         Type: Vector(MachineFloat)
--E 41

--S 42 of 137
output:Asp8(OUTPUT):=coerce vm
--R
--R
--R    (14) OUTPUT
--R                         Type: Asp8(OUTPUT)
```plaintext
result:=d02cjf(xend,# vm,n,tol,relabs,x,y,-1,g,fcn,output)

)clear all

showArrayValues true
(1) true
Type: Boolean

showScalarValues true
(2) true
Type: Boolean

xend:SF:= 10.0
(3) 10.
Type: DoubleFloat

n:=3
(4) 3
Type: PositiveInteger

relabs:="D"
(5) "D"
Type: String

iw:=95
```
--R
--R
--R (6) 95
--R Type: PositiveInteger
--E 49

--S 50 of 137
x:SF:=0.0
--R
--R
--R (7) 0.
--R Type: DoubleFloat
--E 50

--S 51 of 137
y:Matrix SF:=[[1.0,0.0,0.0]]
--R
--R
--R (8) [1.0,0.0,0.0]
--R Type: Matrix(DoubleFloat)
--E 51

--S 52 of 137
tol:=0.0001
--R
--R
--R (9) 0.0001
--R Type: Float
--E 52

--S 53 of 137
ef:Expression Float:=(Y[1]::EXPR FLOAT)-0.9
--R
--R
--R (10) Y - 0.9
--R 1
--R Type: Expression(Float)
--E 53

--S 54 of 137
g:Asp9(G):=retract ef
--R
--R
--R (11) G
--R Type: Asp9(G)
--E 54

--S 55 of 137
vef:Vector Expression Float:=
[-0.04*Y[1]+1.0E4*Y[2]*Y[3],...
\[ -R \]
\[ (12) \]
\[ -R \]
\[ 2 \]
\[ 10000.0 Y Y - 0.04 Y, \]
\[ -R \]
\[ 23 \]
\[ 1 \]
\[ 1 \]
\[ 2 \]
\[ 2 \]
\[ 30000000.0 Y \]
\[ -R \]
\[ 2 \]
\[ Type: Vector(Expression(Float)) \]
\[ -E 55 \]

\[ fcn: ASP7(FCN) := retract vef \]
\[ -R \]
\[ -R \]
\[ (13) FCN \]
\[ -R \]
\[ Type: Asp7(FCN) \]
\[ -E 56 \]

\[ pederv: Asp31(PEDERV) := retract vef \]
\[ -R \]
\[ -R \]
\[ (14) PEDERV \]
\[ -R \]
\[ Type: Asp31(PEDERV) \]
\[ -E 57 \]

\[ vm: Vector MachineFloat := [2, 4, 6, 8] \]
\[ -R \]
\[ -R \]
\[ (15) [2.0, 4.0, 6.0, 8.0] \]
\[ -R \]
\[ Type: Vector(MachineFloat) \]
\[ -E 58 \]

\[ output: Asp8(OUTPUT) := coerce vm \]
\[ -R \]
\[ -R \]
\[ (16) OUTPUT \]
\[ -R \]
\[ Type: Asp8(OUTPUT) \]
\[ -E 59 \]

\[ result := do2ejf(xend, # vm, n, relabs, iw, x, y, tol, -1, g, fcn, pederv, output) \]
\[ -E 60 \]
clear all

showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean

u: Matrix := [[0 ,10 ],[0 ,1],[0 ,0 ]]

+0. 10.+

| |

(3) |0. 1. |

| |

+0. 0. +

Type: Matrix(DoubleFloat)

v: Matrix := [[0.0 ,1.0 ],[0.0 ,0.0],[1.0 ,1.0 ]]

+0. 1.+

| |

(4) |0. 0. |

| |

+1. 1.+

Type: Matrix(DoubleFloat)

n:=3

(5) 3

Type: PositiveInteger
--S 66 of 137
a:SF:=0.0
--R
--R
--R (6) 0.
--R
--E 66

Type: DoubleFloat

--S 67 of 137
b:SF:=10.0
--R
--R
--R (7) 10.
--R
--E 67

Type: DoubleFloat

--S 68 of 137
tol:=1.0e-3
--R
--R
--R (8) 0.001
--R
--E 68

Type: Float

--S 69 of 137
mnp:=64
--R
--R
--R (9) 64
--R
--E 69

Type: PositiveInteger

--S 70 of 137
lw:=3056
--R
--R
--R (10) 3056
--R
--E 70

Type: PositiveInteger

--S 71 of 137
liw:=471
--R
--R
--R (11) 471
--R
--E 71

Type: PositiveInteger

--S 72 of 137
x:Matrix SF:=

Type: Matrix
[[0.0, 0.4, 0.8, 1.2, 1.6, 2.0, 2.4, 2.8, 3.2, 
 3.6, 4.0, 4.4, 4.8, 5.2, 5.6, 6.0, 6.4, 6.8, 
 7.2, 7.6, 8.0, 8.4, 8.8, 9.2, 9.6, 10.0, 0.0, 
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 
]]

---R
---R
---R (12)
---R [0., 0.40000000000000002, 0.80000000000000004, 1.2, 1.60000000000000001, 
 2., 2.3999999999999999, 2.7999999999999998, 3.20000000000000002, 
 3.5999999999999996, 4., 4.40000000000000004, 4.7999999999999998, 
 5.1999999999999993, 5.5999999999999996, 6., 6.40000000000000004, 
 6.7999999999999998, 7.1999999999999993, 7.5999999999999996, 8., 
 8.3999999999999998, 8.80000000000000007, 9.1999999999999993, 
 9.5999999999999996, 10., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 
 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 
 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.] 
---R ]

---R Type: Matrix(DoubleFloat)
---E 72

---S 73 of 137
np:=26
---R
---R
---R (13) 26
---R Type: PositiveInteger
---E 73

---S 74 of 137
vef:Vector Expression Float:=
  [Y[2], Y[3], -Y[1]*Y[3]-0.2*(1-Y[2]*Y[2]) ]
---R
---R
---R (14) [Y, Y, - 1.0 Y Y + 0.2 Y - 0.2]
---R 2 3 1 3 2
---R Type: Vector(Expression(Float))
---E 74

---S 75 of 137
fcn:Asp7(FCN):=retract vef
---R
---R
---R (15) FCN
---R Type: Asp7(FCN)
---E 75
result:=d02gaf(u,v,n,a,b,tol,mnp,liw,x,np,-1,fcn)

)clear all

showArrayValues true
--R
--R (1) true
--R Type: Boolean

showScalarValues true
--R
--R (2) true
--R Type: Boolean

a:SF:=0.0
--R
--R (3) 0.
--R Type: DoubleFloat

b:SF:=1.0
--R
--R (4) 1.
--R Type: DoubleFloat

n:=2
--R
--R (5) 2
--R Type: PositiveInteger

tol:SF:=1.0e-3
--R
--R
--R (6) 1.0E-3
--R
--E 82

--S 83 of 137
mnp:= 70
--R
--R
--R (7) 70
--R
--E 83

--S 84 of 137
lw:=1702
--R
--R
--R (8) 1702
--R
--E 84

--S 85 of 137
liw:=352
--R
--R
--R (9) 352
--R
--E 85

--S 86 of 137
c:Matrix SF:=[[1,0],[0,0]]
--R
--R
--R +1. 0.+
--R (10) |   |
--R +0. 0.+
--R
--E 86

--S 87 of 137
d:Matrix SF:=[[0,0],[1,0]]
--R
--R
--R +0. 0.+
--R (11) |   |
--R +1. 0.+
--R
--E 87

--S 88 of 137
\[
\text{gam: Matrix SF:= } \begin{bmatrix} 0 & 1 \end{bmatrix} \\
\text{Type: Matrix(DoubleFloat)}
\]

\[
x: \text{Matrix SF:= } \\
\begin{bmatrix}
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\
\end{bmatrix}
\text{Type: Matrix(DoubleFloat)}
\]

\[
np:=0 \\
\text{Type: NonNegativeInteger}
\]

\[
\text{ifail:=} 11 \\
\text{Type: PositiveInteger}
\]

\[
\text{mef: Matrix Expression Float:= } \begin{bmatrix} 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & -10 \end{bmatrix} \\
\text{Type: Matrix(DoubleFloat)}
\]
fcnf:Asp77(FCNF):=retract mef

vef:Vector Expression Float := \[0,0\]

fcng:Asp78(FCNG):=retract vef

result:=d02gbf(a,b,n,tol,mnp,lw,liw,c,d,gam,x,np,ifail,fcnf,fcng)

)clear all

showArrayValues true

showScalarValues true
xpoint: Matrix SF := [[0.0, 0.1, 4^(1/3), 30.0, 30.0]]

(3) [0. 0. 0.10000000000000001 1.5874010519681994 30. 30.]
Type: Matrix(DoubleFloat)

m := 5
(4) 5
Type: PositiveInteger

k := 11
(5) 11
Type: PositiveInteger

tol: SF := 0.0001
(6) 9.9999999999999991E-5
Type: DoubleFloat

maxfun := 0
(7) 0
Type: NonNegativeInteger

match := 0
(8) 0
Type: NonNegativeInteger
elam:=14
--R
--R
--R  (9) 14
--R  Type: PositiveInteger
--E 105

--S 106 of 137
delam:=1
--R
--R
--R  (10) 1
--R  Type: PositiveInteger
--E 106

--S 107 of 137
hmax:Matrix SF:=
    [[0.0,0.0],[0.0,0.0],[0.0,0.0],[0.0,0.0],[0.0,0.0]]
--R
--R  +0. 0.+
--R  |    |
--R  |    |
--R  (11) |0. 0.|
--R  |    |
--R  |    |
--R  |0. 0.|
--R  |    |
--R  |    |
--R  +0. 0.+
--R  Type: Matrix(DoubleFloat)
--E 107

--S 108 of 137
maxit:= 0
--R
--R
--R  (12) 0
--R  Type: NonNegativeInteger
--E 108

--S 109 of 137
vef:Vector Expression Float:= [1.0,ELAM-X-2.0/(X*X),1.0]
--R
--R
--R  3 2
--R  - 1.0 X + ELAM X - 2.0
--R  (13) [1.0,------------------------,1.0]
--R  2
--R  X
--R  Type: Vector(Expression(Float))
coeffn: Asp10(COEFFN) := retract vef

\begin{equation}
\text{(14) COEFFN}
\end{equation}

\text{Type: Asp10(COEFFN)}

mef: Matrix Expression Float := \[
\begin{bmatrix}
XL & 2.0 \\
1.0 & -\sqrt{XR-ELAM}
\end{bmatrix}
\]

\text{Type: Matrix(Expression(Float))}

\begin{equation}
\text{(15) | + | + | + | + | + | + | + | +}
\end{equation}

\begin{equation}
+1.0 - 1.0\sqrt{XR-1.0 \text{ELAM}} +
\end{equation}

\text{BDYVAL}

\text{Type: Asp80(BDYVAL)}

\text{result :=
\text{d02kef(xpoint,m,k,tol,maxfun,match,elam,delam,hmax,maxit,-1,coeffn,bdyval)}}

\text{the following are the default Asps used.}

\text{outputAsFortran()}\$\text{Asp12(MONIT)}$

\text{Type: Void}$
SUBROUTINE REPORT(X,V,JINT)
INTEGER JINT
DOUBLE PRECISION X,V(3)
RETURN
END

CLEAR ALL

SHOWARRAYVALUES TRUE
(1) TRUE

SHOWSCALARVALUES TRUE
(2) TRUE

N:=3
(3) 3

MNP:=40
(4) 40

NUMBER:=2
nummix:=0

tol:=1.0e-4

init:=0

iy:=3

ijac:=1

lwork:=1925
liwork := 283

np := 17

x := Matrix(17, 40, [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 10., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]

y := Matrix(3, 40, [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]

Type: Matrix(DoubleFloat)
deleps := 0.1

(16) 0.1

ifail := 11

(17) 11


fcn : Asp41(FCN, JACOBF, JACEPS) := retract vef

(19) [FCN, JACOBF, JACEPS]

vei : Vector Expression Integer := [YA[1], YA[2], YB[2] - 1]

(20) [YA, YA, YB - 1]
d02AgentsPackage contains a set of computational agents for use with Ordinary Differential Equation solvers.

See Also:
- )show d02AgentsPackage
d02AgentsPackage (D02AGNT)

Exports:
accuracyIF combineFeatureCompatibility eval expenseOfEvaluationIF
jacobian sparsityIF stiffnessAndStabilityFactor stiffnessAndStabilityOfODE

— package D02AGNT d02AgentsPackage —

)abbrev package D02AGNT d02AgentsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: January 1997
++ Description:

d02AgentsPackage(): E == I where
LEDF ==> List Expression DoubleFloat
LEEDF ==> List Equation Expression DoubleFloat
EEDF ==> Equation Expression DoubleFloat
VEDF ==> Vector Expression DoubleFloat
MEDF ==> Matrix Expression DoubleFloat
MDF ==> Matrix DoubleFloat
EDF ==> Expression DoubleFloat
DF ==> DoubleFloat
F ==> Float
INT ==> Integer
CDF ==> Complex DoubleFloat
LDF ==> List DoubleFloat
LF ==> List Float
S ==> Symbol
LS ==> List Symbol
MFI ==> Matrix Fraction Integer
LFI ==> List Fraction Integer
FI ==> Fraction Integer
ON ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)
RVE ==> Record(val:EDF,exponent:INT)
RSS ==> Record(stiffnessFactor:F, stabilityFactor:F)
ROA ==> Record(key:ODEA, entry:ATT)

E ==> with
combineFeatureCompatibility: (F,F) -> F
++ combineFeatureCompatibility(C1,C2) is for interacting attributes
combineFeatureCompatibility: (F,LP) -> F
++ combineFeatureCompatibility(C1,L) is for interacting attributes

sparsityIF: MEDF -> F
++ sparsityIF(m) calculates the sparsity of a jacobian matrix
jacobian: (VDF,LS) -> MEDF
++ jacobian(v,w) is a local function to make a jacobian matrix
eval: (MEDF,LS,VEDF) -> MEDF
++ eval(mat,symbols,values) evaluates a multivariable matrix at given values
++ for each of a list of variables

stiffnessAndStabilityFactor: MEDF -> RSS
++ stiffnessAndStabilityFactor(me) calculates the stability and
++ stiffness factor of a system of first-order differential equations
++ (by evaluating the maximum difference in the real parts of the
++ negative eigenvalues of the jacobian of the system for which O(10)
++ equates to mildly stiff whereas stiffness ratios of O(10^6) are not
++ uncommon) and whether the system is likely to show any oscillations
++ (identified by the closeness to the imaginary axis of the complex
++ eigenvalues of the jacobian).

stiffnessAndStabilityOfODEIF:ODEA -> RSS
++ stiffnessAndStabilityOfODEIF(ode) calculates the intensity values
++ of stiffness of a system of first-order differential equations
++ (by evaluating the maximum difference in the real parts of the
++ negative eigenvalues of the jacobian of the system for which O(10)
++ equates to mildly stiff whereas stiffness ratios of O(10^6) are not
++ uncommon) and whether the system is likely to show any oscillations
++ (identified by the closeness to the imaginary axis of the complex
++ eigenvalues of the jacobian).
++
++ It returns two values in the range [0,1].

systemSizeIF:ODEA -> F
++ systemSizeIF(ode) returns the intensity value of the size of
++ the system of ODEs. 20 equations corresponds to the neutral
++ value. It returns a value in the range [0,1].

expenseOfEvaluationIF:ODEA -> F
++ expenseOfEvaluationIF(o) returns the intensity value of the
++ cost of evaluating the input ODE. This is in terms of the number
++ of ‘‘operation units’’. It returns a value in the range
++ [0,1].
++ 400 ‘‘operation units’’ -> 0.75
++ 200 ‘‘operation units’’ -> 0.5
++ 83 ‘‘operation units’’ -> 0.25
++ exponentiation = 4 units , function calls = 10 units.

accuracyIF:ODEA -> F
++ accuracyIF(o) returns the intensity value of the accuracy
++ requirements of the input ODE. A request of accuracy of $10^{-6}$
++ corresponds to the neutral intensity. It returns a value
++ in the range $[0,1]$.

intermediateResultsIF:ODEA -> F
++ intermediateResultsIF(o) returns a value corresponding to the
++ required number of intermediate results required and, therefore,
++ an indication of how much this would affect the step-length of the
++ calculation. It returns a value in the range $[0,1]$.

I ==> add

import ExpertSystemToolsPackage

accuracyFactor:ODEA -> F
expenseOfEvaluation:ODEA -> F
eval1:(LED,LEEDF) -> LEDF
stiffnessAndStabilityOfODE:ODEA -> RSS
intermediateResultsFactor:ODEA -> F
leastStabilityAngle:(LDF,LDF) -> F

intermediateResultsFactor(ode:ODEA):F ==
  resultsRequirement := #(ode.intvals)
  (1.0-exp(-(resultsRequirement::F)/50.0)$F)

intermediateResultsIF(o:ODEA):F ==
  ode := copy o
  (t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
  s := coerce(t)@ATT
  negative?(s.intermediateResults)$F =>
  s.intermediateResults := intermediateResultsFactor(ode)
  r:ROA := [ode,s]
  insert!(r)$ODEIntensityFunctionsTable
  s.intermediateResults

accuracyFactor(ode:ODEA):F ==
  accuracyRequirements := convert(ode.abserr)@F
  if zero?(accuracyRequirements) then
    accuracyRequirements := convert(ode.relerr)@F
  val := inv(accuracyRequirements)$F
  n := log10(val)$F
  (1.0-exp(-(n/(2.0))**2/(15.0))$F)

accuracyIF(o:ODEA):F ==
  ode := copy o
  (t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
s := coerce(t)@ATT
negative?(s.accuracy)$F =>
s.accuracy := accuracyFactor(ode)
r:ROA := [ode,s]
insert!(r)$ODEIntensityFunctionsTable
s.accuracy
s.accuracy
a:ATT := [-1.0,-1.0,-1.0,e:=accuracyFactor(ode),-1.0]
r:ROA := [ode,a]
insert!(r)$ODEIntensityFunctionsTable
e
systemSizeIF(ode:ODEA):F ==
n := #(ode.fn)
(1.0-exp((-n::F/75.0))$F)

expenseOfEvaluation(o:ODEA):F ==
-- expense of evaluation of an ODE -- <0.3 inexpensive - 0.5 neutral - >0.7 very expensive
-- 400 ‘operation units’ -> 0.75
-- 200 ‘operation units’ -> 0.5
-- 83 ‘operation units’ -> 0.25
-- ** = 4 units , function calls = 10 units.
ode := copy o.fn
expenseOfEvaluation(ode)

expenseOfEvaluationIF(o:ODEA):F ==
ode := copy o
(t := showIntensityFunctions(ode)$ODEIntensityFunctionsTable) case ATT =>
s := coerce(t)@ATT
negative?(s.expense)$F =>
s.expense := expenseOfEvaluation(ode)
r:ROA := [ode,s]
insert!(r)$ODEIntensityFunctionsTable
s.expense
s.expense
a:ATT := [-1.0,-1.0,e:=expenseOfEvaluation(ode),-1.0,-1.0]
r:ROA := [ode,a]
insert!(r)$ODEIntensityFunctionsTable
e
leastStabilityAngle(realPartsList:LDF,imagPartsList:LDF):F ==
complexList := [complex(u,v)$CDF for u in realPartsList for v in imagPartsList]
argumentList := [abs((abs(argument(u)$CDF)$DF)-(pi()$DF)/2)$DF for u in complexList]
sortedArgumentList := sort(argumentList)$LDF
list := [u for u in sortedArgumentList | not zero?(u) ]
empty?(list)$LDF => 0$F
convert(first(list)$LDF)$F

stiffnessAndStabilityFactor(me:MEDF):RSS ==
-- search first for real eigenvalues of the jacobian (symbolically)
-- if the system isn't too big
r:INT := ncols(me)$MEDF
b:Boolean := ((# me) < 150)
if b then
  mc:MFI := map(edf2fi,me)$ExpertSystemToolsPackage2(EDF,FI)
e:LFI := realEigenvalues(mc,1/100)$NumericRealEigenPackage(FI)
b := ((# e) >= r-1)@Boolean
b =>
  -- if all the eigenvalues are real, find negative ones
  e := sort(neglist(e)$ExpertSystemToolsPackage1(FI))
  -- if there are two or more, calculate stiffness ratio
  ((n:=#e)>1)@Boolean => [coerce(e.1/e.n)$F,0$F]
  -- otherwise stiffness not present
  [0$F,0$F]
md:MDF := map(edf2df,me)$ExpertSystemToolsPackage2(EDF,DF)
-- otherwise calculate numerically the complex eigenvalues
-- using NAG routine f02aff.
res:Result := f02aff(r,r,md,-1)$NagEigenPackage
realParts:Union(Any,"failed") := search(rr::Symbol,res)$Result
realParts case "failed" => [0$F,0$F]
realPartsMatrix:MDF := retract(realParts)$AnyFunctions1(MDF) -- array === matrix
imagParts:Union(Any,"failed") := search(ri::Symbol,res)$Result
imagParts case "failed" => [0$F,0$F]
imagPartsMatrix:MDF := retract(imagParts)$AnyFunctions1(MDF) -- array === matrix
imagPartsList:LDF := members(imagPartsMatrix)$MDF
realPartsList:LDF := members(realPartsMatrix)$MDF
stabilityAngle := leastStabilityAngle(realPartsList,imagPartsList)
negRealPartsList := sort(neglist(realPartsList)$ExpertSystemToolsPackage1(DF))
empty?(negRealPartsList)$LDF => [0$F,0$F]
((n:=#negRealPartsList)>1)@Boolean =>
  out := convert(negRealPartsList.1/negRealPartsList.n)$F
  [out,0$F]
  -- calculate stiffness ratio
  [-convert(negRealPartsList.1)$F,0$F]

eval1(l:LEDF,e:LEEDF):LEDF ==
  [eval(u,e)$EDF for u in l]
eval(mat:MEDF,symbols:LS,values:VEDF):MEDF ==
  l := listOfLists(mat)
  ledf := entries(values)$VEDF
  e := [equation(u::EDF,v)$EEDF for u in symbols for v in ledf]
  l := [eval1(w,e) for w in l]
  matrix l

combineFeatureCompatibility(C1:F,C2:F):F ==
\[
\begin{align*}
\text{-- } & \quad C1 \quad C2 \\
\text{-- } & \quad s(C1,C2) = \frac{C1*C2}{(C1+C2)+(1-C1)(1-C2)} \\
\text{-- } & \quad (1-C1)(1-C2)
\end{align*}
\]

\[
C1*2/((C1+C2)+(1-F-C1)*(1-F-C2))
\]

```lisp
combineFeatureCompatibility(C1:F,L:LF):F ==
  empty?(L)$LF => C1
  C2 := combineFeatureCompatibility(C1,first(L)$LF)
  combineFeatureCompatibility(C2,rest(L)$LF)
```

```lisp
jacobian(v:VEDF,w:LS):Matrix EDF ==
  jacobian(v,w)$MultiVariableCalculusFunctions(S,EDF,VEDF,LS)
```

```lisp
sparsityIF(m:Matrix EDF):F ==
  l:LEDF := parts m
  z:LEDF := [u for u in l | zero?(u)$EDF]
  ((#z)::F/(#l)::F)
```

```lisp
sum(a:EDF,b:EDF):EDF == a+b
```

```lisp
stiffnessAndStabilityOfODE(ode:ODEA):RSS ==
  odefns := copy ode.fn
  ls:LS := [subscript(Y,[coerce(n)])$Symbol for n in 1..# odefns]
  yvals := copy ode.yinit
  for i in 1..#yvals repeat
    zero?(yvals.i) => yvals.i := 0.1::DF
  yepr := [coerce(v)@EDF for v in yvals]
  yv:VEDF := vector(yexpr)
  j1:MEDF := jacobian(odefn,ls)
  ej1:MEDF := eval(j1,ls,yv)
  ej1 := eval(ej1,variables(reduce(sum,members(ej1)$MEDF)),vector([ode.xinit::EDF]))
  ssf := stiffnessAndStabilityFactor(ej1)
  stability := 1.0-sqrt((ssf.stabilityFactor)*(2.0)/(pi()$F))
  stiffness := (1.0)-exp(-(ssf.stiffnessFactor)/(500.0))
  [stiffness,stability]
```

```lisp
stiffnessAndStabilityOfODEIF(ode:ODEA):RSS ==
  odefn := copy ode
  (t := showIntensityFunctions(odefn)$ODEIntensityFunctionsTable) case ATT =>
  s:ATT := coerce(t)@ATT
  negative?(s.stiffness)$F =>
    ssf:RSS := stiffnessAndStabilityOfODE(odefn)
    s := [ssf.stiffnessFactor,ssf.stabilityFactor,s.expense,
      s.accuracy,s.intermediateResults]
  r:ROA := [odefn,s]
  insert!(r)$ODEIntensityFunctionsTable
  ssf
  [s.stiffness,s.stability]
```
ssf:RSS := stiffnessAndStabilityOfODE(odefn)
s:ATT := [ssf.stiffnessFactor,ssf.stabilityFactor,-1.0,-1.0,-1.0]
r:ROA := [odefn,s]
insert!(r)$ODEIntensityFunctionsTable
ssf

---

D02AGNT.dotabb ---

"D02AGNT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=D02AGNT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"D02AGNT" -> "FS"

package D03AGNT d03AgentsPackage

--- d03AgentsPackage.input ---

)set break resume
)sys rm -f d03AgentsPackage.output
)spool d03AgentsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 91
)show d03AgentsPackage
--R
--R d03AgentsPackage is a package constructor
--R Abbreviation for d03AgentsPackage is D03AGNT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for D03AGNT
--R
--R------------------------------------------------ Operations -------------------
--R central? : (DoubleFloat,DoubleFloat,List(Expression(DoubleFloat))) -> Boolean
--R elliptic? : Record(pde: List(Expression(DoubleFloat)),constraints: List(Record(start: DoubleFloat,finish: DoubleFloat)),f: List(List(Expression(DoubleFloat))),st: String,tol: DoubleFloat) -> Boolean
--R subscriptedVariables : Expression(DoubleFloat) -> Expression(DoubleFloat)
--R varList : (Symbol,NonNegativeInteger) -> List(Symbol)
--R
--E 1

)clear all
showArrayValues true
(1) true
Type: Boolean

showScalarValues true
(2) true
Type: Boolean

ngx := 9
(3) 9
Type: PositiveInteger

ngy := 9
(4) 9
Type: PositiveInteger

lda := 134
(5) 134
Type: PositiveInteger

alpha := 1.7
(6) 1.7
Type: Float
hx := 1/(ngx+1)
--R
--R
--R 1
--R (7) --
--R 10
--R Type: Fraction(Integer)

hy := 1/(ngy+1)
--R
--R
--R 1
--R (8) --
--R 10
--R Type: Fraction(Integer)

a := new(lda,7,0.0)$Matrix DoubleFloat;
--R
--R
--R Type: Matrix(DoubleFloat)

rhs := new(1,lda,0.0)$Matrix DoubleFloat;
--R
--R
--R Type: Matrix(DoubleFloat)

ub := new(1,ngx*ngy,0.0)$Matrix DoubleFloat;
--R
--R
--R Type: Matrix(DoubleFloat)

--S 13 of 91
for j in 1..ngy repeat
  for i in 1..ngx repeat
    k := (j-1)*ngx + i
    a(k,1) := 1 - 0.5*alpha
    a(k,2) := 0.5*alpha
    a(k,3) := 1 - 0.5*alpha
    a(k,4) := -4 + alpha
    a(k,5) := 1 - 0.5*alpha
    a(k,6) := 0.5*alpha
a(k,7) := 1 - 0.5*alpha
rhs(1,k) := -4.0*hx*hy

--R
--R
--E 13

--S 14 of 91
for i in 2..(ngx-1) repeat
  ix := i
  a(ix,1) := 0
  a(ix,2) := 0
  ix := i + (ngy -1)*ngx
  a(ix,6) := 0
  a(ix,7) := 0

--R
--R Compiling function G4674 with type Integer -> Boolean
--R Compiling function G4844 with type NonNegativeInteger -> Boolean
--R
--R Type: Void
--E 14

--S 15 of 91
for j in 2..(ngy-1) repeat
  iy := (j-1)*ngx+1
  a(iy,3) := 0
  a(iy,6) := 0
  iy := j *ngx
  rhs(1,iy) := rhs(1,iy) - a(iy,5) - a(iy,2)
  a(iy,2) := 0
  a(iy,5) := 0

--R
--R
--E 15

--S 16 of 91
k := 1
--R
--R
--R (15) 1
--R Type: PositiveInteger
--E 16

--S 17 of 91
a(1,1) := 0
--R
--R
--R (16) 0.
--R
--R Type: DoubleFloat
--E 17

--S 18 of 91
a(1,2) := 0
--R
--R
--R (17) 0.  
--R
--E 18  
Type: DoubleFloat

--S 19 of 91
a(1,3) := 0
--R
--R
--R (18) 0.  
--R
--E 19  
Type: DoubleFloat

--S 20 of 91
a(1,6) := 0
--R
--R
--R (19) 0.  
--R
--E 20  
Type: DoubleFloat

--S 21 of 91
k := 1 + (ngy-1)*ngx
--R
--R
--R (20) 73  
--R
--E 21  
Type: PositiveInteger

--S 22 of 91
a(k,3) := 0
--R
--R
--R (21) 0.  
--R
--E 22  
Type: DoubleFloat

--S 23 of 91
a(k,6) := 0
--R
--R
--R (22) 0.  
--R
--E 23  
Type: DoubleFloat

--S 24 of 91
a(k,7) := 0
--R
--R
--R (23) 0.
--R Type: DoubleFloat
--E 24

--S 25 of 91
k := ngx
--R
--R (24) 9
--R Type: PositiveInteger
--E 25

--S 26 of 91
rhs(1,k) := rhs(1,k) - a(k,2)*0.5 - a(k,5)
--R
--R (25) - 0.61499999999999999
--R Type: DoubleFloat
--E 26

--S 27 of 91
a(k,1) := 0
--R
--R (26) 0.
--R Type: DoubleFloat
--E 27

--S 28 of 91
a(k,2) := 0
--R
--R (27) 0.
--R Type: DoubleFloat
--E 28

--S 29 of 91
a(k,5) := 0
--R
--R (28) 0.
--R Type: DoubleFloat
--E 29

--S 30 of 91
k := ngx * ngy
--R
--R (29) 81
\[ \text{rhs}(1,k) := \text{rhs}(1,k) - a(k,2) - a(k,5) \]

\[ (30) \quad -1.04 \]

\[ a(k,2) := 0 \]

\[ (31) \quad 0. \]

\[ a(k,5) := 0 \]

\[ (32) \quad 0. \]

\[ a(k,6) := 0 \]

\[ (33) \quad 0. \]

\[ a(k,7) := 0 \]

\[ (34) \quad 0. \]

\[ \text{ifail} := 0 \]

\[ (35) \quad 0 \]
--S 37 of 91
maxit := 15
--R
--R
--R (36) 15
--R Type: PositiveInteger
--E 37

--S 38 of 91
acc := 1.0e-4
--R
--R
--R (37) 0.0001
--R Type: Float
--E 38

--S 39 of 91
iout := 0
--R
--R
--R (38) 0
--R Type: NonNegativeInteger
--E 39

--S 40 of 91
result:=d03edf(ngx,ngy,lda,maxit,acc,iout,a,rhs,ub,ifail)
--E 40
)
clear all

--S 41 of 91
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 41

--S 42 of 91
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 42

--S 43 of 91
xmin := 0.0
--R
\begin{verbatim}
--R
--R  (3)  0.0  
--R
--E 43

--S 44 of 91 
xmax := 1.0 
--R
--R
--R  (4)  1.0  
--R
--E 44

--S 45 of 91 
ymin := 0.0 
--R
--R
--R  (5)  0.0  
--R
--E 45

--S 46 of 91 
ymax := 1.0 
--R
--R
--R  (6)  1.0  
--R
--E 46

--S 47 of 91 
psi := -sin(X)*sin(Y) + 50*cos(X)*cos(Y) + 50*sin(X)*cos(Y) 
--R
--R
--R  (7)  - sin(X)sin(Y) + 50.0 cos(Y)sin(X) + 50.0 cos(X)cos(Y)  
--R
--E 47

--S 48 of 91 
d03eea :Vector Expression Float := [1,1,1,50,50,0,psi] 
--R
--R
--R  (8) 
--R  [1.0, 0.0, 1.0, 50.0, 50.0, 0.0, 
--R  - 1.0 sin(X)sin(Y) + 50.0 cos(Y)sin(X) + 50.0 cos(X)cos(Y)] 
--R
--E 48

--S 49 of 91 
d03eeb :Matrix Expression Float := _ 
    matrix[[0,1,-sin(X)],[1,0,sin(X)*sin(Y)],[1,0,sin(X)*sin(Y)],[0,1,-sin(Y)]]
\end{verbatim}


```plaintext
PACKAGE D03AGENT D03AGENTS PACKAGE

+0.0  1.0 - 1.0 sin(X) +
|  |
|1.0  0.0 sin(X)sin(Y) |
(9) |
|1.0  0.0 sin(X)sin(Y) |
|  |
+0.0  1.0 - 1.0 sin(Y) +

Type: Matrix(Expression(Float))
```

```plaintext
ngx := 9

(10) 9

Type: PositiveInteger
```

```plaintext
ngy := 9

(11) 9

Type: PositiveInteger
```

```plaintext
lda := 133

(12) 133

Type: PositiveInteger
```

```plaintext
scheme := "c"

(13) "c"

Type: String
```

```plaintext
ifail := -1

(14) - 1

Type: Integer
```
--E 54
--S 55 of 91
result:=d03eef(xmin,xmax,ymin,ymax,ngx,ngy,lda,scheme,ifail,_
    d03eea::ASP73('PDEF),d03eeb::ASP74('BNDY))
--E 55

)clear all
--S 56 of 91
showArrayValues true
--R
--R
--R  (1) true
--R
--E 56

--S 57 of 91
showScalarValues true
--R
--R
--R  (2) true
--R
--E 57

--S 58 of 91
xs := 0.0
--R
--R
--R  (3) 0.0
--R
--E 58

--S 59 of 91
xf := 1.0
--R
--R
--R  (4) 1.0
--R
--E 59

--S 60 of 91
l := 16
--R
--R
--R  (5) 16
--R
--E 60

--S 61 of 91
lbdcnd := 1
--R
--R
--R (6) 1
--R Type: PositiveInteger  
--E 61

--S 62 of 91
m := 32
--R
--R
--R (7) 32
--R Type: PositiveInteger  
--E 62

--S 63 of 91
n := 20
--R
--R
--R (8) 20
--R Type: PositiveInteger  
--E 63

--S 64 of 91
maxlm := 32
--R
--R
--R (9) 32
--R Type: PositiveInteger  
--E 64

--S 65 of 91
mdimf := m+1
--R
--R
--R (10) 33
--R Type: PositiveInteger  
--E 65

--S 66 of 91
ldimf := l+1
--R
--R
--R (11) 17
--R Type: PositiveInteger  
--E 66

--S 67 of 91
lwrk := 2*(n+1)*maxlm+3*l+3*m+4*n+6000
--R
bdxs := new(mdimf,n+1,0.0)$Matrix DoubleFloat;

bdxf := new(mdimf,n+1,0.0)$Matrix DoubleFloat;

ys := 0.0

yf := 2*numeric(%pi)

mbdcnd := 0

bdys := new(ldimf,n+1,0.0)$Matrix DoubleFloat;
bdyf := new(ldimf,n+1,0.0)$Matrix DoubleFloat;  Type: Matrix(DoubleFloat)

zs := 0.0  Type: Float

zf := numeric(%pi)/2  Type: Float

nbdcnd := 2  Type: PositiveInteger

bdzs := new(ldimf,m+1,0.0)$Matrix DoubleFloat;  Type: Matrix(DoubleFloat)

bdzf := new(ldimf,m+1,0.0)$Matrix DoubleFloat;  Type: Matrix(DoubleFloat)

lambda := -2  Type: Float
ifail := 0

define grid points

dx := (xf-xs)/l

dy := (yf-ys)/m

dz := (zf-zs)/n

x := [[xs + (i-1)*dx for i in 1..l+1]] :: Matrix DoubleFloat;

y := [[ys + (i-1)*dy for i in 1..m+1]] :: Matrix DoubleFloat;

dz := (zf-zs)/n

Type: Integer

Type: NonNegativeInteger

Type: Float

Type: Matrix(DoubleFloat)

Type: Matrix(DoubleFloat)

Type: Float
--S 87 of 91
z := [[zs + (i-1)*dz for i in 1..n+1]] :: Matrix DoubleFloat;
--R
--R
--R Type: Matrix(DoubleFloat)
--E 87

--S 88 of 91
f:=zeroMatrix(ldimf,mdimf,n+1)$M3D DFLOAT;
--R
--R
--R Type: ThreeDimensionalMatrix(DoubleFloat)
--E 88

--speed up these loops by compiling
--S 89 of 91
foo()==
for k in 1..n+1 repeat
  for j in 1..m+1 repeat
    setelt!(f,1,j,k,sin(y(1,j))*cos(z(1,k)))
  for j in 1..m+1 repeat
    for i in 1..l+1 repeat
      setelt!(f,i,j,1,x(1,i)^4*sin(y(1,j)))
    for k in 2..n+1 repeat
      for j in 1..m+1 repeat
        for i in 2..l repeat
          setelt!(f,i,j,k,4*x(1,i)^2*(3-x(1,i)^2)*sin(y(1,j))*cos(z(1,k)))
      for j in 1..m+1 repeat
        for i in 1..l+1 repeat
          bdzf(i,j):=-x(1,i)^4*sin(y(1,j))
  --R
  --R Type: Void
--E 89

--S 90 of 91
foo()
--R
--R Compiling function foo with type () -> Void
--R
--E 90

--S 91 of 91
-- result:=d03faf(xs,xf,l,lbdcnd,bdxs,bdxf,ys,yf,m,mbdcnd,bdys,bdyf,_
  zs,zf,n,nbdcnd,bdzs,bdxf,lambda,ldimf,mdimf,lwrk,f,ifail)
--E 91

)spool

)lisp (bye)
d03AgentsPackage (D03AGNT)

Exports:
central? elliptic? subscriptedVariables varList

--- package D03AGNT d03AgentsPackage ---

)abbrev package D03AGNT d03AgentsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: December 1997
++ Description:
++ \textit{axiom\{d03AgentsPackage\}} contains a set of computational agents
++ for use with Partial Differential Equation solvers.

d03AgentsPackage(): E == I where
    LEDF ==> List Expression DoubleFloat
PACKAGE D03AGNT D03AGENTSPACKAGE

EDF ==> Expression DoubleFloat
MDF ==> Matrix DoubleFloat
DF ==> DoubleFloat
F ==> Float
INT ==> Integer
NNI ==> NonNegativeInteger
EEDF ==> Equation Expression DoubleFloat
LEEDF ==> List Equation Expression DoubleFloat
LDF ==> List DoubleFloat
LOCDF ==> List OrderedCompletion DoubleFloat
OCDF ==> OrderedCompletion DoubleFloat
LS ==> List Symbol
PDEC ==> Record(start:DF, finish:DF, grid:NNI, boundaryType:INT,
dStart:MDF, dFinish:MDF)
PDEB ==> Record(pde:LEDF, constraints:List PDEC,
f:List LEDF, st:String, tol:DF)
NOA ==> Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)

E ==> with
  varList:(Symbol,NonNegativeInteger) -> LS
  ++ varList(s,n) \ undocumented{}
  subscriptedVariables:EDF -> EDF
  ++ subscriptedVariables(e) \ undocumented{}
  central?(x:DF,y:DF,p:LEDF) -> Boolean
  ++ central?(f,g,l) \ undocumented{}
  elliptic?:PDEB -> Boolean
  ++ elliptic?(r) \ undocumented{}

I ==> add

import ExpertSystemToolsPackage

sum(a:EDF,b:EDF):EDF == a+b

varList(s:Symbol,n:NonNegativeInteger):LS ==
  [subscript(s,[t::OutputForm]) for t in expand([1..n]\$Segment(Integer))]

subscriptedVariables(e:EDF):EDF ==
  oldVars:List Symbol := variables(e)
  o := [a :: EDF for a in oldVars]
  n := [b :: EDF for b in newVars]
  subst(e,[a=b for a in o for b in n])

central?(x:DF,y:DF,p:LEDF):Boolean ==
  ls := variables(reduce(sum,p))
  le := [equation(u::EDF,v)$EEDF for u in ls for v in [x::EDF,y::EDF]]
  l := [eval(u,le)$EDF for u in p]
  max(1.4,1.5) < 20 * max(1.1,\max(1.2,1.3))
elliptic?(args:PDEB):Boolean ==
  (args.st)="elliptic" => true
  p := args.pde
  xcon:PDEC := first(args.constraints)
  ycon:PDEC := second(args.constraints)
  xs := xcon.start
  ys := ycon.start
  xf := xcon.finish
  yf := ycon.finish
  xstart:DF := ((xf-xs)/2)$DF
  ystart:DF := ((yf-ys)/2)$DF
  optStart:LDF := [xstart,ystart]
  lower:LOCDF := [xs::OCDF,ys::OCDF]
  upper:LOCDF := [xf::OCDF,yf::OCDF]
  v := variables(e := 4*first(p)*third(p)-(second(p))**2)
  eq := subscriptedVariables(e)
  noa:NOA :=
    -- one? (# v) =>
    (# v) = 1 =>
      ((first v) = X$Symbol) =>
        [eq,[xstart],[xs::OCDF],empty()$LEDF,[xf::OCDF]]
        [eq,[ystart],[ys::OCDF],empty()$LEDF,[yf::OCDF]]
        [eq,optStart,lower,empty()$LEDF,upper]
    ell := optimize(noa::NumericalOptimizationProblem)$AnnaNumericalOptimizationPackage
    o:Union(Any,"failed") := search(objf::Symbol,ell)$Result
    o case "failed" => false
    ob := o :: Any
    obj:DF := retract(ob)$AnyFunctions1(DF)
    positive?(obj)
Chapter 6

Chapter E

package EP EigenPackage

— EigenPackage.input —

)set break resume
)sys rm -f EigenPackage.output
)spool EigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EigenPackage

--R
--R EigenPackage(R: GcdDomain) is a package constructor
--R Abbreviation for EigenPackage is EP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EP
--R
--R-------------------------------------------------- Operations ----------------------------
--R characteristicPolynomial : (Matrix(Fraction(Polynomial(R))),Symbol) -> Polynomial(R)
--R characteristicPolynomial : Matrix(Fraction(Polynomial(R))) -> Polynomial(R)
--R eigenvalues : Matrix(Fraction(Polynomial(R))) -> List(Union(Fraction(Polynomial(R)),SuchThat(Symbol,Polynomial(R))))
--R eigenvector : (Union(Fraction(Polynomial(R)),SuchThat(Symbol,Polynomial(R))),Matrix(Fraction(Polynomial(R)))) -> List(Matrix(Fraction(Polynomial(R))))
--R eigenvectors : Matrix(Fraction(Polynomial(R))) -> List(Record(eigval: Union(Fraction(Polynomial(R)),SuchThat(Symbol,Polynomial(R))),eigmult: NonNegativeInteger,eigvec: List(Matrix(Fraction(Polynomial(R))))))
--R generalizedEigenvector : (Union(Fraction(Polynomial(R)),SuchThat(Symbol,Polynomial(R))),Matrix(Fraction(Polynomial(R))),NonNegativeInteger,NonNegativeInteger) -> List(Matrix(Fraction(Polynomial(R))))
--R generalizedEigenvector : (Record(eigval: Union(Fraction(Polynomial(R)),SuchThat(Symbol,Polynomial(R))),eigmult: NonNegativeInteger,....),Matrix(Fraction(Polynomial(R)))) -> List(Matrix(Fraction(Polynomial(R))))
--R generalizedEigenvectors : Matrix(Fraction(Polynomial(R))) -> List(Record(eigval: Union(Fraction(Polynomial(R)),SuchThat(Symbol,Polynomial(R))),geneigvec: List(Matrix(Fraction(Polynomial(R))))))

--E 1
This is a package for the exact computation of eigenvalues and eigenvectors. This package can be made to work for matrices with coefficients which are rational functions over a ring where we can factor polynomials. Rational eigenvalues are always explicitly computed while the non-rational ones are expressed in terms of their minimal polynomial.

See Also:
o )show EigenPackage

Exports:
characteristicPolynomial eigenvalues eigenvector eigenvectors generalizedEigenvector generalizedEigenvectors

— package EP EigenPackage —

)abbrev package EP EigenPackage
++ Author: P. Gianni
++ Date Created: summer 1986
++ Date Last Updated: October 1992
++ Description:
++ This is a package for the exact computation of eigenvalues and eigenvectors.
++ This package can be made to work for matrices with coefficients which are
++ rational functions over a ring where we can factor polynomials.
++ Rational eigenvalues are always explicitly computed while the
++ non-rational ones are expressed in terms of their minimal
++ polynomial.
-- Functions for the numeric computation of eigenvalues and eigenvectors
-- are in numeigen.spad.

EigenPackage(R) : C == T
where
  R : GcdDomain
  P ==> Polynomial R
  F ==> Fraction P
  SE ==> Symbol()
  SUP ==> SparseUnivariatePolynomial(P)
  SUF ==> SparseUnivariatePolynomial(F)
  M ==> Matrix(F)
  NNI ==> NonNegativeInteger
  ST ==> SuchThat(SE,P)

Eigenvalue ==> Union(F,ST)
EigenForm ==> Record(eigval:Eigenvalue,eigmult:NNI,eigvec : List M)
GenEigen ==> Record(eigval:Eigenvalue,geneigvec:List M)

C == with
  characteristicPolynomial : (M,Symbol) -> P
  ++ characteristicPolynomial(m,var) returns the
  ++ characteristicPolynomial of the matrix m using
  ++ the symbol var as the main variable.

  characteristicPolynomial : M -> P
  ++ characteristicPolynomial(m) returns the
  ++ characteristicPolynomial of the matrix m using
  ++ a new generated symbol symbol as the main variable.

  eigenvalues : M -> List Eigenvalue
  ++ eigenvalues(m) returns the
  ++ eigenvalues of the matrix m which are expressible
  ++ as rational functions over the rational numbers.

  eigenvector : (Eigenvalue,M) -> List M
  ++ eigenvector(eigval,m) returns the
  ++ eigenvectors belonging to the eigenvalue eigval
  ++ for the matrix m.

  generalizedEigenvector : (Eigenvalue,M,NNI,NNI) -> List M
++ generalizedEigenvector(alpha,m,k,g)
++ returns the generalized eigenvectors
++ of the matrix relative to the eigenvalue alpha.
++ The integers k and g are respectively the algebraic and the
++ geometric multiplicity of the eigenvalue alpha.
++ alpha can be either rational or not.
++ In the second case alpha is the minimal polynomial of the
++ eigenvalue.

generalizedEigenvector : (EigenForm,M) -> List M
++ generalizedEigenvector(eigen,m)
++ returns the generalized eigenvectors
++ of the matrix relative to the eigenvalue eigen, as
++ returned by the function eigenvectors.

generalizedEigenvectors : M -> List GenEigen
++ generalizedEigenvectors(m)
++ returns the generalized eigenvectors
++ of the matrix m.

eigenvectors : M -> List(EigenForm)
++ eigenvectors(m) returns the eigenvalues and eigenvectors
++ for the matrix m.
++ The rational eigenvalues and the correspondent eigenvectors
++ are explicitly computed, while the non rational ones
++ are given via their minimal polynomial and the corresponding
++ eigenvectors are expressed in terms of a "generic" root of
++ such a polynomial.

T == add
PI ==> PositiveInteger

MF := GeneralizedMultivariateFactorize(SE,IndexedExponents SE,R,R,P)
UPCF2:= UnivariatePolynomialCategoryFunctions2(P,SUP,F,SUF)

---- Local Functions ----
tff : (SUF,SE) -> F
fft : (SUF,SE) -> F
charpol : (M,SE) -> F
intRatEig : (F,M,NNI) -> List M
intAlgEig : (ST,M,NNI) -> List M
genEigForm : (EigenForm,M) -> GenEigen

---- next functions needed for defining ModularField ----
reduction(u:SUF,p:SUF):SUF == u rem p
merge(p:SUF,q:SUF):Union(SUF,"failed") ==
p = q => p


```
p = 0 => q
q = 0 => p
"failed"

val:=extendedEuclidean(v,p,u)
val case "failed" => "failed"
val.coef1

---- functions for conversions ----
fft(t:SUF,x:SE):F ==
n:=degree(t)
cof:=monomial(1,x,n)$P :: F
cf * leadingCoefficient t

tff(p:SUF,x:SE) : F ==
degree p=0 => leadingCoefficient p
r:F:=0$F
while p^=0 repeat
  r:=r+fft(p,x)
p := reductum p
r

---- generalized eigenvectors associated to a given eigenvalue ----
genEigForm(eigen : EigenForm,A:M) : GenEigen ==
alpha:=eigen.eigval
k:=eigen.eigmult
g:=#(eigen.eigvec)
k = g => [alpha,eigen.eigvec]
[alpha,generalizedEigenvector(alpha,A,k,g)]

---- characteristic polynomial ----
charpol(A:M,x:SE) : F ==
dimA :PI := (nrows A):PI
dimA ^= ncols A => error " The matrix is not square"
B:M:=zero(dimA,dimA)
for i in 1..dimA repeat
  for j in 1..dimA repeat B(i,j):=A(i,j)
  B(i,i) := B(i,i) - monomial(1$P,x,1):F
determinant B

-------- EXPORTED FUNCTIONS --------

---- characteristic polynomial of a matrix A ----
characteristicPolynomial(A:M):P ==
x:SE:=new():$SE
numer charpol(A,x)

---- characteristic polynomial of a matrix A ----
characteristicPolynomial(A:M,x:SE) : P == numer charpol(A,x)
```
--- Eigenvalues of the matrix A ---
eigenvalues(A:M): List Eigenvalue ==
x:=new()$SE
pol:= charpol(A,x)
lrat:List F :=empty()
lsym:List ST :=empty()
for eq in solve(pol,x)$SystemSolvePackage(R) repeat
  alg:=numer lhs eq
  degree(alg, x)=1 => lrat:=cons(rhs eq,lrat)
  lsym:=cons([x,alg],lsym)
append([lr::Eigenvalue for lr in lrat],
    [ls::Eigenvalue for ls in lsym])

--- Eigenvectors belonging to a given eigenvalue ---
---- the eigenvalue must be exact ----
eigenvector(alpha:Eigenvalue,A:M) : List M ==
alpha case F => intRatEig(alpha::F,A,1$NNI)
intAlgEig(alpha::ST,A,1$NNI)

--- Eigenvectors belonging to a given rational eigenvalue ----
---- Internal function ----
intRatEig(alpha:F,A:M,m:NNI) : List M ==
n:=nrows A
B:M := zero(n,n)$M
for i in 1..n repeat
  for j in 1..n repeat B(i,j):=A(i,j)
  B(i,i):= B(i,i) - alpha
[v::M for v in nullSpace(B**m)]

--- Eigenvectors belonging to a given algebraic eigenvalue ----
------ Internal Function ------
intAlgEig(alpha:ST,A:M,m:NNI) : List M ==
n:=nrows A
MM := ModularField(SUF,SUF,reduction,merge,exactquo)
AM:=Matrix MM
x:SE:=lhs alpha
pol:SUF:=unitCanonical map(coerce,univariate(rhs alpha,x))$UPCF2
alg:MM:=reduce(monomial(1,1),pol)
B:AM := zero(n,n)
for i in 1..n repeat
  for j in 1..n repeat B(i,j):=reduce(A(i,j)::SUF,pol)
  B(i,i):= B(i,i) - alg
sol: List M :=empty()
for vec in nullSpace(B**m) repeat
  w:M:=zero(n,1)
  for i in 1..n repeat w(i,1):=tff((vec.i)::SUF,x)
  sol:=cons(w,sol)
sol
--- Generalized Eigenvectors belonging to a given eigenvalue ----
generalizedEigenvector(alpha:Eigenvalue,A:M,k:NNI,g:NNI) : List M ==
alpha case F => intRatEig(alpha::F,A,(1+k-g)::NNI)
intAlgEig(alpha::ST,A,(1+k-g)::NNI)

--- Generalized Eigenvectors belonging to a given eigenvalue ----
generalizedEigenvector(eigen :EigenForm,A:M) : List M ==
generalizedEigenvector(eigen.eigval,A,eigen.eigmult,# eigen.eigvec)

--- Generalized Eigenvectors ----
generalizedEigenvectors(A:M) : List GenEigen ==
n:= nrows A
leig:=eigenvectors A
[genEigForm(leg,A) for leg in leig]

--- eigenvectors and eigenvalues ----
eigenvectors(A:M):List(EigenForm) ==
n:=nrows A
x:=new()$SE
p:=numer charpol(A,x)
MM := ModularField(SUF,SUF,reduction,merge,exactquo)
AM:=Matrix(MM)
ratSol : List EigenForm := empty()
algSol : List EigenForm := empty()
lff:=factors factor p
for fact in lff repeat
  pol:=fact.factor
degree(pol,x)=1 =>
    vec:F :=-coefficient(pol,x,0)/coefficient(pol,x,degree(pol,x))
    ratSol:=cons([vec,fact.exponent :: NNI,
                   intRatEig(vec,A,1$NNI)]$EigenForm,ratSol)
  alpha:ST:=[x,pol]
    algSol:=cons([alpha,fact.exponent :: NNI,
                   intAlgEig(alpha,A,1$NNI)]$EigenForm,algSol)
append(ratSol,algSol)
package EF ElementaryFunction

-- ElementaryFunction.input --

)set break resume
)sys rm -f ElementaryFunction.output
)spool ElementaryFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 33
)trace EF
--R
--R
--R Parameterized constructors traced:
--R EF
--E 1

--S 2 of 33
D(cos(3*x+6*y),x)
--I
--I1<enter ElementaryFunction.cos,64 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3))) 0 . 1)
--I 1<enter ElementaryFunction.iicos,154 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3))) 0 . 1)
--I 1<enter ElementaryFunction.iisqrt2,58 :
--I 1>exit ElementaryFunction.iisqrt2,58 : ((1 #<vector 0917aab8> (1 0 . 1)) 0 . 1)
--I 1<enter ElementaryFunction.iisqrt3,59 :
--I 1>exit ElementaryFunction.iisqrt3,59 : ((1 #<vector 0917a1dc> (1 0 . 1)) 0 . 1)
--I 1<enter ElementaryFunction.specialTrigs,116 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0917aab8> (1 0 . -1)) 0 . 2)) (((1 #<vector 0917a1dc> (1 0 . 1)) 0 . 2)) (((0 . 1) 0 . 2)) (((0 . 1) 0 . 2)) (((0 . -1) 0 . 2)) (((0 . -1) 0 . 2)))
--I 1<enter ElementaryFunction.pi,46 :
--I 1>exit ElementaryFunction.pi,46 : ((1 #<vector 090c3a64> (1 0 . 1)) 0 . 1)
--I 1>exit ElementaryFunction.specialTrigs,116 : (1 . "failed")
--I 1>exit ElementaryFunction.iicos,154 : ((1 #<vector 0941ed74> (1 0 . 1)) 0 . 1)
--I1>exit ElementaryFunction.cos,64 : ((1 #<vector 0941ed74> (1 0 . 1)) 0 . 1)
--I1<enter ElementaryFunction.sin,63 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3))) 0 . 1)
--I 1<enter ElementaryFunction.iisin,152 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0941eee0> (1 0 . 3))) 0 . 1)
--I 1<enter ElementaryFunction.iisqrt2,58 :
--I 1>exit ElementaryFunction.iisqrt2,58 : ((1 #<vector 0917aab8> (1 0 . 1)) 0 . 1)
--I 1<enter ElementaryFunction.iisqrt3,59 :
--I 1>exit ElementaryFunction.iisqrt3,59 : ((1 #<vector 0917a1dc> (1 0 . 1)) 0 . 1)
--I 1<enter ElementaryFunction.specialTrigs,116 : ((1 #<vector 0941ef18> (1 0 . 6) (0 1 #<vector 0917aab8> (1 0 . -1)) 0 . 2)) (((1 #<vector 0917a1dc> (1 0 . 1)) 0 . 2)) (((0 . 1) 0 . 2)) (((0 . 1) 0 . 2)) (((0 . -1) 0 . 2)) (((0 . -1) 0 . 2)))
--I 1<enter ElementaryFunction.pi,46 :
--I 1>exit ElementaryFunction.pi,46 : ((1 #<vector 090c3a64> (1 0 . 1)) 0 . 1)
--I 1>exit ElementaryFunction.specialTrigs,116 : (1 . "failed")
--I 1>exit ElementaryFunction.iisin,152 : ((1 #<vector 0941eb60> (1 0 . 1)) 0 . 1)
--I 1>exit ElementaryFunction.sin,63 : ((1 #<vector 0941eb60> (1 0 . 1)) 0 . 1)
--R
--R (1) - 3sin(6y + 3x)
Note that both Mathematica and Maxima return this result as $3\sec(3x + 6y)^2$ but Maple returns the same result as Axiom. They are equivalent results.
Note that Mathematica and Maxima return $-3 \csc(3x + 6y)^2$ and Maple returns the same form as Axiom. They are equivalent.

— **ElementaryFunction.input** —

```
--S 8 of 33
D(cot(3*x+6*y),x)
--R
--R
--R  2
--R  (6) - 3cot(6y + 3x) - 3
--R
--E 8
Type: Expression(Integer)

--S 9 of 33
simplify ((-3*cot(6*y+3*x)^2-3) -(-3*csc(3*x+6*y)^2))
--R
--R
--R  (7) 0
--R
--E 9
Type: Expression(Integer)

--S 10 of 33
D(sec(3*x+6*y),x)
--R
--R
--R  (8) 3sec(6y + 3x)tan(6y + 3x)
--R
--E 10
Type: Expression(Integer)

--S 11 of 33
D(csc(3*x+6*y),x)
--R
--R
--R  (9) - 3cot(6y + 3x)csc(6y + 3x)
--R
--E 11
Type: Expression(Integer)

--S 12 of 33
D(asin(3*x+6*y),x)
--R
--R
--R  3
--R  ----------------------------------
--R  +------------------------+
--R  | 2 2
```
Mathematica, Maple, and Maxima give:

\[ \frac{3}{(3x+6y)^2} \sqrt{1 - \frac{1}{(3x+6y)^2}} \]

which proceeds directly from the formula for the derivative of asec:

\[ \frac{d}{dx} \text{arcsec}(x) = \frac{1}{x\sqrt{x^2 - 1}} \]
If we use the same formula for this example:

$$\frac{1}{(3x + 6y)\sqrt{(3x + 6y)^2 - 1}} \frac{d(3x + 6y)}{dx}$$

Mathematica, Maple, and Maxima give

$$\frac{-3}{(3x + 6y)^2 \sqrt{1 - \frac{1}{(3x + 6y)^2}}}$$

which is just the negative of the above result and we can see that the same analysis applies to explain the results.
Mathematica and Maxima return

\[ \frac{3}{\text{sech}(3x + 6y)^2} \]

Maple returns Axiom's answer. Both are equivalent.

--- ElementaryFunction.input ---

```plaintext
--S 21 of 33
D(tanh(3*x+6*y),x)
--R
--R
--R (19) - 3\tanh(6y + 3x) + 3
--R
--E 21
```

--- S 22 of 33
simplify ((-3*tanh(6*y+3*x)^2+3)-(3*sech(3*x+6*y)^2))
--R
Mathematica and Maxima return

\[-3csch(3 \cdot x + 6 \cdot y)^2\]

Maple returns Axiom’s answer. Both are equivalent.

— ElementaryFunction.input —
Mathematica and Maple show

\[
\frac{3}{\sqrt{1 + 3x + 6y} \sqrt{1 + 3x + 6y}}
\]

Maxima gives Axiom’s answer. Both are equivalent, just factored forms.

---

---S 28 of 33
D(acosh(3*x+6*y),x)

---R

---R 3

---R (26) -------------------------

---R +-----------------------+

---R 36y + 36x y + 9x + 1

---R Type: Expression(Integer)

---E 28

---S 29 of 33
D(atanh(3*x+6*y),x)

---R

---R 3

---R (27) - ----------------------

---R 2 2

---R 36y + 36x y + 9x - 1

---R Type: Expression(Integer)

---E 29

---S 30 of 33
D(acoth(3*x+6*y),x)

---R

---R 3

---R (28) - ----------------------

---R 2 2
Mathematica gives
\[- \frac{3}{(3x + 6y)\sqrt{\frac{1-3x-6y}{3x+6y}}(1 + 3x + 6y)}\]
Maxima gives
\[- \frac{3}{(6y + 3x)^2 \sqrt{\frac{1}{(6y+3x)^2} - 1}}\]
Maple gives
\[- \frac{3}{(3x + 6y)^2 \sqrt{\frac{1}{3x+6y} - 1} \sqrt{\frac{1}{3x+6y} + 1}}\]

Axiom cannot simplify these differences to zero but Maxima does which shows they are all equivalent answers.

---

Mathematica, Maple, and Maxima all generate the answer
\[- \frac{3}{(3x + 6y)^2 \sqrt{1 + \frac{1}{(3x+6y)^2}}}\]

Axiom cannot simplify these differences to zero but Maxima does which shows they are all equivalent answers.
D(acsch(3*x+6*y),x)
--R
--R
--R 1
--R (30) +----------------------+
--R | 2 2
--R (2y + x)|36y + 36x y + 9x + 1
--R
--R Type: Expression(Integer)
--E

)show ElementaryFunction
--R
--R ElementaryFunction(R: Join(OrderedSet, IntegralDomain), F: Join(FunctionSpace(R), RadicalCategory)) is a package constructor
--R Abbreviation for ElementaryFunction is EF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EF
--R
--R----------------------------------- Operations -----------------------------------
--R acos : F -> F
--R acot : F -> F
--R acsc : F -> F
--R asec : F -> F
--R asin : F -> F
--R atan : F -> F
--R belong? : BasicOperator -> Boolean
--R cosh : F -> F
--R cot : F -> F
--R csc : F -> F
--R exp : F -> F
--R iacos : F -> F
--R iacot : F -> F
--R iacsc : F -> F
--R iasec : F -> F
--R iasin : F -> F
--R iiatan : F -> F
--R iicos : F -> F
--R iicot : F -> F
--R iiexp : F -> F
--R iisec : F -> F
--R iisin : F -> F
--R iissqrt2 : () -> F
--R iissqrt3 : () -> F
--R iitan : F -> F
--R localReal? : F -> Boolean
--R pi : () -> F
--R sech : F -> F
--R sinh : F -> F
--R tanh : F -> F
ELEMENTARYFUNCTION.help

This package provides elementary functions over an integral domain.

\[
\begin{align*}
D(\sin(3x+6y),x) & = 3\cos(6y + 3x) \\
D(\cos(3x+6y),x) & = -3\sin(6y + 3x) \\
D(\tan(3x+6y),x) & = \frac{2}{3\tan(6y + 3x) + 3} \\
D(\cot(3x+6y),x) & = \frac{2}{-3\cot(6y + 3x) - 3} \\
D(\sec(3x+6y),x) & = 3\sec(6y + 3x)\tan(6y + 3x) \\
D(\csc(3x+6y),x) & = -3\cot(6y + 3x)\csc(6y + 3x) \\
D(\arcsin(3x+6y),x) & = \frac{3}{\sqrt{1 - (36y - 36x - 9x + 1)^2}} \\
D(\arccos(3x+6y),x) & = \frac{3}{\sqrt{1 - (36y - 36x - 9x + 1)^2}} 
\end{align*}
\]
\[
\begin{align*}
D(\text{atan}(3x+6y),x) &= \frac{2}{3} \\
\quad &= \frac{2}{36y + 36x y + 9x + 1} \\
D(\text{acot}(3x+6y),x) &= \frac{-2}{3} \\
\quad &= \frac{-2}{36y + 36x y + 9x + 1} \\
D(\text{asec}(3x+6y),x) &= \frac{1}{3} \\
\quad &= \frac{1}{(2y + x)\sqrt{36y + 36x y + 9x - 1}} \\
D(\text{acsc}(3x+6y),x) &= \frac{1}{3} \\
\quad &= \frac{1}{(2y + x)\sqrt{36y + 36x y + 9x - 1}} \\
D(\sinh(3x+6y),x) &= 3\cosh(6y + 3x) \\
D(\cosh(3x+6y),x) &= 3\sinh(6y + 3x) \\
D(\tanh(3x+6y),x) &= 2 - 3\tanh(6y + 3x) + 3 \\
D(\coth(3x+6y),x) &= 2 - 3\coth(6y + 3x) + 3 \\
D(\text{sech}(3x+6y),x) &= -3\text{sech}(6y + 3x)\tanh(6y + 3x) \\
D(\text{csch}(3x+6y),x) &= -3\coth(6y + 3x)\text{csch}(6y + 3x)
\end{align*}
\]
D(asinh(3*x+6*y),x)
3
-------------------------
| 2 2
\|36y + 36x y + 9x + 1

D(acosh(3*x+6*y),x)
3
-------------------------
| 2 2
\|36y + 36x y + 9x - 1

D(atanh(3*x+6*y),x)
3
- ----------------------
  2 2
 36y + 36x y + 9x - 1

D(acoth(3*x+6*y),x)
3
- ----------------------
  2 2
 36y + 36x y + 9x - 1

D(asech(3*x+6*y),x)
1
- -----------------------------------
| 2 2
(2y + x)\|36y - 36x y - 9x + 1

D(acsch(3*x+6*y),x)
1
- ---------------------------------
| 2 2
(2y + x)\|36y + 36x y + 9x + 1

See Also:
  o )show ElementaryFunction
ElementaryFunction (EF)

Exports:

acos    acosh    acot    acoth   asec
acsc    acsch    asec    asin    asinh
atan    atanh    belong?  cos     cosh
cot     coth     csc     csch    exp
iiacos  iiacosh  iiacot  iiacoth  iiacsc
iiascch iiasech  iiasech  iiasin  iiasinh
iiatan  iiatanh  iicos    iicosh  iicot
iiicoth iiicsc    iiicsch  iiexp   iilog
iiisec  iiisec   iiisin   iiisinh  iiisqrt2
iiisqrt3 iiitan    iiitanh  localReal?  log
operator pi    sec     sech    sin
sinh    specialTrigs tan     tanh

— package EF ElementaryFunction —

)abbrev package EF ElementaryFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 10 April 1995
++ Description:
++ Provides elementary functions over an integral domain.

ElementaryFunction(R, F):Exports == Implementation where
R:Join(OrderedSet, IntegralDomain)
F:Join(FunctionSpace R, RadicalCategory)

B ==> Boolean
L ==> List
Z ==> Integer
OP ==> BasicOperator
K ==> Kernel F
INV ==> error "Invalid argument"
Exports ==> with
exp : F -> F
  ++ exp(x) applies the exponential operator to x
log : F -> F
  ++ log(x) applies the logarithm operator to x
sin : F -> F
  ++ sin(x) applies the sine operator to x
cos : F -> F
  ++ cos(x) applies the cosine operator to x
tan : F -> F
  ++ tan(x) applies the tangent operator to x
cot : F -> F
  ++ cot(x) applies the cotangent operator to x
sec : F -> F
  ++ sec(x) applies the secant operator to x
csc : F -> F
  ++ csc(x) applies the cosecant operator to x
asin : F -> F
  ++ asin(x) applies the inverse sine operator to x
acos : F -> F
  ++ acos(x) applies the inverse cosine operator to x
atan : F -> F
  ++ atan(x) applies the inverse tangent operator to x
acot : F -> F
  ++ acot(x) applies the inverse cotangent operator to x
asec : F -> F
  ++ asec(x) applies the inverse secant operator to x
acsc : F -> F
  ++ acsc(x) applies the inverse cosecant operator to x
sinh : F -> F
  ++ sinh(x) applies the hyperbolic sine operator to x
cosh : F -> F
  ++ cosh(x) applies the hyperbolic cosine operator to x	anh : F -> F
  ++ tanh(x) applies the hyperbolic tangent operator to x
coth : F -> F
  ++ coth(x) applies the hyperbolic cotangent operator to x
sech : F -> F
  ++ sech(x) applies the hyperbolic secant operator to x
csch : F -> F
  ++ csch(x) applies the hyperbolic cosecant operator to x
asinh : F -> F
  ++ asinh(x) applies the inverse hyperbolic sine operator to x
acosh : F -> F
  ++ acosh(x) applies the inverse hyperbolic cosine operator to x
atanh : F -> F
  ++ atanh(x) applies the inverse hyperbolic tangent operator to x
acoth : F -> F
  ++ acoth(x) applies the inverse hyperbolic cotangent operator to x
asech : F -> F
++ asech(x) applies the inverse hyperbolic secant operator to x

acsch : F -> F
++ acsch(x) applies the inverse hyperbolic cosecant operator to x

pi : () -> F
++ pi() returns the pi operator

belong? : OP -> Boolean
++ belong?(p) returns true if operator p is elementary

operator: OP -> OP
++ operator(p) returns an elementary operator with the same symbol as p

-- the following should be local, but are conditional

iisqrt2 : () -> F
++ iisqrt2() should be local but conditional

iisqrt3 : () -> F
++ iisqrt3() should be local but conditional

iiexp : F -> F
++ iiexp(x) should be local but conditional

iilog : F -> F
++ iilog(x) should be local but conditional

iiisin : F -> F
++ iiisin(x) should be local but conditional

iiicos : F -> F
++ iiicos(x) should be local but conditional

iiitan : F -> F
++ iiitan(x) should be local but conditional

iiicot : F -> F
++ iiicot(x) should be local but conditional

iiisec : F -> F
++ iiisec(x) should be local but conditional

iicsc : F -> F
++ iicsc(x) should be local but conditional

iiiasin : F -> F
++ iiiasin(x) should be local but conditional

iiacos : F -> F
++ iiacos(x) should be local but conditional

iitanh : F -> F
++ iitanh(x) should be local but conditional

iiacot : F -> F
++ iiacot(x) should be local but conditional

iiasec : F -> F
++ iiasec(x) should be local but conditional

iiacsc : F -> F
++ iiacsc(x) should be local but conditional

iiisinh : F -> F
++ iiisinh(x) should be local but conditional

iicosh : F -> F
++ iicosh(x) should be local but conditional

iitanh : F -> F
++ iitanh(x) should be local but conditional

iiacoth : F -> F
++ iiacoth(x) should be local but conditional
iisech : F -> F
  ++ iisech(x) should be local but conditional
iicsch : F -> F
  ++ iicsch(x) should be local but conditional
iiasinh : F -> F
  ++ iiasinh(x) should be local but conditional
iiacosh : F -> F
  ++ iiacosh(x) should be local but conditional
iianth : F -> F
  ++ iianth(x) should be local but conditional
iicoth : F -> F
  ++ iicoth(x) should be local but conditional
iiasch : F -> F
  ++ iiasch(x) should be local but conditional
iicasch : F -> F
  ++ iicasch(x) should be local but conditional
specialTrigs : (F, L Record(func:F,pole:B)) -> Union(F, "failed")
  ++ specialTrigs(x,l) should be local but conditional
localReal?: F -> Boolean
  ++ localReal?(x) should be local but conditional

Implementation ==> add
ipi : List F -> F
iexp : F -> F
ilog : F -> F
iilog : F -> F
isin : F -> F
icos : F -> F
itan : F -> F
icot : F -> F
iisec : F -> F
icsc : F -> F
iiasin : F -> F
iacos : F -> F
iatan : F -> F
iacot : F -> F
iasec : F -> F
iacsc : F -> F
isinh : F -> F
icosh : F -> F
itanh : F -> F
icoth : F -> F
iisech : F -> F
icsch : F -> F
iiasinh : F -> F
iacosh : F -> F
iianth : F -> F
iacoth : F -> F
iasech : F -> F
iacsch : F -> F
dropfun : F -> F
kernel : F -> K
posrem : (Z, Z) -> Z
iisqrt1 : () -> F
valueOrPole : Record(func:F, pole:B) -> F

oppi := operator("pi":::Symbol)$CommonOperators
oplog := operator("log":::Symbol)$CommonOperators
opexp := operator("exp":::Symbol)$CommonOperators
opsin := operator("sin":::Symbol)$CommonOperators
opcsc := operator("csc":::Symbol)$CommonOperators
opadam := operator("asin":::Symbol)$CommonOperators
opacos := operator("acos":::Symbol)$CommonOperators
opatan := operator("atan":::Symbol)$CommonOperators
opacot := operator("acot":::Symbol)$CommonOperators
opcsc := operator("acsc":::Symbol)$CommonOperators
opsinh := operator("sinh":::Symbol)$CommonOperators
opcosh := operator("cosh":::Symbol)$CommonOperators
opatanh := operator("tanh":::Symbol)$CommonOperators
opcoth := operator("coth":::Symbol)$CommonOperators
opasinh := operator("asinh":::Symbol)$CommonOperators
opacosh := operator("acosh":::Symbol)$CommonOperators
opatanh := operator("atanh":::Symbol)$CommonOperators
opasech := operator("sech":::Symbol)$CommonOperators
opcosh := operator("csch":::Symbol)$CommonOperators
opasech := operator("csch":::Symbol)$CommonOperators

-- Pi is a domain...
Pie, iisqrt1, iisqrt2, iisqrt3: F

-- following code is conditionalized on arbitraryPrecision to recompute in
-- case user changes the precision

if R has TranscendentalFunctionCategory then
  Pie := pi()$R :: F
else
  Pie := kernel(oppi, nil()$List(F))

if R has TranscendentalFunctionCategory and R has arbitraryPrecision then
  pi() == pi()$R :: F
else
  pi() == Pie
if \( R \) has imaginary: () \( \rightarrow R \) then

\[ \text{isqrt1} := \text{imaginary()$R :: F} \]

else \( \text{isqrt1} := \text{sqrt(-1::F)} \)

if \( R \) has RadicalCategory then

\[ \text{isqrt2} := \text{sqrt(2::R)::F} \]

\[ \text{isqrt3} := \text{sqrt(3::R)::F} \]

else

\[ \text{isqrt2} := \text{sqrt(2::F)} \]

\[ \text{isqrt3} := \text{sqrt(3::F)} \]

\[ \text{iisqrt1()} == \text{isqrt1} \]

if \( R \) has RadicalCategory and \( R \) has arbitraryPrecision then

\[ \text{iisqrt2()} == \text{sqrt(2::R)::F} \]

\[ \text{iisqrt3()} == \text{sqrt(3::R)::F} \]

else

\[ \text{iisqrt2()} == \text{isqrt2} \]

\[ \text{iisqrt3()} == \text{isqrt3} \]

\[ \text{ipi l} == \text{pi()} \]

\[ \log x == \text{oplog x} \]

\[ \exp x == \text{opexp x} \]

\[ \sin x == \text{opsin x} \]

\[ \cos x == \text{opcos x} \]

\[ \tan x == \text{optan x} \]

\[ \cot x == \text{opcot x} \]

\[ \sec x == \text{opsec x} \]

\[ \csc x == \text{opcsc x} \]

\[ \text{asin x} == \text{opasin x} \]

\[ \text{acos x} == \text{opacos x} \]

\[ \text{atan x} == \text{opatan x} \]

\[ \text{acot x} == \text{opacot x} \]

\[ \text{asec x} == \text{opasec x} \]

\[ \text{acsc x} == \text{opacsc x} \]

\[ \text{sinh x} == \text{opsinh x} \]

\[ \text{cosh x} == \text{opcosh x} \]

\[ \text{tanh x} == \text{optanh x} \]

\[ \text{coth x} == \text{opcoth x} \]

\[ \text{sech x} == \text{opsech x} \]

\[ \text{csch x} == \text{opcsch x} \]

\[ \text{asinh x} == \text{opasinh x} \]

\[ \text{acosh x} == \text{opacosh x} \]

\[ \text{atanh x} == \text{opatanh x} \]

\[ \text{acoth x} == \text{opcoth x} \]

\[ \text{asech x} == \text{opasech x} \]

\[ \text{acsch x} == \text{opcsch x} \]

\[ \text{kernel x} == \text{retract(x)$K} \]

\[ \text{posrem(n, m)} == ((r := n \text{ rem m}) < 0 \Rightarrow r + m; r) \]

\[ \text{valueOrPole rec} == (\text{rec.pole} \Rightarrow \text{INV}; \text{rec.func}) \]
belong? op  == has?(op, "elem")

operator op ==
  is?(op, "pi"::Symbol) => oppi
  is?(op, "log"::Symbol) => oplog
  is?(op, "exp"::Symbol) => opexp
  is?(op, "sin"::Symbol) => opsín
  is?(op, "cos"::Symbol) => opcós
  is?(op, "tan"::Symbol) => optan
  is?(op, "cot"::Symbol) => opcot
  is?(op, "sec"::Symbol) => opsec
  is?(op, "csc"::Symbol) => opcsc
  is?(op, "asin"::Symbol) => opasin
  is?(op, "acos"::Symbol) => opacos
  is?(op, "atan"::Symbol) => opatan
  is?(op, "acot"::Symbol) => opacot
  is?(op, "asec"::Symbol) => opasec
  is?(op, "acsc"::Symbol) => opacsch
  is?(op, "sinh"::Symbol) => opsinh
  is?(op, "cosh"::Symbol) => opcosh
  is?(op, "tanh"::Symbol) => opatan
  is?(op, "coth"::Symbol) => opcoth
  is?(op, "sech"::Symbol) => opsech
  is?(op, "csch"::Symbol) => opcsch
  is?(op, "asinh"::Symbol) => opasinh
  is?(op, "acosh"::Symbol) => opacosh
  is?(op, "atanh"::Symbol) => opatanh
  is?(op, "acoth"::Symbol) => opacoth
  is?(op, "asech"::Symbol) => opasech
  is?(op, "acsch"::Symbol) => opacsch
error "Not an elementary operator"

dropfun x ==
  (k := retractIfCan(x)@Union(K, "failed")) case "failed") or
  empty?(argument(k::K)) => 0
  first argument(k::K)

if R has RetractableTo Z then
  specialTrigs(x, values) ==
    (r := retractIfCan(y := x/pi())@Union(Fraction Z, "failed")) case "failed"
    q := r::Fraction(Integer)
    m := minIndex values
    (n := retractIfCan(q)@Union(Z, "failed")) case Z =>
      even?(n::Z) => valueOrPole(values.m)
      valueOrPole(values.(m+1))
    (n := retractIfCan(2*q)@Union(Z, "failed")) case Z =>
      one?(s := posrem(n::Z, 4)) => valueOrPole(values.(m+2))
      valueOrPole(values.(m+3))
(n := retractIfCan(3*q)@Union(Z, "failed")) case Z =>
- one?(s := posrem(n::Z, 6)) => valueOrPole(values.(m+4))
  (s := posrem(n::Z, 6)) = 1 => valueOrPole(values.(m+4))
  s = 2 => valueOrPole(values.(m+5))
  s = 4 => valueOrPole(values.(m+6))
  valueOrPole(values.(m+7))

(n := retractIfCan(4*q)@Union(Z, "failed")) case Z =>
- one?(s := posrem(n::Z, 8)) => valueOrPole(values.(m+8))
  (s := posrem(n::Z, 8)) = 1 => valueOrPole(values.(m+8))
  s = 3 => valueOrPole(values.(m+9))
  s = 5 => valueOrPole(values.(m+10))
  valueOrPole(values.(m+11))

(n := retractIfCan(6*q)@Union(Z, "failed")) case Z =>
- one?(s := posrem(n::Z, 12)) => valueOrPole(values.(m+12))
  (s := posrem(n::Z, 12)) = 1 => valueOrPole(values.(m+12))
  s = 5 => valueOrPole(values.(m+13))
  s = 7 => valueOrPole(values.(m+14))
  valueOrPole(values.(m+15))
"failed"

else specialTrigs(x, values) == "failed"

isin x ==
  zero? x => 0
  y := dropfun x
  is?(x, opasin) => y
  is?(x, opacos) => sqrt(1 - y**2)
  is?(x, opatan) => y / sqrt(1 + y**2)
  is?(x, opacot) => inv sqrt(1 + y**2)
  is?(x, opasec) => inv y
  h := inv(2::F)
  s2 := h * iisqrt2()
  s3 := h * iisqrt3()
  u := specialTrigs(x, [[0,false], [0,false], [1,false], [-1,false],
                         [s3,false], [s3,false], [-s3,false], [-s3,false],
                         [s2,false], [s2,false], [-s2,false], [-s2,false],
                         [h,false], [h,false], [-h,false], [-h,false]])
  u case F => u :: F
  kernel(opsin, x)

icos x ==
  zero? x => 1
  y := dropfun x
  is?(x, opasin) => sqrt(1 - y**2)
  is?(x, opacos) => y
  is?(x, opatan) => inv sqrt(1 + y**2)
  is?(x, opacot) => y / sqrt(1 + y**2)
  is?(x, opasec) => inv y
  is?(x, opacsc) => sqrt(y**2 - 1) / y
h := inv(2::F)
s2 := h * iisqrt2()
s3 := h * iisqrt3()
u := specialTrigs(x, [[1,false],[-1,false], [0,false], [0,false],
[h,false],[-h,false],[h,false],[h,false],[-h,false],[h,false],
[s2,false],[-s2,false],[s2,false],[s2,false],[s2,false],[s2,false],
[s3,false],[-s3,false],[s3,false],[s3,false],[s3,false],
[s3,false], [s3,false],[s3,false],[s3,false],
[s3,false],[s3,false],[s3,false], [s3,false],
[s3,false],[s3,false],[s3,false],[s3,false])

u case F => u :: F
kernel(opcos, x)

itan x ==
zero? x => 0
y := dropfun x
is?(x, opasin) => y / sqrt(1 - y**2)
is?(x, opacos) => sqrt(1 - y**2) / y
is?(x, opatan) => y
is?(x, opacot) => inv y
is?(x, opasec) => sqrt(y**2 - 1)
is?(x, opacsc) => inv sqrt(y**2 - 1)
s33 := (s3 := iisqrt3()) / (3::F)
u := specialTrigs(x, [0,false], [0,false], [0,true], [0,true],
[s3,false], [-s3,false], [s3,false], [-s3,false],
[1,false], [-1,false], [1,false], [-1,false],
[s33,false], [-s33,false], [s33,false], [-s33,false])

u case F => u :: F
kernel(optan, x)

icot x ==
zero? x => INV
y := dropfun x
is?(x, opasin) => sqrt(1 - y**2) / y
is?(x, opacos) => y / sqrt(1 - y**2)
is?(x, opatan) => inv y
is?(x, opacot) => y
is?(x, opasec) => inv sqrt(y**2 - 1)
is?(x, opacsc) => y
s33 := (s3 := iisqrt3()) / (3::F)
u := specialTrigs(x, [0,true], [0,true], [0,false], [0,false],
[s33,false], [-s33,false], [s33,false], [-s33,false],
[1,true], [-1,true], [1,true], [-1,true],
[s33,false], [-s33,false], [s33,false], [-s33,false])

u case F => u :: F
kernel(opcot, x)

isec x ==
zero? x => 1
y := dropfun x
is?(x, opasin) => inv sqrt(1 - y**2)
is?(x, opacos) => inv y
is?(x, opatan) => sqrt(1 + y**2)
544
CHAPTER 6. CHAPTER E

is?(x, opacot) => sqrt(1 + y**2) / y
is?(x, opasec) => y
is?(x, opacsc) => y / sqrt(y**2 - 1)
s2 := iisqrt2()
s3 := 2 * iisqrt3() / (3::F)
h := 2::F
u := specialTrigs(x, 
[[1,false],[1,false],[0,true],[0,true],
 [h,false], [-h,false], [-h,false], [h,false],
 [s2,false], [-s2,false], [-s2,false], [s2,false],
 [s3,false], [-s3,false], [-s3,false], [s3,false]])
u case F => u :: F
kernel(opsec, x)

icsc x ==
zero? x => INV
y := dropfun x
is?(x, opasin) => inv y
is?(x, opacos) => inv sqrt(1 - y**2)
is?(x, opatan) => sqrt(1 + y**2) / y
is?(x, opacot) => sqrt(1 + y**2)
is?(x, opasec) => y / sqrt(y**2 - 1)
is?(x, opacsc) => y
s2 := iisqrt2()
s3 := 2 * iisqrt3() / (3::F)
h := 2::F
u := specialTrigs(x, 
[[0,true], [0,true], [1,false], [-1,false],
 [s3,false], [s3,false], [-s3,false], [s3,false],
 [s2,false], [-s2,false], [-s2,false], [s2,false],
 [h,false], [h,false], [-h,false], [-h,false]])
u case F => u :: F
kernel(opcsc, x)

iasin x ==
zero? x => 0
-- one? x => pi() / (2::F)
(x = 1) => pi() / (2::F)
(x = -1) => - pi() / (2::F)
y := dropfun x
is?(x, opsin) => y
is?(x, opcos) => pi() / (2::F) - y
kernel(opasin, x)

iacos x ==
zero? x => pi() / (2::F)
-- one? x => 0
(x = 1) => 0
(x = -1) => pi()
y := dropfun x
is?(x, opsin) => pi() / (2::F) - y
is?(x, opcos) => y
kernel(opacos, x)

iatan x ==
  zero? x => 0
-- one? x => pi() \div (4::F)
(x = 1) => pi() \div (4::F)
x = -1 => - pi() \div (4::F)
x = (r3:=iisqrt3()) => pi() \div (3::F)
-- one?(x*r3) => pi() \div (6::F)
(x*r3) = 1 => pi() \div (6::F)
y := dropfun x
is?(x, optan) => y
is?(x, opcot) => pi() \div (2::F) - y
kernel(opatan, x)

iacot x ==
  zero? x => pi() \div (2::F)
-- one? x => pi() \div (4::F)
(x = 1) => pi() \div (4::F)
x = -1 => 3 * pi() \div (4::F)
x = (r3:=iisqrt3()) => pi() \div (6::F)
x = -r3 => 5 * pi() \div (6::F)
-- one?(xx:=x*r3) => pi() \div (3::F)
(xx:=x*r3) = 1 => pi() \div (3::F)
xx = -1 => 2* pi() \div (3::F)
y := dropfun x
is?(x, optan) => pi() \div (2::F) - y
is?(x, opcot) => y
kernel(opcatan, x)

iasec x ==
  zero? x => INV
-- one? x => 0
(x = 1) => 0
x = -1 => pi()
y := dropfun x
is?(x, opsec) => y
is?(x, opcsc) => pi() \div (2::F) - y
kernel(opasec, x)

iacsc x ==
  zero? x => INV
-- one? x => pi() \div (2::F)
(x = 1) => pi() \div (2::F)
x = -1 => - pi() \div (2::F)
y := dropfun x
is?(x, opsec) => pi() \div (2::F) - y
is?(x, opcsc) => y
kernel(opascsc, x)
\[
isinh x ==
\]
zero? x => 0
y := dropfun x
is?(x, opasinh) => y
is?(x, opacosh) => sqrt(y**2 - 1)
is?(x, opatanh) => y / sqrt(1 - y**2)
is?(x, opacoth) => - inv sqrt(y**2 - 1)
is?(x, opasech) => sqrt(1 - y**2) / y
is?(x, opacsch) => inv y
kernel(opsinh, x)

\[
icosh x ==
\]
zero? x => 1
y := dropfun x
is?(x, opasinh) => sqrt(y**2 + 1)
is?(x, opacosh) => y
is?(x, opatanh) => inv sqrt(1 - y**2)
is?(x, opacoth) => y / sqrt(y**2 - 1)
is?(x, opasech) => inv y
is?(x, opacsch) => sqrt(y**2 + 1) / y
kernel(opcosh, x)

\[
itanh x ==
\]
zero? x => 0
y := dropfun x
is?(x, opasinh) => y / sqrt(y**2 + 1)
is?(x, opacosh) => sqrt(y**2 - 1) / y
is?(x, opatanh) => y
is?(x, opacoth) => inv y
is?(x, opasech) => sqrt(1 - y**2)
is?(x, opacsch) => inv sqrt(y**2 + 1)
kernel(optanh, x)

\[
icoth x ==
\]
zero? x => INV
y := dropfun x
is?(x, opasinh) => sqrt(y**2 + 1) / y
is?(x, opacosh) => y / sqrt(y**2 - 1)
is?(x, opatanh) => inv y
is?(x, opacoth) => y
is?(x, opasech) => inv sqrt(1 - y**2)
is?(x, opacsch) => sqrt(y**2 + 1)
kernel(opcoth, x)

\[
isech x ==
\]
zero? x => 1
y := dropfun x
is?(x, opasinh) => inv sqrt(y**2 + 1)
is?(x, opacosh) => inv y
is?(x, opatanh) => sqrt(1 - y**2)
is?(x, opacth) => sqrt(y**2 - 1) / y
is?(x, opasech) => y
is?(x, opacsch) => y / sqrt(y**2 + 1)
kernel(opsech, x)

icsch x ==
zero? x => INV
y := dropfun x
is?(x, opasinh) => inv y
is?(x, opacosh) => inv sqrt(y**2 - 1)
is?(x, optanh) => inv (1 - y**2) / y
is?(x, opcoth) => - sqrt(y**2 - 1)
is?(x, opasech) => y / sqrt(1 - y**2)
is?(x, opacsch) => y
kernel(opcsch, x)

iasinh x ==
is?(x, opsinh) => first argument kernel x
kernel(opasinh, x)

iacosh x ==
is?(x, opcosh) => first argument kernel x
kernel(opacosh, x)

iation x ==
is?(x, optanh) => first argument kernel x
kernel(opatanh, x)

iacth x ==
is?(x, opcoth) => first argument kernel x
kernel(opacoth, x)

iasech x ==
is?(x, opsech) => first argument kernel x
kernel(opasech, x)

iacsch x ==
is?(x, opcsch) => first argument kernel x
kernel(opacsch, x)

iexp x ==
zero? x => 1
is?(x, oplog) => first argument kernel x
x < 0 and empty? variables x => inv iexp(-x)
h := inv(2::F)
i := iisqrt1()
s2 := h * iisqrt2()
s3 := h * iisqrt3()
u := specialTrigs(x / i, [[1,false],[-1,false],[1, false], [-1,false],
[h + i * s3,false], [-h + i * s3, false], [-h - i * s3, false],

$[h - i \cdot s3, \text{false}], [s2 + i \cdot s2, \text{false}], [-s2 + i \cdot s2, \text{false}],$ 
$[-s2 - i \cdot s2, \text{false}], [-s2 + i \cdot s2, \text{false}], [s2 - i \cdot s2, \text{false}],$ 
$[s3 + i \cdot h, \text{false}], [-s3 + i \cdot h, \text{false}], [-s3 - i \cdot h, \text{false}], [s3 - i \cdot h, \text{false}]]$

u case F => u :: F

kernel(opexp, x)

-- THIS DETERMINES WHEN TO PERFORM THE log exp f -> f SIMPLIFICATION
-- CURRENT BEHAVIOR:
-- IF R IS COMPLEX(S) THEN ONLY ELEMENTS WHICH ARE RETRACTABLE TO R
-- AND EQUAL TO THEIR CONJUGATES ARE DEEMED REAL (OVERRESTRICTIVE FOR NOW)
-- OTHERWISE (e.g. R = INT OR FRAC INT), ALL THE ELEMENTS ARE DEEMED REAL

if (R has imaginary: () -> R) and (R has conjugate: R -> R) then
localReal? x ==
  (u := retractIfCan(x)@Union(R, "failed")) case R
  and (u::R) = conjugate(u::R)
else localReal? x == true

iiilog x ==
  zero? x => INV
  one? x => 0
  (x = 1) => 0
  (u := isExpt(x, opexp)) case Record(var:K, exponent:Integer) =>
    rec := u::Record(var:K, exponent:Integer)
    arg := first argument(rec.var);
    localReal? arg => rec.exponent * first argument(rec.var);
    ilog x
ilog x

ilog x ==
  ((num1 := one?(num := numer x)) or num = -1) and (den := denom x) ^= 1
  ((num1 := ((num := numer x) = 1)) or num = -1) and (den := denom x) ^= 1
  and empty? variables x => - kernel(oplog, (num1 => den; -den)::F)
  kernel(oplog, x)

if R has ElementaryFunctionCategory then
iiilog x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iiilog x
  log(r::R)::F
iiexp x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iexp x
  exp(r::R)::F
else
  iiilog x == iiilog x
  iiexp x == iexp x

if R has TrigonometricFunctionCategory then
iisin x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isin x
  sin(r::R)::F

iicos x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icos x
  cos(r::R)::F

iitan x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => itan x
  tan(r::R)::F

iicot x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icot x
  cot(r::R)::F

iisec x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isec x
  sec(r::R)::F

iicsc x ==
  (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icsc x
  csc(r::R)::F

else
  iisin x == isin x
  iicos x == icos x
  iitan x == itan x
  iicot x == icot x
  iisec x == isec x
  iicsc x == icsc x

if R has ArcTrigonometricFunctionCategory then
  iiasin x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasin x
    asin(r::R)::F

  iiacos x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacos x
    acos(r::R)::F

  iiatan x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iatan x
    atan(r::R)::F

  iiacot x ==
    (r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iiacot x
    acot(r::R)::F

  iiasec x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasec x
asec(r::R)::F

iiacsc x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacsc x
acsc(r::R)::F

else
iiasin x == iasin x
iiacos x == iacos x
iiatan x == iatan x
iiacot x == iacot x
iiasec x == iasec x
iiacsc x == iacsc x

if R has HyperbolicFunctionCategory then
iisinh x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isinh x
sinh(r::R)::F

iicosh x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icosh x
cosh(r::R)::F

iitanh x ==
(r:=retractIfCan(x)@ Union(R, "failed")) case "failed" => itanh x
tanh(r::R)::F

iicoth x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icoth x
coth(r::R)::F

iisech x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => isech x
sech(r::R)::F

iicsch x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => icsch x
csch(r::R)::F

else
iisinh x == isinh x
iicosh x == icosh x
iitanh x == tanh x
iicoth x == coth x
iisech x == sech x
iicsch x == csch x

if R has ArcHyperbolicFunctionCategory then
iiasinh x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasinh x asinh(r::R)::F

iiacosh x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacosh x acosh(r::R)::F

iiatanh x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iatanh x atanh(r::R)::F

iiacoth x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacoth x acoth(r::R)::F

iiasech x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iasech x asech(r::R)::F

iiacsch x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iacsch x acsch(r::R)::F

else

  iiasinh x == iasinh x
  iiacosh x == iacosh x
  iiatanh x == iatanh x
  iiacoth x == iacoth x
  iiasech x == iasech x
  iiacsch x == iacsch x

import BasicOperatorFunctions1(F)
evaluate(oppi, ipi)
evaluate(oplog, iilog)
evaluate(opexp, iiexp)
evaluate(opsin, iisin)
evaluate(opcos, iicos)
evaluate(optan, iitan)
evaluate(opcot, iicot)
evaluate(opsec, iisec)
evaluate(opcsc, iicsc)
evaluate(opasin, iiasin)
evaluate(opacos, iiacos)
evaluate(opatan, iiatan)
evaluate(opacot, iiacot)
evaluate(opasec, iiasec)
evaluate(opacsc, iiacsc)
evaluate(opsinh, iisinh)
evaluate(opcosh, iicosh)
evaluate(optanh, iitanh)
evaluate(opcoth, iicoth)
evaluate(opsech, iisech)
evaluate(opcsch, iicsch)
evaluate(opasinh, iiasinh)
evaluate(opacosh, iiacosh)
evaluate(opatanh, iiatanh)
evaluate(opacoth, iiacoth)
evaluate(opasech, iiasch)
evaluate(opacsch, iiacsch)
derivative(opexp, exp)
derivative(oplog, inv)
derivative(opsin, cos)
derivative(opcos, (x:F):F +-> - sin x)
derivative(optan, (x:F):F +-> 1 + tan(x)**2)
derivative(opcot, (x:F):F +-> - 1 - cot(x)**2)
derivative(opsec, (x:F):F +-> tan(x) * sec(x))
derivative(opcsc, (x:F):F +-> - cot(x) * csc(x))
derivative(opasin, (x:F):F +-> inv sqrt(1 - x**2))
derivative(opacos, (x:F):F +-> - inv sqrt(1 - x**2))
derivative(opatan, (x:F):F +-> inv(1 + x**2))
derivative(opacot, (x:F):F +-> - inv(1 + x**2))
derivative(opasec, (x:F):F +-> inv(x * sqrt(x**2 - 1)))
derivative(opacsc, (x:F):F +-> - inv(x * sqrt(x**2 - 1)))
derivative(opsinh, cosh)
derivative(opcosh, sinh)
derivative(optanh, (x:F):F +-> 1 - tanh(x)**2)
derivative(opcoth, (x:F):F +-> 1 - coth(x)**2)
derivative(opsech, (x:F):F +-> - tanh(x) * sech(x))
derivative(opcsch, (x:F):F +-> - coth(x) * csch(x))
derivative(opasinh, (x:F):F +-> inv sqrt(1 + x**2))
derivative(opacosh, (x:F):F +-> inv sqrt(x**2 - 1))
derivative(opatanh, (x:F):F +-> inv(1 - x**2))
derivative(opcoth, (x:F):F +-> inv(1 - x**2))
derivative(opasech, (x:F):F +-> - inv(x * sqrt(1 - x**2)))
derivative(opacsch, (x:F):F +-> - inv(x * sqrt(1 + x**2)))
package DEFINTEF ElementaryFunctionDefiniteIntegration

--- ElementaryFunctionDefiniteIntegration.input ---

)set break resume
)sys rm -f ElementaryFunctionDefiniteIntegration.output
)spool ElementaryFunctionDefiniteIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionDefiniteIntegration
--R
--R ElementaryFunctionDefiniteIntegration(R: Join(EuclideanDomain,OrderedSet,CharacteristicZero,Retractable?,Ring,IntegralDomain,Retractable,Field,IntegralDomain)()) -> Union(f1: OrderedCompletion(F),f2: List(OrderedCompletion(F)),fail: failed,pole: potentialPole)
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DEFINTEF
--R
--R------------------------------------------------- Operations --------------------------
--R innerint : (F,Symbol,OrderedCompletion(F),OrderedCompletion(F),Boolean) -> Union(f1: OrderedCompletion(F),f2: List(OrderedCompletion(F)),fail: failed,pole: potentialPole)
--R integrate : (F,SegmentBinding(OrderedCompletion(F))) -> Union(f1: OrderedCompletion(F),f2: List(OrderedCompletion(F)),fail: failed,pole: potentialPole)
--R integrate : (F,SegmentBinding(OrderedCompletion(F)),String) -> Union(f1: OrderedCompletion(F),f2: List(OrderedCompletion(F)),fail: failed,pole: potentialPole)
--R
--E 1

)spool
)lisp (bye)

-----

--- ElementaryFunctionDefiniteIntegration.help ---

====================================================================
ElementaryFunctionDefiniteIntegration examples
====================================================================

ElementaryFunctionDefiniteIntegration provides functions to compute definite integrals of elementary functions.

See Also:
  o )show ElementaryFunctionDefiniteIntegration

-----
ElementaryFunctionDefiniteIntegration (DEFINTEF)

Exports:
innerint integrate integrate

— package DEFINTEF ElementaryFunctionDefiniteIntegration —

)abbrev package DEFINTEF ElementaryFunctionDefiniteIntegration
++ Author: Manuel Bronstein
++ Date Created: 14 April 1992
++ Date Last Updated: 2 February 1993
++ Description:
++ \spadtype{ElementaryFunctionDefiniteIntegration}
++ provides functions to compute definite
++ integrals of elementary functions.

ElementaryFunctionDefiniteIntegration(R, F): Exports == Implementation where
R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
AlgebraicallyClosedFunctionSpace R)
B ==> Boolean
SE ==> Symbol
Z ==> Integer
P ==> SparseMultivariatePolynomial(R, K)
K ==> Kernel F
UP ==> SparseUnivariatePolynomial F
OFE ==> OrderedCompletion F
U ==> Union(f1:OFE, f2:List OFE, fail:"failed", pole:"potentialPole")

Exports == with
    integrate: (F, SegmentBinding OFE) -> U
      ++ integrate(f, x = a..b) returns the integral of
      ++ \spad{\int f(x)dx} from a to b.
      ++ Error: if f has a pole for x between a and b.
    integrate: (F, SegmentBinding OFE, String) -> U
++ integrate(f, x = a..b, "noPole") returns the
++ integral of \spad{f(x)dx} from a to b.
++ If it is not possible to check whether f has a pole for x
++ between a and b (because of parameters), then this function
++ will assume that f has no such pole.
++ Error: if f has a pole for x between a and b or
++ if the last argument is not "noPole".

innerint: (F, SE, OFE, OFE, B) -> U
++ innerint(f, x, a, b, ignore?) should be local but conditional

Implementation ==> add
import ElementaryFunctionSign(R, F)
import DefiniteIntegrationTools(R, F)
import FunctionSpaceIntegration(R, F)

polyIfCan : (P, K) -> Union(UP, "failed")
int : (F, SE, OFE, OFE, B) -> U
nopole : (F, SE, K, OFE, OFE) -> U
checkFor0 : (P, K, OFE, OFE) -> Union(B, "failed")
checkSMP : (P, SE, K, OFE, OFE) -> Union(B, "failed")
checkForPole: (F, SE, K, OFE, OFE) -> Union(B, "failed")
posit : (F, SE, K, OFE, OFE) -> Union(B, "failed")
negat : (F, SE, K, OFE, OFE) -> Union(B, "failed")
moreThan : (OFE, Fraction Z) -> Union(B, "failed")

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
and F has SpecialFunctionCategory then
    import PatternMatchIntegration(R, F)

innerint(f, x, a, b, ignor?) ==
    ((u := int(f, x, a, b, ignor?)) case f1) or (u case f2)
    or ((v := pmintegrate(f, x, a, b)) case "failed") => u
    [v::F::OFE]

else
    innerint(f, x, a, b, ignor?) == int(f, x, a, b, ignor?)

integrate(f:F, s:SegmentBinding OFE) ==
    innerint(f, variable s, lo segment s, hi segment s, false)

integrate(f:F, s:SegmentBinding OFE, str:String) ==
    innerint(f, variable s, lo segment s, hi segment s, ignore? str)

int(f, x, a, b, ignor?) ==
    a = b => [0::OFE]
    k := kernel(x)@Kernel(F)
    (z := checkForPole(f, x, k, a, b)) case "failed" =>
        ignor? => nopole(f, x, k, a, b)
        ["potentialPole"]
    z::B => error "integrate: pole in path of integration"
nopole(f, x, k, a, b)

checkForPole(f, x, k, a, b) ==
((u := checkFor0(d := denom f, k, a, b)) case "failed") or (u:B) => u
((u := checkSMP(d, x, k, a, b)) case "failed") or (u:B) => u

checkSMP(numer f, x, k, a, b)

-- true if p has a zero between a and b exclusive
checkFor0(p, x, a, b) ==
(u := polyIfCan(p, x)) case UP => checkForZero(u:UP, a, b, false)
(v := isTimes p) case List(P) =>
  for t in v::List(P) repeat
    ((w := checkFor0(t, x, a, b)) case "failed") or (w:B) => return w
  false
(r := retractIfCan(p)@Union(K, "failed")) case "failed" => "failed"

k := r::K

-- functions with no real zeros
is?(k, "exp":SE) or is?(k, "acot":SE) or is?(k, "cosh":SE) => false

-- special case for log
is?(k, "log":SE) =>
  (w := moreThan(b, 1)) case "failed" or not(w:B) => w
  moreThan(-a, -1)
  "failed"

-- returns true if a > b, false if a < b, "failed" if can’t decide
moreThan(a, b) ==
(r := retractIfCan(a)@Union(F, "failed")) case "failed" => -- infinite
  whatInfinity(a) > 0
(u := retractIfCan(r::F)@Union(Fraction Z, "failed")) case "failed" =>
  "failed"
  u::Fraction(Z) > b

-- true if p has a pole between a and b
checkSMP(p, x, k, a, b) ==
(u := polyIfCan(p, k)) case UP => false
(v := isTimes p) case List(P) =>
  for t in v::List(P) repeat
    ((w := checkSMP(t, x, k, a, b)) case "failed") or (w:B) => return w
  false
(v := isPlus p) case List(P) =>
  n := 0 -- number of summand having a pole
  for t in v::List(P) repeat
    (w := checkSMP(t, x, k, a, b)) case "failed" => return w
    if w:B then n := n + 1
  zero? n => false -- no summand has a pole
  one? n => true -- only one summand has a pole
  (n = 1) => true -- only one summand has a pole
  "failed" -- at least 2 summands have a pole
(r := retractIfCan(p)@Union(K, "failed")) case "failed" => "failed"
kk := r::K
-- nullary operators have no poles

nullary? operator kk => false

f := first argument kk

-- functions which are defined over all the reals:
is?(kk, "exp"::SE) or is?(kk, "sin"::SE) or is?(kk, "cos"::SE)
or is?(kk, "sinh"::SE) or is?(kk, "cosh"::SE) or is?(kk, "tanh"::SE)
or is?(kk, "sech"::SE) or is?(kk, "atan"::SE) or is?(kk, "acot"::SE)
or is?(kk, "asinh"::SE) => checkForPole(f, x, k, a, b)

-- functions which are defined on (-1,+1):

is?(kk, "asin"::SE) or is?(kk, "acos"::SE) or is?(kk, "atanh"::SE) =>

((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w

((w := posit(f - 1, x, k, a, b)) case "failed") or (w::B) => w

negat(f + 1, x, k, a, b)

-- functions which are defined on (+1, +infty):

is?(kk, "acosh"::SE) =>

((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w

negat(f - 1, x, k, a, b)

-- functions which are defined on (0, +infty):

is?(kk, "log"::SE) =>

((w := checkForPole(f, x, k, a, b)) case "failed") or (w::B) => w

negat(f, x, k, a, b)

"failed"

-- returns true if it is certain that f takes at least one strictly positive
-- value for x in (a,b), false if it is certain that f takes no strictly
-- positive value in (a,b), "failed" otherwise

-- f must be known to have no poles in (a,b)

posit(f, x, k, a, b) ==

z :=

(r := retractIfCan(a)@Union(F, "failed")) case "failed" => sign(f, x, a)
sign(f, x, r::F, "right")
(b1 := z case Z) and z::Z > 0 => true

z :=

(r := retractIfCan(b)@Union(F, "failed")) case "failed" => sign(f, x, b)
sign(f, x, r::F, "left")
(b2 := z case Z) and z::Z > 0 => true

b1 and b2 =>

((w := checkFor0(numer f, k, a, b)) case "failed") or (w::B) => "failed"
false
"failed"

-- returns true if it is certain that f takes at least one strictly negative
-- value for x in (a,b), false if it is certain that f takes no strictly
-- negative value in (a,b), "failed" otherwise

-- f must be known to have no poles in (a,b)

negat(f, x, k, a, b) ==

z :=

(r := retractIfCan(a)@Union(F, "failed")) case "failed" => sign(f, x, a)
sign(f, x, r::F, "right")
(b1 := z case Z) and z::Z < 0 => true
z :=
  (r := retractIfCan(b) Union(F, "failed")) case "failed" => sign(f, x, b)
  sign(f, x, r::F, "left")
(b2 := z case Z) and z::Z < 0 => true
b1 and b2 =>
  ((w := checkFor0(numer f, k, a, b)) case "failed") or (w::B) => "failed"
false
"failed"

-- returns a UP if p is only a poly w.r.t. the kernel x
polyIfCan(p, x) ==
  q := univariate(p, x)
  ans:UP := 0
  while q ^= 0 repeat
    member?(x, tower(c := leadingCoefficient(q)::F)) => return "failed"
    ans := ans + monomial(c, degree q)
    q := reductum q
  ans

-- integrate f for x between a and b assuming that f has no pole in between
nopole(f, x, k, a, b) ==
  (u := integrate(f, x)) case F =>
    (v := computeInt(k, u::F, a, b, false)) case "failed" => ["failed"]
    [v::OFE]
  ans := empty()$List(OFE)
  for g in u::List(F) repeat
    (v := computeInt(k, g, a, b, false)) case "failed" => return ["failed"]
    ans := concat_!(ans, [v::OFE])
  [ans]

-----

— DEFINTEF.dotabb —

"DEFINTEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DEFINTEF"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"DEFINTEF" -> "ACFS"

-----

package LODEEF ElementaryFunctionLODESolver

— ElementaryFunctionLODESolver.input —

)set break resume
ElementaryFunctionLODESolver provides the top-level functions for finding closed form solutions of linear ordinary differential equations and initial value problems.

See Also:
  o )show ElementaryFunctionLODESolver
 CHAPTER 6. CHAPTER E

ElementaryFunctionLODESolver (LODEEF)

Exports:
solve

— package LODEEF ElementaryFunctionLODESolver —

)abbrev package LODEEF ElementaryFunctionLODESolver
++ Author: Manuel Bronstein
++ Date Created: 3 February 1994
++ Date Last Updated: 9 March 1994
++ Description:
++ \spad{ElementaryFunctionLODESolver} provides the top-level
++ functions for finding closed form solutions of linear ordinary
++ differential equations and initial value problems.

ElementaryFunctionLODESolver(R, F, L):Exports == Implementation where
R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
    LinearlyExplicitRingOver Integer, CharacteristicZero)
F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
    PrimitiveFunctionCategory)
L: LinearOrdinaryDifferentialOperatorCategory F

SY ==> Symbol
N ==> NonNegativeInteger
K ==> Kernel F
V ==> Vector F
M ==> Matrix F
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
UPUP ==> SparseUnivariatePolynomial RF
P ==> SparseMultivariatePolynomial(R, K)
P2 ==> SparseMultivariatePolynomial(P, K)
LQ ==> LinearOrdinaryDifferentialOperator1 RF
REC ==> Record(particular: F, basis: List F)
U ==> Union(REC, "failed")
ALGOP ==> "\%alg"
Exports ==> with
solve: (L, F, SY) -> U
  ++ solve(op, g, x) returns either a solution of the ordinary differential
  ++ equation \spad{op y = g} or "failed" if no non-trivial solution can be
  ++ found; When found, the solution is returned in the form
  ++ \spad{\{h, \{b1, ..., bm\}\}} where \spad{h} is a particular solution and
  ++ and \spad{\{b1, ..., bm\}} are linearly independent solutions of the
  ++ associated homogenous equation \spad{op y = 0}.
  ++ A full basis for the solutions of the homogenous equation
  ++ is not always returned, only the solutions which were found;
  ++ \spad{x} is the dependent variable.
solve: (L, F, SY, F, List F) -> Union(F, "failed")
  ++ solve(op, g, x, a, \{y0, ..., ym\}) returns either the solution
  ++ of the initial value problem \spad{op y = g, y(a) = y0, y'(a) = y1, ...}
  ++ or "failed" if the solution cannot be found;
  ++ \spad{x} is the dependent variable.

Implementation ==> add
import Kovacic(F, UP)
import ODETools(F, L)
import RationalLODE(F, UP)
import RationalRicDE(F, UP)
import ODEIntegration(R, F)
import ConstantLODE(R, F, L)
import IntegrationTools(R, F)
import ReductionOfOrder(F, L)
import ReductionOfOrder(RF, LQ)
import PureAlgebraicIntegration(R, F, L)
import FunctionSpacePrimitiveElement(R, F)
import LinearSystemMatrixPackage(F, V, V, M)
import SparseUnivariatePolynomialFunctions2(RF, F)
import FunctionSpaceUnivariatePolynomialFactor(R, F, UP)
import LinearOrdinaryDifferentialOperatorFactorizer(F, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

upmp : (P, List K) -> P2
downmp : (P2, List K, List P) -> P
xpart : (F, SY) -> F
smpxpart : (P, SY, List K, List P) -> P
multint : (F, List F, SY) -> F
ulodo : (L, K) -> LQ
firstOrder : (F, F, F, SY) -> REC
rfSolve : (L, F, K, SY) -> U
ratlogsol : (LQ, List RF, K, SY) -> List F
expsls : (LQ, K, SY) -> List F
homosolve : (L, LQ, List RF, K, SY) -> List F
homosolve1 : (L, List F, K, SY) -> List F
norf1 : (L, K, SY, N) -> List F
kovode : (LQ, K, SY) -> List F

doVarParams: (L, F, List F, SY) -> U

localmap : (F -> F, L) -> L

algSolve : (L, F, K, List K, SY) -> U

palgSolve : (L, F, K, K, SY) -> U

lastChance : (L, F, SY) -> U

diff := D()$L

smpxpart(p, x, l, lp) == downmp(primitivePart upmp(p, l), l, lp)
downmp(p, l, lp) == ground eval(p, l, lp)
homosolve(lf, op, sols, k, x) == homosolve1(lf, ratlogsol(op,sols,k,x),k,x)

-- left hand side has algebraic (not necessarily pure) coefficients
algSolve(op, g, k, l, x) ==
  symbolIfCan(kx := ksec(k, l, x)) case SY => palgSolve(op, g, kx, k, x)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    z := rootOf(rec.prim)
    lk:List K := [kx, k]
    lv:List F := [(rec.pol1) z, (rec.pol2) z]
    (u := solve(localmap((f1:F):F +-> eval(f1, lk, lv), op),
      eval(g, lk, lv), x))
    case "failed" => "failed"
  rc := u::REC
  kz := retract(z)@K
  [eval(rc.particular, kz, rec.primelt),
   [eval(f, kz, rec.primelt) for f in rc.basis]]

lastChance(op, g, x)

doVarParams(eq, g, bas, x) ==
  (u := particularSolution(eq, g, bas, (f1:F):F +-> int(f1, x)))
  case "failed" => lastChance(eq, g, x)
  [u::F, bas]

lastChance(op, g, x) ==
  -- one? degree op => firstOrder(coefficient(op,0), leadingCoefficient op,g,x)
  (degree op) = 1 => firstOrder(coefficient(op,0), leadingCoefficient op,g,x)
  "failed"

-- solves a0 y + a1 y' = g
-- does not check whether there is a solution in the field generated by
-- a0, a1 and g
firstOrder(a0, a1, g, x) ==
  h := xpart(expint(- a0 / a1, x), x)
  [h * int((g / h) / a1, x), [h]]

-- xpart(f,x) removes any constant not involving x from f
xpart(f, x) ==
  l := reverse.! varselect(tower f, x)
lp := [k::P for k in l]
  smpxpart(numer f, x, l, lp) / smpxpart(denom f, x, l, lp)

upmp(p, 1) ==
  empty? 1 => p::P2
  up := univariate(p, k := first l)
  l := rest l
  ans:P2 := 0
  while up ^= 0 repeat
    ans := ans + monomial(upmp(leadingCoefficient up, l), k, degree up)
    up := reductum up
  ans

-- multint(a, [g1,...,gk], x) returns gk \int(g(k-1) \int(...g1 \int(a))...)
multint(a, l, x) ==
  for g in l repeat a := g * xpart(int(a, x), x)
  a

expso1s(op, k, x) ==
  one? degree op =>
  (degree op) = 1 =>
    firstOrder(multivariate(coefficient(op, 0), k),
              multivariate(leadingCoefficient op, k), 0, x).basis
    [xpart(expint(multivariate(h, k), x), x) for h in ricDsolve(op, ffactor)]

-- Finds solutions with rational logarithmic derivative
ratlogsol(oper, sols, k, x) ==
  bas := [xpart(multivariate(h, k), x) for h in sols]
  degree(oper) = #bas => bas -- all solutions are found already
  rec := ReduceOrder(oper, sols)
  le := expso1s(rec.eq, k, x)
  int:List(F) := [xpart(multivariate(h, k), x) for h in rec.op]
  concat_!(sols, [multint(e, int, x) for e in le])

homosolve1(oper, sols, k, x) ==
  zero?(n := (degree(oper) - #sols)::N) => sols -- all solutions found
  rec := ReduceOrder(oper, sols)
  int:List(F) := [xpart(h, x) for h in rec.op]
  concat_!(sols, [multint(e, int, x) for e in norf1(rec.eq, k, x, n::N)])

-- if the coefficients are rational functions, then the equation does not
-- not have a proper 1st-order right factor over the rational functions
norf1(op, k, x, n) ==
  one? n => firstOrder(coefficient(op, 0), leadingCoefficient op,0,x).basis
  (n = 1) => firstOrder(coefficient(op, 0), leadingCoefficient op,0,x).basis
  (n > 2) =>
    symbolIfCan(kmax vark(coefficients op, x)) case SY =>
    eq := ulodo(op, k)
    n = 2 => kovode(eq, k, x)
eq := last factor1 eq  -- eq cannot have order 1
degree(eq) = 2 =>
  empty?(bas := kovode(eq, k, x)) => empty()
  homosolve1(op, bas, k, x)
empty()

kovode(op, k, x) ==
b := coefficient(op, 1)
a := coefficient(op, 2)
u := kovacic(coefficient(op, 0), b, a, ffactor)) case "failed" => empty()
p := map(z1+->multivariate(z1, k), u::UPUP)
ba := multivariate(- b / a, k)
-- if p has degree 2 (case 2), then it must be squarefree since the
-- ode is irreducible over the rational functions, so the 2 roots of p
-- are distinct and must yield 2 independent solutions.
degree(p) = 2 => [xpart(expint(ba/(2::F) + e, x), x) for e in zerosOf p]
-- otherwise take 1 root of p and find the 2nd solution by reduction of order
y1 := xpart(expint(ba / (2::F) + zeroOf p, x), x)
y1, y1 * xpart(int(expint(ba, x) / y1**2, x), x)

solve(op:L, g:F, x:SY) ==
empty?(l := vark(coefficients op, x)) => constDsolve(op, g, x)
symbolIfCan(k := kmax l) case SY => rfSolve(op, g, k, x)
has?(operator k, ALGOP) => algSolve(op, g, k, l, x)
lastChance(op, g, x)

ulodo(eq, k) ==
  op:LQ := 0
  while eq ^= 0 repeat
    op := op + monomial(univariate(leadingCoefficient eq, k), degree eq)
    eq := reductum eq
  op

-- left hand side has rational coefficients
rfSolve(eq, g, k, x) ==
op := ulodo(eq, k)
empty? remove_!(k, varselect(kernels g, x)) =>  -- i.e. rhs is rational
  rc := ratDsolve(op, univariate(eq, k))
  rc.particular case "failed" =>  -- this implies g ^= 0
    doVarParams(eq, g, homosolve(eq, op, rc.basis, k, x), x)
    [multivariate(rc.particular::RF, k), homosolve(eq, op, rc.basis, k, x)]
    doVarParams(eq, g, homosolve(eq, op, ratDsolve(op, 0).basis, k, x), x)
solve(op, g, x, a, y0) ==
  (u := solve(op, g, x)) case "failed" => "failed"
hp := h := (u::REC).particular
  b := (u::REC).basis
v:V := new(n := #y0, 0)
  kx:K := kernel x
for i in minIndex v .. maxIndex v for yy in y0 repeat
  v.i := yy - eval(h, kx, a)
  h := diff h
  (sol := particularSolution(
    map_!((f1:F):F+->eval(f1,kx,a),wronskianMatrix(b,n)), v))
  case "failed" => "failed"
for f in b for i in minIndex(s := sol::V) .. repeat
  hp := hp + s.i * f
  hp

localmap(f, op) ==
  ans:L := 0
  while op ^= 0 repeat
    ans := ans + monomial(f leadingCoefficient op, degree op)
    op := reductum op
  ans

-- left hand side has pure algebraic coefficients
palgSolve(op, g, kx, k, x) ==
  rec := palgLODE(op, g, kx, k, x) -- finds solutions in the coef. field
  rec.particular case "failed" =>
  doVarParams(op, g, homosolve1(op, rec.basis, k, x), x)
  [(rec.particular)::F, homosolve1(op, rec.basis, k, x)]
ElementaryFunctionODESolver provides the top-level functions for finding closed form solutions of ordinary differential equations and initial value problems.

See Also:

- )show ElementaryFunctionODESolver
ElementaryFunctionODESolver (ODEEF)

Exports:
solve

— package ODEEF ElementaryFunctionODESolver —

\text{ElementaryFunctionODESolver}(R, F): \text{Exports == Implementation where}
R: \text{Join(OrderedSet, EuclideanDomain, RetractableTo Integer,}
\text{LinearlyExplicitRingOver Integer, CharacteristicZero)}
F: \text{Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,}
\text{PrimitiveFunctionCategory)}

N \Rightarrow \text{NonNegativeInteger}
OP \Rightarrow \text{BasicOperator}
SY \Rightarrow \text{Symbol}
K \Rightarrow \text{Kernel F}
EQ \Rightarrow \text{Equation F}
V \Rightarrow \text{Vector F}
M \Rightarrow \text{Matrix F}
UP \Rightarrow \text{SparseUnivariatePolynomial F}
P \Rightarrow \text{SparseMultivariatePolynomial}(R, K)
LEQ \Rightarrow \text{Record(left:UP, right:F)}
NLQ \Rightarrow \text{Record(dx:F, dy:F)}
REC \Rightarrow \text{Record(particular: F, basis: List F)}
VEC \Rightarrow \text{Record(particular: V, basis: List V)}
ROW \Rightarrow \text{Record(index: Integer, row: V, rh: F)}
SYS \Rightarrow \text{Record(mat:M, vec: V)}
CHAPTER 6. CHAPTER E

U ==> Union(REC, F, "failed")
UU ==> Union(F, "failed")
OPDIFF ==> "%diff":::SY

Exports ==> with
solve: (M, V, SY) -> Union(VEC, "failed")
  ++ solve(m, v, x) returns \begin{verbatim}[v_p, \begin{verbatim}[v_1,...,v_m]\end{verbatim}]\end{verbatim} such that
  ++ the solutions of the system \begin{verbatim}D y = m y + v\end{verbatim} are
  ++ constants, and the \begin{verbatim}[v_i's]\end{verbatim} form a basis for the solutions of
  ++ \begin{verbatim}D y = m y\end{verbatim}.
  ++ \begin{verbatim}v_p\end{verbatim} is the dependent variable.
solve: (M, SY) -> Union(List V, "failed")
  ++ solve(m, x) returns a basis for the solutions of \begin{verbatim}D y = m y\end{verbatim}.
  ++ \begin{verbatim}v_p\end{verbatim} is the dependent variable.

solve: (List EQ, List OP, SY) -> Union(VEC, "failed")
  ++ solve([eq_1,...,eq_n], [y_1,...,y_n], x) returns either "failed"
  ++ or, if the equations form a first order linear system, a solution
  ++ of the form \begin{verbatim}[h_p, \begin{verbatim}[b_1,...,b_n]\end{verbatim}]\end{verbatim} where \begin{verbatim}h_p\end{verbatim} is a
  ++ particular solution and \begin{verbatim}[b_1,...b_m]\end{verbatim} are linearly independent
  ++ solutions of the associated homogeneous system.
  ++ error if the equations do not form a first order linear system
solve: (List F, List OP, SY) -> Union(VEC, "failed")
  ++ solve([eq_1,...,eq_n], [y_1,...,y_n], x) returns either "failed"
  ++ or, if the equations form a first order linear system, a solution
  ++ of the form \begin{verbatim}[h_p, \begin{verbatim}[b_1,...,b_n]\end{verbatim}]\end{verbatim} where \begin{verbatim}h_p\end{verbatim} is a
  ++ particular solution and \begin{verbatim}[b_1,...b_m]\end{verbatim} are linearly independent
  ++ solutions of the associated homogeneous system.
  ++ error if the equations do not form a first order linear system

solve: (EQ, OP, SY) -> U
  ++ solve(eq, y, x) returns either a solution of the ordinary differential
  ++ equation \begin{verbatim}eq\end{verbatim} or "failed" if no non-trivial solution can be found;
  ++ If the equation is linear ordinary, a solution is of the form
  ++ \begin{verbatim}h, \begin{verbatim}[b_1,...,b_m]\end{verbatim}\end{verbatim} where \begin{verbatim}h\end{verbatim} is a particular solution
  ++ and \begin{verbatim}[b_1,...b_m]\end{verbatim} are linearly independent solutions of the
  ++ associated homogeneous equation \begin{verbatim}f(x,y) = 0\end{verbatim};
  ++ A full basis for the solutions of the homogeneous equation
  ++ is not always returned, only the solutions which were found;
  ++ If the equation is of the form \begin{verbatim}dy/dx = f(x,y)\end{verbatim}, a solution is of
  ++ the form \begin{verbatim}h(x,y)\end{verbatim} where \begin{verbatim}h(x,y) = c\end{verbatim} is a first integral
  ++ of the equation for any constant \begin{verbatim}c\end{verbatim};
  ++ error if the equation is not one of those 2 forms;
solve: (F, OP, SY) -> U
  ++ solve(eq, y, x) returns either a solution of the ordinary differential
  ++ equation \begin{verbatim}eq\end{verbatim} or "failed" if no non-trivial solution can be found;
  ++ If the equation is linear ordinary, a solution is of the form
  ++ \begin{verbatim}h, \begin{verbatim}[b_1,...,b_m]\end{verbatim}\end{verbatim} where \begin{verbatim}h\end{verbatim} is a particular solution and
  ++ and \begin{verbatim}[b_1,...b_m]\end{verbatim} are linearly independent solutions of the
  ++ associated homogeneous equation \begin{verbatim}f(x,y) = 0\end{verbatim};
  ++ A full basis for the solutions of the homogeneous equation
++ is not always returned, only the solutions which were found;
++ If the equation is of the form \(dy/dx = f(x,y)\), a solution is of
++ the form \(h(x,y)\) where \(h(x,y) = c\) is a first integral
++ of the equation for any constant \(c\);

solve: (EQ, OP, EQ, List F) -> UU
++ solve(eq, y, x = a, \([y0,...,ym]\)) returns either the solution
++ of the initial value problem \(eq, y(a) = y0, y'(a) = y1,\ldots\)
++ or "failed" if the solution cannot be found;
++ error if the equation is not one linear ordinary or of the form
++ \(dy/dx = f(x,y)\);

solve: (F, OP, EQ, List F) -> UU
++ solve(eq, y, x = a, \([y0,...,ym]\)) returns either the solution
++ of the initial value problem \(eq, y(a) = y0, y'(a) = y1,\ldots\)
++ or "failed" if the solution cannot be found;
++ error if the equation is not one linear ordinary or of the form
++ \(dy/dx = f(x,y)\);

Implementation ==> add
import ODEIntegration(R, F)
import IntegrationTools(R, F)
import NonLinearFirstOrderODESolver(R, F)

getfreelincoeff : (F, K, SY) -> F
getfreelincoeff1: (F, K, List F) -> F
getlincoeff : (F, K) -> F
getcoeff : (F, K) -> UU
parseODE : (F, OP, SY) -> Union(LEQ, NLQ)
parseLODE : (F, List K, UP, SY) -> LEQ
parseSYS : (List F, List OP, SY) -> Union(SYS, "failed")
parseSYSeq : (F, List K, List K, List F, SY) -> Union(ROW, "failed")

solve(diffeq:EQ, y:OP, x:SY) == solve(lhs diffeq - rhs diffeq, y, x)
solve(leq: List EQ, lop: List OP, x:SY) ==
++ solve([lhs eq - rhs eq for eq in leq], lop, x)
solve(diffeq:EQ, y:OP, center:EQ, y0:List F) ==
++ solve(lhs diffeq - rhs diffeq, y, center, y0)
solve(m:M, x:SY) ==
++ (u := solve(m, new(nrows m, 0), x)) case "failed" => "failed"
++ u.basis

solve(m:M, v:V, x:SY) ==
++ Lx := LinearOrdinaryDifferentialOperator(F, diff x)
uu := solve(m, v, (z1,z2) -> solve(z1, z2, x)_
++ $ElementaryFunctionLODESolver(R, F, Lx)$SystemODESolver(F, Lx)
++ uu case "failed" => "failed"
++ rec := uu::Record(particular: V, basis: M)
++ [rec.particular, [column(rec.basis, i) for i in 1..ncols(rec.basis)]]
solve(diffeq:F, y:OP, center:EQ, y0:List F) ==
a := rhs center
kx:K := kernel(x := retract(lhs(center))@SY)
(ur := parseODE(diffeq, y, x)) case NLQ =>
   -- not one?(#y0) => error "solve: more than one initial condition!"
   not ((#y0) = 1) => error "solve: more than one initial condition!"
   rc := ur::NLQ
   (u := solve(rc.dx, rc.dy, y, x)) case "failed" => "failed"
   u::F - eval(u::F, [kx, retract(y(x::F))@K], [a, first y0])
rec := ur::LEQ
p := rec.left
Lx := LinearOrdinaryDifferentialOperator(F, diff x)
op:Lx := 0
while p ^= 0 repeat
   op := op + monomial(leadingCoefficient p, degree p)
p := reductum p
solve(op, rec.right, x, a, y0)$ElementaryFunctionLODESolver(R, F, Lx)

solve(leq: List F, lop: List OP, x:SY) ==
   (u := parseSYS(leq, lop, x)) case SYS =>
      rec := u::SYS
      solve(rec.mat, rec.vec, x)
      error "solve: not a first order linear system"

solve(diffeq:F, y:OP, x:SY) ==
   (u := parseODE(diffeq, y, x)) case NLQ =>
      rc := u::NLQ
      (uu := solve(rc.dx, rc.dy, y, x)) case "failed" => "failed"
      uu::F
rec := u::LEQ
p := rec.left
Lx := LinearOrdinaryDifferentialOperator(F, diff x)
op:Lx := 0
while p ^= 0 repeat
   op := op + monomial(leadingCoefficient p, degree p)
p := reductum p
   (uuu := solve(op, rec.right, x)$ElementaryFunctionLODESolver(R, F, Lx))
   case "failed" => "failed"
   uuu::REC

-- returns \([M, v]\) s.t. the equations are \(D x = M x + v\)
parseSYS(eqs, ly, x) ==
   (n := #eqs) ^= #ly => "failed"
   m:M := new(n, n, 0)
v:V := new(n, 0)
xx := x::F
if := [y xx for y in ly]
lk0:List(K) := [retract(f)@K for f in if]
lk1:List(K) := [retract(differentiate(f, x))@K for f in if]
for eq in eqs repeat
  (u := parseSYSeq(eq, 1k0, 1k1, 1lf, x)) case "failed" => return "failed"
  rec := u::ROW
  setRow_!(m, rec.index, rec.row)
  v(rec.index) := rec.rh
  [m, v]
parseSYSeq(eq, 10, 11, 1lf, x) ==
  l := [k for k in varselect(kernels eq, x) | is?(k, OPDIFF)]
  empty? l or not empty? rest l or zero?(n := position(k := first l, l1)) =>
    "failed"
  c := getfreelincoeff1(eq, k, lf)
  eq := eq - c * k::F
  v:V := new(#l0, 0)
  for y in l0 for i in 1.. repeat
    ci := getfreelincoeff1(eq, y, lf)
    v.i := - ci / c
    eq := eq - ci * y::F
  [n, v, -eq]

-- returns either [p, g] where the equation (diffeq) is of the form p(D)(y) = g
-- or [p, q] such that the equation (diffeq) is of the form p dx + q dy = 0
parseODE(diffeq, y, x) ==
  f := y(x::F)
  l:List(K) := [retract(f)@K]
  n:N := 2
  for k in varselect(kernels diffeq, x) | is?(k, OPDIFF) repeat
    if (m := height k) > n then n := m
  n := (n - 2)::N
-- build a list of kernels in the order [y^(n)(x),...,y''(x),y'(x),y(x)]
  for i in 1..n repeat
    l := concat(retract(f := differentiate(f, x))@K, l)
  k:K -- #$^#& compiler requires this line and the next one too...
  c:F
  while not(empty? l) and zero?(c := getlincoeff(diffeq, k := first l))
    repeat l := rest l
  empty? l or empty? rest l => error "parseODE: equation has order 0"
  diffeq := diffeq - c * (k::F)
  ny := name y
  l := rest l
  height(k) > 3 => parseLODE(diffeq, l, monomial(c, #l), ny)
  (u := getcoeff(diffeq, k := first l)) case "failed" => [diffeq, c]
  eqrhs := (d := u::F) * (k::F) - diffeq
  freeOf?(eqrhs, ny) and freeOf?(c, ny) and freeOf?(d, ny) =>
    [monomial(c, 1) + d::UP, eqrhs]
    [diffeq, c]
-- returns [p, g] where the equation (diffeq) is of the form p(D)(y) = g
parseLODE(diffeq, l, p, y) ==
  not freeOf?(leadingCoefficient p, y) =>
error "parseLODE: not a linear ordinary differential equation"

d := degree(p)::Integer - 1
for k in l repeat
  p := p + monomial(c := getfreelincoeff(diffeq, k, y), d::N)
  d := d - 1
  diffeq := diffeq - c * (k::F)
free0f?(diffeq, y) => [p, - diffeq]
error "parseLODE: not a linear ordinary differential equation"

getfreelincoeff(f, k, y) ==
  free0f?(c := getlincoeff(f, k), y) => c
error "getfreelincoeff: not a linear ordinary differential equation"

getfreelincoeff1(f, k, ly) ==
  c := getlincoeff(f, k)
  for y in ly repeat
    not free0f?(c, y) =>
      error "getfreelincoeff: not a linear ordinary differential equation"
    c

getlincoeff(f, k) ==
  (u := getcoeff(f, k)) case "failed" =>
    error "getlincoeff: not an appropriate ordinary differential equation"
  u::F

getcoeff(f, k) ==
  (r := retractIfCan(univariate(denom f, k))@Union(P, "failed"))
  case "failed" or degree(p := univariate(numer f, k)) > 1 => "failed"
  coefficient(p, 1) / (r::P)

package SIGNEF ElementaryFunctionSign

package SIGNEF ElementaryFunctionSign
This package provides functions to determine the sign of an elementary function around a point or infinity.

See Also:
- \( \text{show ElementaryFunctionSign} \)
ElementaryFunctionSign (SIGNEF)

Exports:

sign

— package SIGNEF ElementaryFunctionSign —

)abbrev package SIGNEF ElementaryFunctionSign
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1989
++ Date Last Updated: 4 May 1992
++ Description:
++ This package provides functions to determine the sign of an
++ elementary function around a point or infinity.

ElementaryFunctionSign(R,F): Exports == Implementation where
R : Join(IntegralDomain,OrderedSet,RetractableTo Integer,_,
         LinearlyExplicitRingOver Integer,GcdDomain)
F : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_,
         FunctionSpace R)

N ==> NonNegativeInteger
Z ==> Integer
SY ==> Symbol
RF ==> Fraction Polynomial R
ORF ==> OrderedCompletion RF
OFE ==> OrderedCompletion F
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
U ==> Union(Z, "failed")
FS2 ==> FunctionSpaceFunctions2
POSIT ==> "positive"
NEGAT ==> "negative"

Exports ==> with
sign: F -> U
++ sign(f) returns the sign of f if it is constant everywhere.
sign: (F, SY, OFE) -> U
++ sign(f, x, a) returns the sign of f as x nears \spad{a}, from both
++ sides if \spad{a} is finite.

Implementation ==> add
import ToolsForSign R
import RationalFunctionSign(R)
import PowerSeriesLimitPackage(R, F)
import TrigonometricManipulations(R, F)

smpsign : P -> U
sqfrSign: P -> U
termSign: P -> U
kerSign : K -> U
listSign: (List P, Z) -> U
insign : (F, SY, OFE, N) -> U
psign : (F, SY, OF, String, N) -> U
ofesign : OFE -> U
overRF : OFE -> Union(ORF, "failed")

sign(f, x, a) ==
not real? f => "failed"
insign(f, x, a, 0)

sign(f, x, a, st) ==
not real? f => "failed"
pssign(f, x, a, st, 0)

sign f ==
not real? f => "failed"
(u := retractIfCan(f)@Union(RF, "failed")) case RF => sign(u::RF)
(un := smpsign numer f) case Z and (ud := smpsign denom f) case Z =>
  un::Z * ud::Z
-- abort if there are any variables
not empty? variables f => "failed"
-- abort in the presence of algebraic numbers
member?(coerce("rootOf")::Symbol,
          map(name,operators f)$ListFunctions2(BasicOperator,Symbol)) => "failed"
-- In the last resort try interval evaluation where feasible.
if R has ConvertibleTo Float then
  import Interval(Float)
  import Expression(Interval Float)
  mapfun : (R -> Interval(Float)) := z -> interval(convert(z)$R)
f2 : Expression(Interval Float) :=
  map(mapfun,f)$FS2(R,F,Interval(Float),Expression(Interval Float))
r : Union(Interval(Float), "failed") := retractIfCan f2
if r case "failed" then return "failed"
negative? r => return(-1)
positive? r => return 1
zero? r => return 0
"failed"
"failed"

overRF a ==
(n := whatInfinity a) = 0 =>
(u := retractIfCan(retract(a)@F)@Union(RF,"failed")) _
case "failed" => "failed"
u::RF::ORF
n * plusInfinity()$ORF

ofesign a ==
(n := whatInfinity a) ^= 0 => convert(n)@Z
sign(retract(a)@F)

insign(f, x, a, m) ==
m > 10 => "failed"  -- avoid infinite loops for now
(uf := retractIfCan(f)@Union(RF,"failed")) case RF and
   (ua := overRF a) case ORF => sign(uf::RF, x, ua::ORF)
eq : Equation OFE := equation(x :: F :: OFE,a)
(u := limit(f,eq)) case "failed" => "failed"
u case OFE =>
(n := whatInfinity(u::OFE)) ^= 0 => convert(n)@Z
(v := retract(u::OFE)@F) = 0 =>
(s := insign(differentiate(f, x), x, a, m + 1)) case "failed"
   => "failed"
- s::Z * n
sign v
(u.leftHandLimit case "failed") or
   (u.rightHandLimit case "failed") => "failed"
(ul := ofesign(u.leftHandLimit::OFE)) case "failed" => "failed"
(ur := ofesign(u.rightHandLimit::OFE)) case "failed" => "failed"
(ul::Z) = (ur::Z) => ul
"failed"

psign(f, x, a, st, m) ==
m > 10 => "failed"  -- avoid infinite loops for now
f = 0 => 0
(uf := retractIfCan(f)@Union(RF,"failed")) case RF and
   (ua := retractIfCan(a)@Union(RF,"failed")) case RF =>
      sign(uf::RF, x, ua::RF, st)
eq : Equation F := equation(x :: F,a)
(u := limit(f,eq,st)) case "failed" => "failed"
u case OFE =>
(n := whatInfinity(u::OFE)) ^= 0 => convert(n)@Z
(v := retract(u::OFE)@F) = 0 =>
(s := psign(differentiate(f,x),x,a,st,m + 1)) case "failed"=>
"failed"
direction(st) * s::Z

sign v

smpSign p ==
  (r := retractIfCan(p) Union(R,"failed")) case R => sign(r::R)
  (u := sign(retract(unit(s := squareFree p)@R)) case "failed" =>
    "failed"
  ans := u::Z
  for term in factorList s | odd?(term.xpnt) repeat
    (u := sqfrSign(term.fctr)) case "failed" => return "failed"
  ans := ans * u::Z
  ans

sqfrSign p ==
  (u := termSign first(l := monomials p)) case "failed" => "failed"
  listSign(rest l, u::Z)

listSign(l, s) ==
  for term in l repeat
    (u := termSign term) case "failed" => return "failed"
    not(s = u::Z) => return "failed"
  s

termSign term ==
  (us := sign leadingCoefficient term) case "failed" => "failed"
  for var in (lv := variables term) repeat
    odd? degree(term, var) =>
      empty? rest lv and (vs := kerSign first lv) case Z =>
        return(us::Z * vs::Z)
    return "failed"

us::Z

kerSign k ==
  has?(op := operator k, "NEGAT") => -1
  has?(op, "POSIT") or is?(op,"pi"::SY) or is?(op,"exp"::SY) or
    is?(op,"cosh"::SY) or is?(op,"sech"::SY) => 1
  empty?(arg := argument k) => "failed"
  (s := sign first arg) case "failed" =>
    is?(op,"nthRoot" :: SY) =>
      even?(retract(second arg)@Z) => 1
    "failed"
  "failed"
  is?(op,"log" :: SY) =>
    s::Z < 0 => "failed"
    sign(first arg - 1)
  is?(op,"tanh" :: SY) or is?(op,"sinh" :: SY) or
    is?(op,"csch" :: SY) or is?(op,"coth" :: SY) => s
  is?(op,"nthRoot" :: SY) =>
    even?(retract(second arg)@Z) =>
    s::Z < 0 => "failed"
package EFSTRUC ElementaryFunctionStructurePackage

--- ElementaryFunctionStructurePackage.input ---

)set break resume
)sys rm -f ElementaryFunctionStructurePackage.output
)spool ElementaryFunctionStructurePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryFunctionStructurePackage
--R
--R ElementaryFunctionStructurePackage(R: Join(IntegralDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),F: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(R))) is a package constructor
--R Abbreviation for ElementaryFunctionStructurePackage is EFSTRUC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EFSTRUC
--R
--R----------------------------- Operations ---------------------------------
--R normalize : F -> F normalize : (F,Symbol) -> F
--R realElementary : F -> F realElementary : (F,Symbol) -> F
--R rootNormalize : (F,Kernel(F)) -> F tanQ : (Fraction(Integer),F) -> F
--R rischNormalize : (F,Symbol) -> Record(func: F,kers: List(Kernel(F)),vals: List(F))
--R validExponential : (List(Kernel(F)),F,Symbol) -> Union(F,"failed")
--R
--E 1

)spool
)lisp (bye)
ElementaryFunctionStructurePackage provides functions to test the algebraic independence of various elementary functions, using the Risch structure theorem (real and complex versions). It also provides transformations on elementary functions which are not considered simplifications.

See Also:
o )show ElementaryFunctionStructurePackage
 CHAPTER 6.  CHAPTER E

++ Risch structure theorem (real and complex versions).
++ It also provides transformations on elementary functions
++ which are not considered simplifications.

ElementaryFunctionStructurePackage(R,F): Exports == Implementation where
R : Join(IntegralDomain, OrderedSet, RetractableTo Integer,
        LinearlyExplicitRingOver Integer)
F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
        FunctionSpace R)

B ==> Boolean
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Z
SY ==> Symbol
K ==> Kernel F
UP ==> SparseUnivariatePolynomial F
SMP ==> SparseMultivariatePolynomial(R, K)
REC ==> Record(func:F, kers: List K, vals:List F)
U ==> Union(vec:Vector Q, func:F, fail: Boolean)
POWER ==> "^"::SY
NTHR ==> "nthRoot":SY

Exports ==> with
  normalize: F -> F
    ++ normalize(f) rewrites \spad{f} using the least possible number of
    ++ real algebraically independent kernels.
  normalize: (F, SY) -> F
    ++ normalize(f, x) rewrites \spad{f} using the least possible number of
    ++ real algebraically independent kernels involving \spad{x}.
  rischNormalize: (F, SY) -> REC
    ++ rischNormalize(f, x) returns \spad{[g, [k1,...,kn], [h1,...,hn]]}
    ++ such that \spad{g = normalize(f, x)} and each \spad{ki} was
    ++ rewritten as \spad{hi} during the normalization.
  realElementary: F -> F
    ++ realElementary(f) rewrites \spad{f} in terms of the 4 fundamental real
    ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
  realElementary: (F, SY) -> F
    ++ realElementary(f,x) rewrites the kernels of \spad{f} involving
    ++ \spad{x} in terms of the 4 fundamental real
    ++ transcendental elementary functions: \spad{log, exp, tan, atan}.
  validExponential: (List K, F, SY) -> Union(F, "failed")
    ++ validExponential([k1,...,kn],f,x) returns \spad{g} if \spad{exp(f)=g}
    ++ and \spad{g} involves only \spad{k1...kn}, and "failed" otherwise.
  rootNormalize: (F, K) -> F
    ++ rootNormalize(f, k) returns \spad{f} rewriting either \spad{k} which
    ++ must be an nth-root in terms of radicals already in \spad{f}, or some
    ++ radicals in \spad{f} in terms of \spad{k}.
  tanQ: (Q, F) -> F
    ++ tanQ(q,a) is a local function with a conditional implementation.
Implementation ==> add
import TangentExpansions F
import IntegrationTools(R, F)
import IntegerLinearDependence F
import AlgebraicManipulations(R, F)
import InnerCommonDenominator(Z, Q, Vector Z, Vector Q)

k2Elem : (K, List SY) -> F
realElem : (F, List SY) -> F
smpElem : (SMP, List SY) -> F
deprel : (List K, K, SY) -> U
rootDep : (List K, K) -> U
qdeprel : (List F, F) -> U
factdeprel : (List K, K) -> U
toR : (List K, F) -> List K
toY : List K -> List F
toZ : List K -> List F
toU : List K -> List F
toV : List K -> List F
ktoY : K -> F
ktoZ : K -> F
ktoU : K -> F
ktoV : K -> F
gdCoef? : (Q, Vector Q) -> Boolean
goodCoef : (Vector Q, List K, SY) ->
          Union(Record(index:Z, ker:K), "failed")
tanRN : (Q, K) -> F
localnorm : F -> F
rooteval : (F, List K, K, Q) -> REC
logeval : (F, List K, K, Vector Q) -> REC
expEval : (F, List K, K, Vector Q) -> REC
taneval : (F, List K, K, Vector Q) -> REC
ataneval : (F, List K, K, Vector Q) -> REC
depEval : (F, List K, K, Vector Q) -> REC
expnose:mp : (F, List K, K, Vector Q, List F, F) -> REC
tannose:mp : (F, List K, K, Vector Q, List F, F) -> REC
rtNormalize : F -> F
rootNormalize0 : F -> REC
rootKernelNormalize : (F, List K, K) -> Union(REC, "failed")
tanSum : (F, List F) -> F

comb? := F has CombinatorialOpsCategory
mpiover2:F := pi()$F / (-2::F)
realElem(f, 1) == smpElem(numer f, 1) / smpElem(denom f, 1)
realElementary(f, x) == realElem(f, [x])
realElementary f == realElem(f, variables f)
toY ker == [func for k in ker | (func := ktoY k) ^= 0]
toZ ker == [func for k in ker | (func := ktoZ k) ^= 0]
toU ker == [func for k in ker | (func := ktoU k) ^= 0]
toV ker == [func for k in ker | (func := ktoV k) ^= 0]
rtNormalize f == rootNormalize0(f).func
toR(ker, x) == select(s+->is?(s, NTHR) and first argument(s) = x, ker)

if R has GcdDomain then
  tanQ(c, x) ==
    tanNa(rootSimp zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)
else
  tanQ(c, x) ==
    tanNa(zeroOf tanAn(x, denom(c)::PositiveInteger), numer c)

-- tanSum(c, [a1,...,an]) returns f(c, a1,...,an) such that
-- if ai = tan(ui) then f(c, a1,...,an) = tan(c + u1 + ... + un).
-- MUST BE CAREFUL FOR WHEN c IS AN ODD MULTIPLE of pi/2
tanSum(c, l) ==
  k := c / mpiover2 -- k = - 2 c / pi, check for odd integer
  (r := retractIfCan(k)@Union(Z, "failed")) case Z and odd?(r::Z) =>
    - inv tanSum l
  tanSum concat(tan c, l)

rootNormalize0 f ==
k er := select_!(s+->is?(s, NTHR) and empty? variables first argument s,
  tower f)$List(K)
empty? ker => [f, empty(), empty()]
(n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
for i in 1..n for kk in rest ker repeat
  (u := rootKernelNormalize(f, first(ker, i), kk)) case REC =>
    rec := u::REC
    rn := rootNormalize0(rec.func)
    return [rn.func, concat(rec.kers,rn.kers), concat(rec.vals, rn.vals)]
  [f, empty(), empty()]

depr el(ker, k, x) ==
  is?(k, "log"::SY) or is?(k, "exp"::SY) =>
    qdeprel([[differentiate(g, x) for g in toY ker],
              differentiate(ktoY k, x))
  is?(k, "atan"::SY) or is?(k, "tan"::SY) =>
    qdeprel([[differentiate(g, x) for g in toU ker],
              differentiate(ktoU k, x))
  is?(k, NTHR) => rootDep(ker, k)
  comb? and is?(k, "factorial"::SY) =>
    factdeprel([x for x in ker | is?(x,"factorial"::SY) and x^=k],k)
    [true]

ktoY k ==
  is?(k, "log"::SY) => k::F
  is?(k, "exp"::SY) => first argument k
  0
ktoZ k ==
  is?(k, "log"::SY) => first argument k
  is?(k, "exp"::SY) => k::F
  0

ktoU k ==
  is?(k, "atan"::SY) => k::F
  is?(k, "tan"::SY) => first argument k
  0

ktoV k ==
  is?(k, "atan"::SY) => k::F
  is?(k, "tan"::SY) => first argument k
  0

smpElem(p, l) ==
  map(x+->k2Elem(x, l), y+->y::F, p)
  $PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, F)

k2Elem(k, l) ==
  ez, iez, tz2: F
  kf := k::F
  not(empty? l) and empty? [v for v in variables kf | member?(v, l)] => kf
  empty?(args :List F := [realElem(a, l) for a in argument k]) => kf
  z := first args
  is?(k, POWER) => (zero? z => 0; exp(last(args) * log z))
  is?(k, "cot"::SY) => inv tan z
  is?(k, "acot"::SY) => atan inv z
  is?(k, "asin"::SY) => atan(z / sqrt(1 - z**2))
  is?(k, "acos"::SY) => atan(sqrt(1 - z**2) / z)
  is?(k, "asec"::SY) => atan sqrt(1 - z**2)
  is?(k, "acsc"::SY) => atan inv sqrt(1 - z**2)
  is?(k, "asinh"::SY) => log(sqrt(1 + z**2) + z)
  is?(k, "acosh"::SY) => log(sqrt(z**2 - 1) + z)
  is?(k, "atanh"::SY) => log((z + 1) / (1 - z)) / (2::F)
  if has?(op := operator k, "htrig") then iez := inv(ez := exp z)
  if has?(op, "trig") then tz2 := tan(z / (2::F))
is?(k, "sin"::SY) => 2 * tz2 / (1 + tz2**2)
is?(k, "cos"::SY) => (1 - tz2**2) / (1 + tz2**2)
is?(k, "sec"::SY) => (1 + tz2**2) / (1 - tz2**2)
is?(k, "csc"::SY) => (1 + tz2**2) / (2 * tz2)

op args

--The next 5 functions are used by normalize, once a relation is found
depeval(f, lk, k, v) ==
is?(k, "log"::SY) => logeval(f, lk, k, v)
is?(k, "exp"::SY) => expeval(f, lk, k, v)
is?(k, "tan"::SY) => taneval(f, lk, k, v)
is?(k, "atan"::SY) => ataneval(f, lk, k, v)
is?(k, NTHR) => rooteval(f, lk, k, v(minIndex v))
[f, empty(), empty()]

rooteval(f, lk, k, n) ==
  nv := nthRoot(x := first argument k, m := retract(n)@Z)
  l := [r for r in concat(k, toR(lk, x)) |
    retract(second argument r)@Z ^= m]
  lv := [nv ** (n / (retract(second argument r)@Z::Q)) for r in l]
  [eval(f, l, lv), l, lv]

ataneval(f, lk, k, v) ==
  w := first argument k
  s := tanSum [tanQ(qelt(v,i), x) for i in minIndex v .. maxIndex v for x in toV lk]
  g := +[qelt(v,i) * x for i in minIndex v .. maxIndex v for x in toU lk]
  h:F :=
    zero?(d := 1 + s * w) => mpiover2
    atan((w - s) / d)
    g := g + h
  [eval(f, [k], [g]), [k], [g]]

gdCoef?(c, v) ==
  for i in minIndex v .. maxIndex v repeat
    retractIfCan(qelt(v, i) / c)@Union(Z, "failed") case "failed" =>
      return false
     true

goodCoef(v, l, s) ==
  for i in minIndex v .. maxIndex v for k in l repeat
    is?(k, s) and
    ((r:=recip(qelt(v,i))) case Q) and
    (retractIfCan(r::Q)@Union(Z, "failed") case Z)
    and gdCoef?(qelt(v, i), v) => return([i, k])
"failed"

taneval(f, lk, k, v) ==
  u := first argument k
  fns := toU lk
c := u - \sum_{i \in \text{minIndex} v .. \text{maxIndex} v \text{ for } x \in \text{fns}} [qelt(v, i) * x]

(rec := \text{goodCoeff}(v, lk, "tan"::SY)) \text{ case } "\text{failed}" =>
\text{tannosimp}(f, lk, k, v, fns, c)

v0 := \text{retract}(\text{inv qelt}(v, \text{rec.index}))@Z

l := \sum_{i \in \text{minIndex} v .. \text{maxIndex} v} [qelt(v, i) \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v]

\text{tannosimp}(f, lk, k, v, fns, c) ==
\text{every?(x+->\text{is?(x, "tan"::SY), lk}) =}
\text{dd := (d := (cd := \text{splitDenominator} v).\text{den})::F}
\text{newt := [\text{tan}(u / dd) \text{ for } u \in \text{fns}]@List(F)}
\text{newtan := [\text{tanNa}(t, d) \text{ for } t \in \text{newt}]@List(F)}
\text{h := tanSum(c, [\text{tanNa}(t, qelt(cd.num, i))}
     \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v \text{ for } t \in \text{newt})]

\text{lk := concat(k, lk)}
\text{newtan := concat(h, newtan)}
\text{[eval(f, lk, newtan), lk, newtan]}
\text{h := tanSum(c, [\text{tanQ}(qelt(v, i), x)}
     \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v \text{ for } x \in \text{toV} \text{ lk})]

\text{expnosimp}(f, lk, k, v, fns, g) ==
\text{every?(x+->\text{is?(x, "exp"::SY), lk}) =}
\text{dd := (d := (cd := \text{splitDenominator} v).\text{den})::F}
\text{newe := [\text{exp}(y / dd) \text{ for } y \in \text{fns}]@List(F)}
\text{newexp := [e ** d \text{ for } e \in \text{newe}]@List(F)}
\text{h := \prod\text{[e ** qelt(cd.num, i)}}
     \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v \text{ for } e \in \text{newe}] * g
\text{lk := concat(k, lk)}
\text{newexp := concat(h, newexp)}
\text{[eval(f, lk, newexp), lk, newexp]}
\text{h := \prod\text{[exp}(y) ** qelt(v, i)}
     \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v \text{ for } y \in \text{fns}] * g
\text{[eval(f, [k], [h]), [k], [h]]}

logeval(f, lk, k, v) ==
z := \text{first argument} k
\text{c := z / \prod[qelt(v, i)}
     \text{ for } x \in \text{toZ} \text{ lk} \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v])
\text{-- CHANGED log ktoZ x TO ktoY x SINCE WE WANT log exp f TO BE REPLACED BY f.}
g := \sum[qelt(v, i) * x]
     \text{ for } i \in \text{minIndex} v .. \text{maxIndex} v \text{ for } x \in \text{toY} \text{ lk}] + \log c
\text{[eval(f, [k], [g]), [k], [g]]}

\text{rischNormalize}(f, v) ==
\text{empty?(ker := \text{varselect(tower} f, v)) =}
\text{[f, empty(), empty()]}
first(ker) ^= kernel(v)@k => error "Cannot happen"
ker := rest ker
(n := (#ker)::Z - 1) < 1 => [f, empty(), empty()]
for i in 1..n for kk in rest ker repeat
  klist := first(ker, i)
  -- NO EVALUATION ON AN EMPTY VECTOR, WILL CAUSE INFINITE LOOP
  (c := deprel(klist, kk, v)) case vec and not empty?(c.vec) =>
    rec := depeval(f, klist, kk, c.vec)
    rn := rischNormalize(rec.func, v)
    return [rn.func,
    concat(rec.kers, rn.kers), concat(rec.vals, rn.vals)]
  c case func =>
    rn := rischNormalize(eval(f, [kk], [c.func]), v)
    return [rn.func, concat(kk, rn.kers), concat(c.func, rn.vals)]
[f, empty(), empty()]

rootNormalize(f, k) ==
  (u := rootKernelNormalize(f, toR(tower f, first argument k), k))
  case "failed" => f
  (u::REC).func

rootKernelNormalize(f, l, k) ==
  (c := rootDep(l, k)) case vec =>
    rooteval(f, l, k, (c.vec)(minIndex(c.vec)))
  "failed"

localnorm f ==
  for x in variables f repeat
    f := rischNormalize(f, x).func
  f

validExponential(twr, eta, x) ==
  (c := solveLinearlyOverQ(construct([differentiate(g, x)
    for g in (fns := toY twr)$List(F))@Vector(F),
    differentiate(eta, x))) case "failed" => "failed"
  v := c::Vector(Q)
  g := eta - +/[qelt(v, i) * yy
    for i in minIndex v .. maxIndex v for yy in fns]
  */[exp(yy) ** qelt(v, i)
    for i in minIndex v .. maxIndex v for yy in fns] * exp g

rootDep(ker, k) ==
  empty?(ker := toR(ker, first argument k)) => [true]
  [new(1,lcm(retract(second argument k)@Z,
    "lcm"/[retract(second argument r)@Z for r in ker])::Q)$Vector(Q)]

qdeprel(l, v) ==
  (u := solveLinearlyOverQ(construct(l)$Vector(F), v))
  case Vector(Q) => [u::Vector(Q)]
  [true]
expeval(f, lk, k, v) ==
y := first argument k
fns := toY lk
g := y - +/[qelt(v, i) * z for i in minIndex v .. maxIndex v for z in fns]
(rec := goodCoeff(v, lk, "exp":SY)) case "failed" =>
    expnosimp(f, lk, k, v, fns, expand g)
v0 := retract(inv qelt(v, rec.index))@Z
lv := [qelt(v, i) for i in minIndex v .. maxIndex v | i ^= rec.index]$List(Q)
l := [kk for kk in lk | kk ^= rec.ker]
h :F := */[exp(z) ** (- retract(a * v0)@Z) for a in lv for z in toY l]
h := h * exp(-v0 * g) * (k::F) ** v0
[eval(f, [rec.ker], [h]), [rec.ker], [h]]

if F has CombinatorialOpsCategory then
normalize f == rtNormalize localnorm factorials realElementary f
normalize(f, x) ==
    rtNormalize(rischNormalize(factorials(realElementary(f,x),x),x).func)

factdeprel(l, k) ==
    (r := retractIfCan(n := first argument k)@Union(Z, "failed"))
    case Z and (r::Z > 0) => [factorial(r::Z)::F]
    for x in l repeat
        m := first argument x
        ((r := retractIfCan(n - m)@Union(Z, "failed")) case Z) and
        (r::Z > 0) => return([*/[(m + i::F) for i in 1..r] * x::F])
    [true]

else
    normalize f == rtNormalize localnorm realElementary f
    normalize(f, x) == rtNormalize(rischNormalize(realElementary(f,x),x).func)
package INTEF ElementaryIntegration

--- ElementaryIntegration.input ---

)set break resume
)sys rm -f ElementaryIntegration.output
)spool ElementaryIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryIntegration
--R
--R ElementaryIntegration(R: Join(GcdDomain,OrderedSet,CharacteristicZero,RetractableTo(Integer)),F: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(R))) is a package constructor
--R Abbreviation for ElementaryIntegration is INTEF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTEF
--R
--R------------------------------- Operations --------------------------------
--R lfextendedint : (F,Symbol,F) -> Union(Record(ratpart: F,coeff: F),"failed")
--R lfextlimint : (F,Symbol,Kernel(F),List(Kernel(F))) -> Union(Record(ratpart: F,coeff: F),"failed")
--R lfinfieldint : (F,Symbol) -> Union(F,"failed")
--R lfintegrate : (F,Symbol) -> IntegrationResult(F)
--R lflimitedint : (F,Symbol,List(F)) -> Union(Record(mainpart: F,limitedlogs: List(Record(coeff: F,logand: F))),"failed")

--E 1

)spool
)lisp (bye)

--- ElementaryIntegration.help ---

====================================================================
ElementaryIntegration examples
====================================================================

This package provides functions for integration, limited integration, extended integration and the risch differential equation for elementary functions.

See Also:
  o )show ElementaryIntegration
ElementaryIntegration (INTEF)

Exports:
lfextendedint lfextlimint lfintfieldint lfintegrate lflimitedint

— package INTEF ElementaryIntegration —

)abbrev package INTEF ElementaryIntegration
++ Author: Manuel Bronstein
++ Date Created: 1 February 1988
++ Date Last Updated: 24 October 1995
++ Description:
++ This package provides functions for integration, limited integration,
++ extended integration and the risch differential equation for
++ elementary functions.

ElementaryIntegration(R, F): Exports == Implementation where
R : Join(GcdDomain, OrderedSet, CharacteristicZero,
          RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
          FunctionSpace R)
SE ==> Symbol
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
IR ==> IntegrationResult F
FF ==> Record(ratpart:RF, coeff:RF)
LLG ==> List Record(coeff:F, logand:F)
U2 ==> Union(Record(ratpart:F, coeff:F), "failed")
U3 ==> Union(Record(mainpart:F, limitedlogs:LLG), "failed")
ANS ==> Record(special:F, integrand:F)
PSOL ==> Record(ans:F, right:F, sol?:Boolean)
FAIL ==> error "failed - cannot handle that integrand"
ALGOP ==> "%alg"
OPDIFF ==> "%diff"::SE

Exports ==> with
lfextendedint: (F, SE, F) -> U2
  ++ lfextendedint(f, x, g) returns functions \spad{[h, c]} such that
  ++ \spad{dh/dx = f - cg}, if (h, c) exist, "failed" otherwise.
lflimitedint : (F, SE, List F) -> U3
  ++ lflimitedint(f,x,[g1,...,gn]) returns functions \spad{[h,[[ci, gi]]]} and
  ++ \spad{d(h+sum(ci log(gi)))/dx = f}, if possible, "failed" otherwise.
lfinfieldint : (F, SE) -> Union(F, "failed")
  ++ lfinfieldint(f, x) returns a function g such that \spad{dg/dx = f}
  ++ if g exists, "failed" otherwise.
lfintegrate : (F, SE) -> IR
  ++ lfintegrate(f, x) = g such that \spad{dg/dx = f}.
lfextlimint : (F, SE, K, List K) -> U2
  ++ lfextlimint(f,x,k,[k1,...,kn]) returns functions \spad{[h, c]}
  ++ such that \spad{dh/dx = f - c dk/dx}. Value h is looked for in a field
  ++ containing f and k1,...,kn (the ki's must be logs).

Implementation ==> add
import IntegrationTools(R, F)
import ElementaryRischDE(R, F)
import RationalIntegration(F, UP)
import AlgebraicIntegration(R, F)
import AlgebraicManipulations(R, F)
import ElementaryRischDESystem(R, F)
import TranscendentalIntegration(F, UP)
import PureAlgebraicIntegration(R, F, F)
import IntegrationResultFunctions2(F, F)
import IntegrationResultFunctions2(RF, F)
import FunctionSpacePrimitiveElement(R, F)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

algfint : (F, K, List K, SE) -> IR
algfextint : (F, K, List K, SE, F) -> U2
algflimint : (F, K, List K, SE, List F) -> U3
primextint : (F, SE, K) -> U2
expextint : (F, SE, K) -> U2
prilimint : (F, SE, K, List F) -> U3
exlimint : (F, SE, K, List F) -> U3
algprimint : (F, K, K, SE) -> IR
algexpint : (F, K, K, SE) -> IR
primint : (F, SE, K) -> IR
expint : (F, SE, K) -> IR
tanint : (F, SE, K) -> IR
prim? : (K, SE) -> Boolean
isx? : (F, SE) -> Boolean
addx : (IR, F) -> IR
cfind : (F, LLG) -> F
lfintegrate0: (F, SE) -> IR
unknownint : (F, SE) -> IR
unkextint : (F, SE, F) -> U2
unklimint : (F, SE, List F) -> U3
tryChangeVar: (F, K, SE) -> Union(IR, "failed")
droponex : (F, F, K, F) -> Union(F, "failed")

prim? (k, x) == is? (k, "log"::SE) or has? (operator k, "prim")

tanint(f, x, k) ==
eta' := differentiate(eta := first argument k, x)
r1 :=
tanintegrate(univariate(f, k),
(x1:UP):UP +-> differentiate(x1, x),
(x2:F):F +-> differentiate(x2, x),
monomial(eta', 2) + eta'::UP),
(x3:Integer,x4:F,x5:F):Union(List F,"failed") +->
rischDEsys(x3, 2 * eta, x4, x5, x),
(x6:F,x7:List F):U3 +-> lflimitedint(x6, x, x7),
(x8:F,x9:F):U2 +-> lfextendedint(x8, x, x9))
map((x1:RF):F +-> multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

-- tries various tricks since the integrand contains something not elementary
unknownint(f, x) ==
((r := retractIfCan(f)@Union(K, "failed")) case K) and
is?(k := r::K, OPDIFF) and
((ka:=retractIfCan(a:=second(l:=argument k))@Union(K,"failed"))case K)
and ((z := retractIfCan(zz := third l)@Union(SE, "failed")) case SE)
and (z::SE = x)
and ((u := droponex(first l, a, ka, zz)) case F) => u::F::IR
(da := differentiate(a := denom(f)::F, x)) ^= 0 and
zero? differentiate(c := numer(f)::F / da, x) => (c * log a)::IR
mkAnswer(0, empty(), [\[f, x::F\]])

droponex(f, a, ka, x) ==
(r := retractIfCan(f)@Union(K, "failed")) case "failed" => "failed"
is?(op := operator(k := r::K), OPDIFF) =>
(z := third(arg := argument k)) = a => op [first arg, second arg, x]
(u := droponex(first arg, a, ka, x)) case "failed" => "failed"
op [u::F, second arg, z]
eval(f, [ka], [x])

unklimint(f, x, lu) ==
for u in lu | u ^= 0 repeat
zero? differentiate(c := f * u / differentiate(u, x), x) => [0,[[c,u]]]
"failed"
unkextint(f, x, g) ==
  zero?(g' := differentiate(g, x)) => "failed"
  zero? differentiate(c := f / g', x) => [0, c]
"failed"

isx?(f, x) ==
  (k := retractIfCan(f)@Union(K, "failed")) case "failed" => false
  (r := symbolIfCan(k::K)) case "failed" => false
  r::SE = x

alglfint(f, k, l, x) ==
  xf := x::F
  symbolIfCan(kx := ksec(k,l,x)) case SE => addx(palgint(f, kx, k), xf)
  is?(kx, "exp":SE) => addx(algexpint(f, kx, k, x), xf)
  prim?(kx, x) => addx(algprimint(f, kx, k, x), xf)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y := rootOf(rec.prim)
    map((x1:F):F +-> eval(x1, retract(y)@K, rec.primelt),
    lfintegrate(eval(f, [kx,k], [(rec.pol1) y, (rec.pol2) y]), x))
  unknowmint(f, x)

alglfextint(f, k, l, x, g) ==
  symbolIfCan(kx := ksec(k,l,x)) case SE => palgextint(f, kx, k, g)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y := rootOf(rec.prim)
    lrhs := [(rec.pol1) y, (rec.pol2) y]$List(F)
    (u := lfextendedint(eval(f, [kx,k], lrhs), x,
      eval(g, [kx,k], lrhs))) case "failed" => "failed"
    ky := retract(y)@K
    r := u::Record(ratpart:F, coeff:F)
    [eval(r.ratpart,ky,rec.primelt), eval(r.coeff,ky,rec.primelt)]
  is?(kx, "exp":SE) or is?(kx, "log":SE) => FAIL
  unkextint(f, x, g)

alglflimint(f, k, l, x, lu) ==
  symbolIfCan(kx := ksec(k,l,x)) case SE => palglimitint(f, kx, k, lu)
  has?(operator kx, ALGOP) =>
    rec := primitiveElement(kx::F, k::F)
    y := rootOf(rec.prim)
    lrhs := [(rec.pol1) y, (rec.pol2) y]$List(F)
    (u := lflimitedint(eval(f, [kx,k], lrhs), x,
      map((x1:F):F +-> eval(x1,[kx,k],lrhs), lu))) case "failed" => "failed"
    ky := retract(y)@K
    r := u::Record(mainpart:F, limitedlogs:LLG)
    [eval(r.mainpart,ky,rec.primelt),
      map([rc.coeff, ky, rec.primelt] for rc in r.limitedlogs)]
  is?(kx, "exp":SE) or is?(kx, "log":SE) => FAIL
unklimint(f, x, lu)

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
and F has Join(LiouvilleFunctionCategory, RetractableTo SE)
then
import PatternMatchIntegration(R, F)
lfintegrate(f, x) == intPatternMatch(f, x, lfintegrate0, pmintegrate)
else lfintegrate(f, x) == lfintegrate0(f, x)

lfintegrate0(f, x) ==
zero? f => 0
xf := x::F
empty?(l := varselect(kernels f, x)) => (xf * f)::IR
symbolIfCan(k := kmax l) case SE =>
map((x1:RF):F +-> multivariate(x1, k), integrate univariate(f, k))
is?(k, "tan":SE) => addx(tanint(f, x, k), xf)
is?(k, "exp":SE) => addx(expint(f, x, k), xf)
prim?(k, x) => addx(primint(f, x, k), xf)
has?(operator k, ALGOP) => alglfint(f, k, l, x)
unknownint(f, x)

addx(i, x) ==
elem? i => i
mkAnswer(ratpart i, logpart i,
[[ne.integrand, x] for ne in notelem i])

tryChangeVar(f, t, x) ==
z := new()$Symbol
  g := subst(f / differentiate(t::F, x), [t], [z::F])
  freeOf?(g, x) => -- can we do change of variables?
    map((x1:F):F +-> eval(x1, kernel z, t::F), lfintegrate(g, z))
"failed"

algexpint(f, t, y, x) ==
  (u := tryChangeVar(f, t, x)) case IR => u::IR
  algint(f, t, y,
    (x1:UP):UP +-> differentiate(x1,
      (x2:F):F +-> differentiate(x2, x),
      monomial(differentiate(first argument t, x), 1)))
algprimint(f, t, y, x) ==
  (u := tryChangeVar(f, t, x)) case IR => u::IR
  algint(f, t, y,
    (x1:UP):UP +-> differentiate(x1,
      (x2:F):F +-> differentiate(x2, x),
      differentiate(t::F, x)::UP))
Bug #100 is an infinite loop that eventually kills Axiom from the input

\[
\text{integrate}((z^{a+1})^b, z)
\]

Line 2 of this function used to read:

\[
\text{symbolIfCan}(k := \text{kmax}(l := \text{union}(l, \text{varselect}(\text{kernels} g, x))))
\]

The loop occurs when the call to union causes

\[
\text{a log}(z) \ \ %e
\]

to get added to the list every time. This gives the argument to kmax

\[
\text{a log}(z) \\
\text{arg1} = [z, %e]
\]

and the result being

\[
\text{a log}(z) \\
%e
\]

We keep coming back to process this term, which ends up putting the same term back on the list and we loop. Waldek's solution is to remove the union call.

The original patch fixed the infinite regression mentioned above but caused Axiom to return a closed form of the integral:

\[
\text{integrate}(\text{asech}(x)/x, x)
\]

which should not have a closed form. This is referenced in the FriCAS SVN revision 279.

Essentially this new patch uses only logarithms of rational functions when integrating rational functions. It is unclear whether this is the correct fix.

— package INTEF ElementaryIntegration —

\[
\text{lfextendedint}(f, x, g) ==
\text{empty?(l := \text{varselect}(\text{kernels} f, x))} \Rightarrow [x::F * f, 0]
\text{symbolIfCan}(k := \text{kmax}(l))
\text{case \text{SE} =>}
\text{g1 :=}
\text{empty?(l1 := \text{varselect}(\text{kernels} g, x))} \Rightarrow 0::F
\text{kmax(l1) = k} \Rightarrow g
0::F
\text{map}((x1:RF):F \mapsto \text{multivariate}(x1, k),
\text{extendedint}((\text{univariate}(f, k),
\text{univariate}(g1, k))))
\text{is?(k, "exp", \text{SE})} \Rightarrow \text{expextint}(f, x, k, g)
\]
PACKAGE INTEF ELEMENTARY INTEGRATION

This is part of the fix for bug 100. Line 2 of this function used to read:

```plaintext
symbolIfCan(k := kmax(l := union(l, vark(lu, x)))) case SE =>
```

See the above discussion for why this causes an infinite loop.

--- package INTEF ElementaryIntegration ---

```plaintext
lflimitedint(f, x, lu) ==
  empty?(l := varselect(kernels f, x)) => [x::F * f, empty()]
  symbolIfCan(k := kmax(l)) case SE =>
  map((x1:RF):F +-> multivariate(x1, k),
      limitedint(univariate(f, k),
      [univariate(u, k) for u in lu]))
  is?(k, "exp":SE) => explimint(f, x, k, lu)
  prim?(k, x) => primlimint(f, x, k, lu)
  has?(operator k, ALGOP) => alglfextint(f, k, l, x, g)
  unkextint(f, x, g)
```

```plaintext
lfinfieldint(f, x) ==
  (u := lfextendedint(f, x, 0)) case "failed" => "failed"
  u.ratpart
```

```plaintext
primextint(f, x, k, g) ==
  lk := varselect([a for a in tower f | k ^= a and is?(a, "log":SE)], x)
  (u1 := primextendedint(univariate(f, k),
    (x1:UP):UP +-> differentiate(x1,
    (x2:F):F +-> differentiate(x2, k),
    (x3:F):U2+->lfextlimint(x3,x,k,lk), univariate(g, k))) case "failed"
    => "failed"
  u1 case FF =>
    [multivariate(u1.ratpart, k), multivariate(u1.coeff, k)]
  (u2 := lfextendedint(u1.a0, x, g)) case "failed" => "failed"
  [multivariate(u1.answer, k) + u2.ratpart, u2.coeff]
```

```plaintext
expextint(f, x, k, g) ==
  (u1 := expextendedint(univariate(f, k),
    (x1:UP):UP +-> differentiate(x1,
    (x2:F):F +-> differentiate(x2, x),
    monomial(differentiate(first argument k, x), 1)),
    (x3:Integer,x4:F):PSOL+->rischDE(x3, first argument k, x4, x,
    (x5:F,x6:List F):U3 +-> lflimitedint(x5, x, x6),
    (x7:F,x8:F):U2+->lfextendedint(x7, x, x8)), univariate(g, k)))
  case "failed" => "failed"
```
u1 case FF =>
  [multivariate(u1.ratpart, k), multivariate(u1.coeff, k)]
(u2 := lftendedint(u1.a0, x, g)) case "failed" => "failed"
  [multivariate(u1.answer, k) + u2.ratpart, u2.coeff]

primint(f, x, k) ==
  lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
  r1 := primintegrate(univariate(f, k),
        (x1:UP):UP +-> differentiate(x1,
        (x2:F):F +-> differentiate(x2, x), differentiate(k::F, x)::UP),
        (x3:F):U2 +-> lftlimint(x3, x, k, lk))
  map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

lftlimint(f, x, k, lk) ==
  not((u1 := lftendedint(f, x, differentiate(k::F, x)))
    case "failed") => u1
  twr := tower f
  empty?(lg := [kk for kk in lk | not member?(kk, twr)]) => "failed"
  is?(k, "log"::SE) =>
    (u2 := lflimitedint(f, x, [first argument u for u in union(lg, [k])])
      case "failed" => "failed"
    cf := cfind(first argument k, u2.limitedlogs)
    [u2.mainpart - cf * k::F +
      +/[c.coeff * log(c.logand) for c in u2.limitedlogs], cf]
  "failed"

cfind(f, l) ==
  for u in l repeat
    f = u.logand => return u.coeff
  0

expint(f, x, k) ==
  eta := first argument k
  r1 :=
    expintegrate(univariate(f, k),
        (x1:UP):UP +-> differentiate(x1,
        (x2:F):F +-> differentiate(x2, x), monomial(differentiate(eta, x), 1)),
        (x3:Integer,x4:F):PSOL+->rischDE(x3, eta, x4, x,
        (x5:F,x6:List F):U3 +-> 1limitedint(x5, x, x6),
        (x7:F,x8:F):U2+->lftendedint(x7, x, x8)))
  map((x1:RF):F+->multivariate(x1, k), r1.answer) + lfintegrate(r1.a0, x)

primlimint(f, x, k, lu) ==
  lk := varselect([a for a in tower f | k ^= a and is?(a, "log"::SE)], x)
  (u1 :=
    primlimitedint(univariate(f, k),
        (x1:UP):UP+->differentiate(x1,
        (x2:F):F+->differentiate(x2, x), differentiate(k::F, x)::UP),
        (x3:F):U2 +-> lftlimint(x3, x, k, lk))
  0
package RDEEF ElementaryRischDE

---

INTEF.dotabb ---

"INTEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTEF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTEF" -> "ACF"
"INTEF" -> "FS"

---

package RDEEF ElementaryRischDE

--- ElementaryRischDE.input ---

)set break resume
)sys rm -f ElementaryRischDE.output
)spool ElementaryRischDE.output
\texttt{\textbackslash set message test on}
\texttt{\textbackslash set message auto off}
\texttt{\textbackslash clear all}

--- S 1 of 1
\texttt{\textbackslash show ElementaryRischDE}
--- R
--- R ElementaryRischDE(R: Join(GcdDomain,OrderedSet,CharacteristicZero,RetractableTo(Integer),Lin
--- R Abbreviation for ElementaryRischDE is RDEEF
--- R This constructor is not exposed in this frame.
--- R Issue \texttt{\textbackslash edit bookvol10.4.pamphlet} to see algebra source code for RDEEF
--- R
--- R-------------------------------------- Operations --------------------------------------
--- R rischDE : (Integer,F,F,Symbol,((F,List(F)) \rightarrow \text{Union(Record(mainpart: F,limitedlogs: List(Rec}
--- R
--- E 1

\texttt{\textbackslash spool}
\texttt{\textbackslash lisp (bye)}

\text{---

\texttt{\textbackslash ElementaryRischDE.help --}

\text{===============================================
ElementaryRischDE examples
\text{===============================================

Risch differential equation, elementary case.

See Also:
\texttt{\textbackslash show ElementaryRischDE

\text{---}
ElementaryRischDE (RDEEF)

Exports:
rischDE

— package RDEEF ElementaryRischDE —

)abbrev package RDEEF ElementaryRischDE
++ Author: Manuel Bronstein
++ Date Created: 1 February 1988
++ Date Last Updated: 2 November 1995
++ Description:
++ Risch differential equation, elementary case.

ElementaryRischDE(R, F):Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, AlgebraicallyClosedField,
           FunctionSpace R)

N  ==> NonNegativeInteger
Z  ==> Integer
SE ==> Symbol
LF ==> List F
K  ==> Kernel F
LK ==> List K
P  ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
RF ==> Fraction UP
GP ==> LaurentPolynomial(F, UP)
Data ==> List Record(coef:Z, argument:P)
RRF ==> Record(mainpart:F,limitedlogs:List NL)
NL ==> Record(coef:F,logand:F)
U  ==> Union(RRF, "failed")
UF ==> Union(F, "failed")
UUP ==> Union(UP, "failed")
UGP ==> Union(GP, "failed")
URF ==> Union(RF, "failed")
UEX ==> Union(Record(ratpart:F, coeff:F), "failed")
PSOL==> Record(ans:F, right:F, sol?:Boolean)
FAIL==> error("Function not supported by Risch d.e.")
ALGOP ==> "%alg"

Exports ==> with
  rischDE: (Z, F, F, SE, (F, LF) -> U, (F, F) -> UEX) -> PSOL
    ++ rischDE(n, f, g, x, lim, ext) returns \spad{[y, h, b]} such that
    ++ \spad{dy/dx + n df/dx y = h} and \spad{b := h = g}.
    ++ The equation \spad{dy/dx + n df/dx y = g} has no solution
    ++ if \spad{h \not\sim g} (\text{\textit{y}} is a partial solution in that case).
    ++ Notes: \textit{lim} is a limited integration function, and
    ++ \textit{ext} is an extended integration function.

Implementation ==> add
  import IntegrationTools(R, F)
  import TranscendentalRischDE(F, UP)
  import TranscendentalIntegration(F, UP)
  import PureAlgebraicIntegration(R, F, F)
  import FunctionSpacePrimitiveElement(R, F)
  import ElementaryFunctionStructurePackage(R, F)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

RF2GP: RF -> GP
makeData : (F, SE, K) -> Data
normal0 : (Z, F, F, SE) -> UF
normalise0: (Z, F, F, SE) -> PSOL
normalise : (Z, F, F, F, SE, K, (F, LF) -> U, (F, F) -> UEX) -> PSOL
rischDEalg: (Z, F, F, F, K, LK, SE, (F, LF) -> U, (F, F) -> UEX) -> PSOL
rischDElog: (LK, RF, RF, SE, K, UP->UP,(F,LF)->U,(F,F)->UEX) -> URF
rischDEexp: (LK, RF, RF, SE, K, UP->UP,(F,LF)->U,(F,F)->UEX) -> URF
boundAt0 : (LK, F, Z, Z, SE, K, (F, LF) -> U) -> Z
boundInf : (LK, F, Z, Z, SE, K, (F, LF) -> U) -> Z
logdegrad : (LK, F, UP, Z, SE, K, (F, LF) -> U) -> Z
expdegrad : (LK, F, UP, Z, SE, K, (F, LF) -> U) -> Z
logdeg : (UP, F, Z, SE, F, (F, LF) -> U) -> Z
expdeg : (UP, F, Z, SE, F, (F, LF) -> U) -> Z
exppolyint: (UP, (Z, F) -> PSOL) -> UUP
RRF2F : RRF -> F
logdiff : (List K, List K) -> List K

tab:AssociationList(F, Data) := table()

RF2GP f == (numer(f)::GP exquo denom(f)::GP)::GP
logdiff(twr, bad) ==
[u for u in twr | is?(u, "log"::SE) and not member?(u, bad)]

rischDEalg(n, nfp, f, g, k, l, x, limint, extint) ==
symbolIfCan(kx := ksec(k, l, x)) case SE =>
(u := palgRDE(nfp, f, g, kx, k,
 (z1,z2,z3) +-> normal0(n, z1, z2, z3))) case "failed"
 => [0, 0, false]
[u::F, g, true]
has?(operator kx, ALGOP) =>
rec := primitiveElement(kx::F, k::F)
lk:LK := [kx, k]
lv:LF := [(rec.pol1) y, (rec.pol2) y]
rc := rischDE(n, eval(f, lk, lv), eval(g, lk, lv), x, limint, extint)
rc.sol? => [eval(rc.ans, retract(y)@K, rec.primelt), rc.right, true]
[0, 0, false]
FAIL

-- solve y' + n f'y = g for a rational function y
rischDE(n, f, g, x, limitedint, extendedint) ==
zero? g => [0, g, true]
zero?(nfp := n * differentiate(f, x)) =>
(u := limitedint(g, empty())) case "failed" => [0, 0, false]
[u.mainpart, g, true]
freeOf?(y := g / nfp, x) => [y, g, true]
vl := varselect(union(kernels nfp, kernels g), x)
symbolIfCan(k := kmax vl) case SE => normalise0(n, f, g, x)
is?(k, "log"::SE) or is?(k, "exp"::SE) =>
normalise(n, nfp, f, g, x, k, limitedint, extendedint)
has?(operator k, ALGOP) =>
rischDEalg(n, nfp, f, g, k, vl, x, limitedint, extendedint)
FAIL

normal0(n, f, g, x) ==
rec := normalise0(n, f, g, x)
rec.sol? => rec.ans
"failed"

-- solve y' + n f' y = g
-- when f' and g are rational functions over a constant field
normalise0(n, f, g, x) ==
k := kernel(x)@K
if (data1 := search(f, tab)) case "failed" then
tab.f := data := makeData(f, x, k)
else data := data1::Data
f' := nfprime := n * differentiate(f, x)
p:P := 1
for v in data | (m := n * v.coeff) > 0 repeat
  p := p * v.argument ** (m::N)
\[
f' := f' - m \times \text{differentiate}(v\text{.argument::F}, x) / (v\text{.argument::F})
\]
\[
\text{rec} := \text{baseRDE}(\text{univariate}(f', k), \text{univariate}(p\text{::F} \times g, k))
\]
\[
y := \text{multivariate}(\text{rec.ans}, k) / p\text{::F}
\]
\[
\text{rec.nosol} \Rightarrow [y, \text{differentiate}(y, x) + nfprime \times y, \text{false}]
\]
\[
[y, g, \text{true}]
\]

\[\text{-- make f weakly normalized, and solve } y' + n f' y = g\]
\[
\text{normalise}(n, nf, f, g, x, k, \text{limitedint}, \text{extendedint}) ==
\]
\[
\text{if (data1:= search(f, tab)) case "failed" then}
\quad \text{tab.f := data := makeData(f, x, k)}
\]
\[
\text{else data := data1::Data}
\quad p:P := 1
\quad \text{for v in data | (m := n \times v.coeff) > 0 repeat}
\quad \quad p := p \times v\text{.argument}^{(m::N)}
\quad \quad f := f - v.coeff \times \log(v\text{.argument::F})
\quad \quad nf := nf - m \times \text{differentiate}(v\text{.argument::F}, x) / (v\text{.argument::F})
\quad \quad newf := \text{univariate}(nf, k)
\quad \quad newg := \text{univariate}(p\text{::F} \times g, k)
\quad \text{twr := union(logdiff(tower f, empty()), logdiff(tower g, empty()))}
\quad \text{ans1 :=}
\quad \quad \text{is?(k, "log"::SE) =>}
\quad \quad \quad \text{rischDElog(twr, newf, newg, x, k,}
\quad \quad \quad \quad \quad z1 +\to \text{differentiate}(z1, (z2:F) : F +\to \text{differentiate}(z2, x),
\quad \quad \quad \quad \quad \text{differentiate(k::F, x)::UP},
\quad \quad \quad \quad \quad \text{limitedint, extendedint)}
\quad \quad \text{is?(k, "exp"::SE) =>}
\quad \quad \quad \text{rischDEexp(twr, newf, newg, x, k,}
\quad \quad \quad \quad \quad z1 +\to \text{differentiate}(z1, (z2:F) : F +\to \text{differentiate}(z2, x),
\quad \quad \quad \quad \quad \text{monomial(differentiate(first argument k, x), 1)),
\quad \quad \quad \quad \quad \text{limitedint, extendedint)}
\quad \quad \text{ans1 case "failed" => [0, 0, false]}
\quad [\text{multivariate(ans1::RF, k) / p::F, g, true}]
\]

\[\text{-- find the } n \times \log(P) \text{ appearing in } f, \text{ where } P \text{ is in } P, n \in \mathbb{Z}\]
\[
\text{makeData}(f, x, k) ==
\text{disasters := empty()$Data}
\text{fnum := numer f}
\text{fden := denom f}
\text{for u in varselect(kernels f, x) | is?(u, "log"::SE) repeat}
\text{logand := first argument u}
\quad \text{if zero?(degree univariate(fden, u)) and}
\quad \quad \text{one?(degree(num := univariate(fnum, u))) then}
\quad \quad \text{(degree(num := univariate(fnum, u)) = 1) then}
\quad \quad \text{cf := (leadingCoefficient num) / fden}
\quad \quad \text{if (n := retractIfCan(cf)@Union(Z, "failed")) case Z then}
\quad \quad \text{if degree(numer logand, k) > 0 then}
\quad \quad \quad \text{disasters := concat([n::Z, numer logand], disasters)}
\quad \quad \text{if degree(denom logand, k) > 0 then}
\quad \quad \quad \text{disasters := concat([-n::Z], denom logand], disasters)}
\quad \text{disasters}
rischDElog(twr, f, g, x, theta, driv, limint, extint) ==
  (u := monomRDE(f, g, driv)) case "failed" => "failed"
  (v := polyDElog(twr, u.a, retract(u.b), retract(u.c), x, theta, driv, limint, extint)) case "failed" => "failed"
  v::UP / u.t

rischDEexp(twr, f, g, x, theta, driv, limint, extint) ==
  (u := monomRDE(f, g, driv)) case "failed" => "failed"
  (v := gpolDEexp(twr, u.a, RF2GP(u.b), RF2GP(u.c), x, theta, driv, limint, extint)) case "failed" => "failed"
  convert(v::GP)@RF / u.t::RF

polyDElog(twr, aa, bb, cc, x, t, driv, limint, extint) ==
  zero? cc => 0
  t' := differentiate(t::F, x)
  zero? bb =>
    (u := cc exquo aa) case "failed" => "failed"
    primintfldpoly(u::UP, z1 +-> extint(z1, t'), t')
  n := degree(cc)::Z - (db := degree(bb)::Z)
  if ((da := degree(aa)::Z) = db) and (da > 0) then
    lk0 := tower(f0 :=
      - (leadingCoefficient bb) / (leadingCoefficient aa))
    lk1 := logdiff(twr, lk0)
    (if0 := limint(f0, [first argument u for u in lk1]))
      case "failed" => error "Risch's theorem violated"
    (alph := validExponential(lk0, RRF2F(if0::RRF), x)) case F =>
      return
    (ans := polyDElog(twr, alph::F * aa,
        differentiate(alph::F, x) * aa + alph::F * bb,
        cc, x, t, driv, limint, extint)) case "failed" => "failed"
    alph::F * ans::UP
  if (da > db + 1) then n := max(0, degree(cc)::Z - da + 1)
  if (da = db + 1) then
    i := limint(- (leadingCoefficient bb) / (leadingCoefficient aa),
        [first argument t])
  if not(i case "failed") then
    r :=
      null(i.limitedlogs) => 0$F
      i.limitedlogs.first.coeff
    if (nn := retractIfCan(r)@Union(Z, "failed")[0]) then
      n := max(nn::Z, n)
  (v := polyRDE(aa, bb, cc, n, driv)) case ans =>
    v.ans.nosol => "failed"
    v.ans.ans
  w := v.eq
  zero?(w.b) =>
    degree(w.c) > w.m => "failed"
    (u := primintfldpoly(w.c, zi +-> extint(1, t'), t')) case "failed" => "failed"
degree(u::UP) > w.m => "failed"
  w.alpha * u::UP + w.beta
(u := logdegrad(twr, retract(w.b), w.c, w.m, x, t, limint, extint))
case "failed" => "failed"
  w.alpha * u::UP + w.beta

gpolDEexp(twr, a, b, c, x, t, driv, limint, extint) ==
  zero? c => 0
  zero? b =>
    (u := c exquo (a::GP)) case "failed" => "failed"
    expintfldpoly(u::GP,
      (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint))
  lb := boundAt0(twr, - coefficient(b, 0) / coefficient(a, 0),
    nb := order b, nc := order c, x, t, limint)
  tm := monomial(1, (m := max(0, max(-nb, lb - nc)))::N)$UP
  (v := polyDEexp(twr,a * tm,lb * differentiate(first argument t, x)
    * a * tm + retract(b * tm::GP)$UP,
    retract(c * monomial(1, m - lb))$UP,
    x, t, driv, limint, extint)) case "failed" => "failed"
  v::UP::GP * monomial(1, lb)

polyDEexp(twr, aa, bb, cc, x, t, driv, limint, extint) ==
  zero? cc => 0
  zero? bb =>
    (u := cc exquo aa) case "failed" => "failed"
    exppolyint(u::UP,
      (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint))
  n := boundInf(twr,-leadingCoefficient(bb) / (leadingCoefficient aa),
    degree(aa)::Z, degree(bb)::Z, degree(cc)::Z, x, t, limint)
  (v := polyRDE(aa, bb, cc, n, driv)) case ans =>
    v.ans.nosol => "failed"
    v.ans.ans
    v := v.eq
    zero?(v.b) =>
      degree(w.c) > w.m => "failed"
      (u := exppolyint(w.c,
        (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint)))
      case "failed" => "failed"
      w.alpha * u::UP + w.beta
    (u := expdegrad(twr, retract(w.b), w.c, w.m, x, t, limint, extint))
    case "failed" => "failed"
    w.alpha * u::UP + w.beta
  exppolyint(p, rischdiffeq) ==
    (u := expintfldpoly(p::GP, rischdiffeq)) case "failed" => "failed"
    retractIfCan(u::GP)$Union(UP, "failed")

boundInf(twr, f0, da, db, dc, x, t, limitedint) ==
  da < db => dc - db
  da > db => max(0, dc - da)
l1 := logdiff(twr, l0 := tower f0)
(if0 := limitedint(f0, [first argument u for u in l1]))
  case "failed" => error "Risch's theorem violated"
  (alpha := validExponential(concat(t, l0), RRF2F(if0::RRF), x))
  case F =>
    al := separate(univariate(alpha::F, t))$GP
    zero?(al.fracPart) and monomial?(al.polyPart) =>
      max(0, max(degree(al.polyPart), dc - db))
  dc - db

boundAt0(twr, f0, nb, nc, x, t, limitedint) ==
  nb ^= 0 => min(0, nc - min(0, nb))
l1 := logdiff(twr, l0 := tower f0)
  (if0 := limitedint(f0, [first argument u for u in l1]))
  case "failed" => error "Risch's theorem violated"
  (alpha := validExponential(concat(t, l0), RRF2F(if0::RRF), x))
  case F =>
    al := separate(univariate(alpha::F, t))$GP
    zero?(al.fracPart) and monomial?(al.polyPart) =>
      min(0, min(degree(al.polyPart), nc))
  min(0, nc)

-- case a = 1, deg(B) = 0, B <> 0
-- cancellation at infinity is possible
logdegrad(twr, b, c, n, x, t, limitedint, extint) ==
  t' := differentiate(t::F, x)
lk1 := logdiff(twr, lk0 := tower(f0 := - b))
  (if0 := limitedint(f0, [first argument u for u in lk1]))
  case "failed" => error "Risch's theorem violated"
  (alpha := validExponential(lk0, RRF2F(if0::RRF), x))
  case F =>
    (u1 := primintfldpoly(inv(alpha::F) * c, z1+->extint(z1, t'), t'))
    case "failed" => "failed"
  degree(u1::UP)::Z > n => "failed"
  alpha::F * u1::UP
logdeg(c, - if0.mainpart -
  +/[v.coef * log(v.logand) for v in if0.limitedlogs],
  n, x, t', limitedint, extint) ==

-- case a = 1, degree(b) = 0, and (exp integrate b) is not in F
-- this implies no cancellation at infinity
logdeg(c, f, n, x, t', limitedint, extint) ==
  answr:UP := 0
  repeat
    zero? c => return answr
    zero? c or (m := degree(c)::Z > n) => return "failed"
    u := rischDE(1, f, leadingCoefficient c, x, limitedint, extint)
    "u.sol?" => return "failed"
    zero? m => return(answr + u.an::UP)
\begin{verbatim}
606  \textbf{CHAPTER 6. CHAPTER E}

\begin{verbatim}
n := m::Z - 1
c := (reductum c) - monomial(m::Z * t' * u.ans, (m-1)::N)
answr := answr + monomial(u.ans, m)

-- case a = 1, deg(B) = 0, B <> 0
-- cancellation at infinity is possible
expdegrad(twr, b, c, n, x, t, limint, extint) ==
lk1 := logdiff(twr, lk0 := tower(f0 := - b))
(if0 := limint(f0, [first argument u for u in lk1]))
  case "failed" => error "Risch's theorem violated"
  intf0 := - if0.mainpart -
  +\[v.coeff * log(v.logand) for v in if0.limitedlogs\]
(alpha := validExponential(concat(t, lk0), RRF2F(if0::RRF), x))
case F =>
al := separate(univariate(alpha::F, t))$GP
zero?(al.fracPart) and monomial?(al.polyPart) and
  (degree(al.polyPart) <= 0) =>
  (u1 := expintfldpoly(c::GP * recip(al.polyPart)::GP,
    (z1,z2) +-> rischDE(z1, first argument t, z2, x, limint, extint)))
  case "failed" => "failed"
  degree(u1::GP) > n => "failed"
  retractIfCan(al.polyPart * u1::GP)@Union(UP, "failed")
expdeg(c, intf0, n, x, first argument t, limint, extint)
expdeg(c, intf0, n, x, first argument t, limint, extint)

-- case a = 1, degree(b) = 0, and (exp integral b) is not a monomial
-- this implies no cancellation at infinity
expdeg(c, f, n, x, eta, limitedint, extint) ==
answr:UP := 0
repeat
  zero? c => return answr
  (n < 0) or ((m := degree c)::Z > n) => return "failed"
u := rischDE(1, f + m * eta, leadingCoefficient c, x, limitedint, extint)
  "u.sol? => return "failed"
zero? m => return(answr + u.ans::UP)
n := m::Z - 1
c := reductum c
answr := answr + monomial(u.ans, m)

RRF2F rrf ==
  rrf.mainpart + +\[v.coeff*log(v.logand) for v in rrf.limitedlogs\]

\end{verbatim}
\end{verbatim}

--- RDEEF.dotabb ---

"RDEEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDEEF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
package RDEEFS ElementaryRischDESystem

--- ElementaryRischDESystem.input ---

)set break resume
)sys rm -f ElementaryRischDESystem.output
)spool ElementaryRischDESystem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ElementaryRischDESystem
--R ElementaryRischDESystem(R: Join(GcdDomain,OrderedSet,CharacteristicZero,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),F: Join(TranscendentalFunctionCategory,AlgebraicallyClosedField,FunctionSpace(R))) is a package constructor
--R Abbreviation for ElementaryRischDESystem is RDEEFS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RDEEFS
--R
--R------------------------------- Operations --------------------------------
--R rischDEsys : (Integer,F,F,F,Symbol,((F,List(F)) -> Union(Record(mainpart: F,limitedlogs: List(Record(coeff: F,logand: F)),"failed"),"failed")) -> Union(List(F),"failed")
--R
--E 1

)spool
)lisp (bye)

---

--- ElementaryRischDESystem.help ---

====================================================================
ElementaryRischDESystem examples
====================================================================

Risch differential equation, elementary case.

See Also:
o )show ElementaryRischDESystem
ElementaryRischDESystem (RDEEFS)

Exports:
rischDEsys

— package RDEEFS ElementaryRischDESystem —

)abbrev package RDEEFS ElementaryRischDESystem
++ Author: Manuel Bronstein
++ Date Created: 12 August 1992
++ Date Last Updated: 17 August 1992
++ Description:
++ Risch differential equation, elementary case.

ElementaryRischDESystem(R, F): Exports == Implementation where
  R : Join(GcdDomain, OrderedSet, CharacteristicZero,
        RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, AlgebraicallyClosedField,
        FunctionSpace R)
  Z ==> Integer
  SE ==> Symbol
  K ==> Kernel F
  P ==> SparseMultivariatePolynomial(R, K)
  UP ==> SparseUnivariatePolynomial F
  RF ==> Fraction UP
  NL ==> Record(coeff:F,logand:F)
  RRF ==> Record(mainpart:F,limitedlogs:List NL)
  U ==> Union(RRF, "failed")
  ULF ==> Union(List F, "failed")
  UEX ==> Union(Record(ratpart:F, coeff:F), "failed")
Exports ==> with
rischDEsys: (Z, F, F, F, SE, (F, List F) -> U, (F, F) -> UEX) -> ULF
  ++ rischDEsys(n, f, g_1, g_2, x, lim, ext) returns \( y_1, y_2 \) such that
  ++ \( \frac{dy_1}{dx}, \frac{dy_2}{dx} + \left( (0, -n \frac{df}{dx}), (n \frac{df}{dx}, 0) \right) (y_1, y_2) = (g_1, g_2) \)
  ++ if \( y_1, y_2 \) exist, "failed" otherwise.
  ++ lim is a limited integration function,
  ++ ext is an extended integration function.

Implementation ==> add
import IntegrationTools(R, F)
import ElementaryRischDE(R, F)
import TranscendentalRischDESystem(F, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

-- sm1 := sqrt(-1::F)
-- ks1 := retract(sm1)@K

-- gcoeffs : P -> ULF
-- gets1coeffs: F -> ULF
-- cheat : (Z, F, F, F, SE, (F, List F) -> U, (F, F) -> UEX) -> ULF
basecase : (F, F, F, K) -> ULF
  basecase(nfp, g1, g2, k) ==
    (ans := baseRDEsys(univariate(nfp, k), univariate(g1, k), univariate(g2, k))) case "failed" => "failed"
    l := ans::List(RF)
    [multivariate(first l, k), multivariate(second l, k)]

-- solve \( (y_1', y_2') + ((0, -nfp), (nfp, 0)) (y_1, y_2) = (g_1, g_2) \), base case
basecase(nfp, g1, g2, k) ==
  (ans := baseRDEsys(univariate(nfp, k), univariate(g1, k), univariate(g2, k))) case "failed" => "failed"
  l := ans::List(RF)
  [multivariate(first l, k), multivariate(second l, k)]

-- returns \( [x, y] \) s.t. \( f = x + y \cdot \text{i} \)
-- f can be of the form \( (a + b \cdot \text{i}) / (c + d \cdot \text{i}) \)
-- gets1coeffs f ==
-- (l1num := gcoeffs(numer f)) case "failed" => "failed"
-- (l1den := gcoeffs(denom f)) case "failed" => "failed"
-- a := first(l1num::List F)
-- b := second(l1num::List F)
-- c := first(l1den::List F)
-- d := second(l1den::List F)
-- degree(q := univariate(p, ks1)) > 1 => "failed"
-- [coefficient(q, 0)::F, coefficient(q, 1)::F]
-- cheat(n, f, g1, g2, x, limint, extint) ==
-- (u := rischDE(n, sm1 * f, g1 + sm1 * g2, x, limint, extint))
-- case "failed" => "failed"
-- (l := gets1coeffs(u::F)) case "failed" =>
error "rischDEsys: expect linear result in sqrt(-1)"

\[ l::\text{List } F \]

\[ \text{solve } (y1',y2') + ((0, -n f'), (n f', 0)) (y1,y2) = (g1, g2) \]

\[
\text{rischDEsys}(n, f, g1, g2, x, \text{limint, extint}) ==
\begin{align*}
\text{zero? } g1 \text{ and zero? } g2 \Rightarrow [0, 0] \\
\text{zero?(} nfp := n \ast \text{differentiate(f, x)} \text{) } \Rightarrow \\
\left( \begin{array}{l}
(\text{u1 := limint}(g1, \text{empty}())) \text{ case "failed" } \Rightarrow \\
(\text{u2 := limint}(g1, \text{empty}())) \text{ case "failed" } \Rightarrow "\text{failed}"
\end{array} \right) \\
\left[ \text{u1.mainpart, u2.mainpart} \right]
\end{align*}
\]

\[
\text{freeOf?}(y1 := g2 / nfp, x) \text{ and freeOf?}(y2 := - g1 / nfp, x) \Rightarrow [y1, y2]
\]

\[
\text{vl := varselect(union(kernels nfp, union(kernels g1, kernels g2)), x)}
\]

\[
\text{symbolIfCan}(k := \text{kmax vl}) \text{ case SE } \Rightarrow \text{basecase(nfp, g1, g2, k)}
\]

error "rischDEsys: can only handle rational functions for now"

---

--- RDEEFS.dotabb ---

"RDEEFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDEEFS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"RDEEFS" -> "ACF"
"RDEEFS" -> "FS"

---

package ELFUTS EllipticFunctionsUnivariateTaylorSeries

--- EllipticFunctionsUnivariateTaylorSeries.input ---

)set break resume
)sys rm -f EllipticFunctionsUnivariateTaylorSeries.output
)spool EllipticFunctionsUnivariateTaylorSeries.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EllipticFunctionsUnivariateTaylorSeries
--R
--R EllipticFunctionsUnivariateTaylorSeries(Coef: Field,UTS: UnivariateTaylorSeriesCategory(Coef)
--R Abbreviation for EllipticFunctionsUnivariateTaylorSeries is ELFUTS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ELFUTS
--R
--R------------------------------- Operations --------------------------------
--R cn : (UTS,Coef) -> UTS                                           dn : (UTS,Coef) -> UTS
--R sn : (UTS,Coef) -> UTS                                           
--R sncndn : (Stream(Coef),Coef) -> List(Stream(Coef))
--R
--E 1

)spool
)lisp (bye)

---

EllipticFunctionsUnivariateTaylorSeries (ELFUTS)

Exports:
   cn  dn  sn  sncndn
"package ELFUTS EllipticFunctionsUnivariateTaylorSeries"

)abbrev package ELFUTS EllipticFunctionsUnivariateTaylorSeries
++ Author: Bill Burge, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 17 February 1992
++ Description:
++ The elliptic functions sn, sc and dn are expanded as Taylor series.

EllipticFunctionsUnivariateTaylorSeries(Coef,UTS):
Exports == Implementation where
Coef : Field
UTS : UnivariateTaylorSeriesCategory Coef
L ==> List
I ==> Integer
RN ==> Fraction Integer
ST ==> Stream Coef
STT ==> StreamTaylorSeriesOperations Coef
YS ==> Y$ParadoxicalCombinatorsForStreams(Coef)

Exports ==> with
sn : (UTS,Coef) -> UTS
++ spad(sn(x,k)) expands the elliptic function sn as a Taylor
++ series.

cn : (UTS,Coef) -> UTS
++ spad(cn(x,k)) expands the elliptic function cn as a Taylor
++ series.

dn : (UTS,Coef) -> UTS
++ spad(dn(x,k)) expands the elliptic function dn as a Taylor
++ series.

sncndn: (ST,Coef) -> L ST
++ spad(sncndn(s,c)) is used internally.

Implementation ==> add
import StreamTaylorSeriesOperations Coef
UPS=> StreamTaylorSeriesOperations Coef
integrate => lazyIntegrate
sncndnre:(Coef,L ST,ST,Coef) -> L ST
sncndnre(k,scd,dx,sign) ==
   [integrate(0, scd.2*$UPS scd.3*$UPS dx), _
    integrate(1, sign*scd.1*$UPS scd.3*$UPS dx), _
    integrate(1,sign*k**2*$UPS scd.1*$UPS scd.2*$UPS dx)]

sncndn(z,k) ==
   empty? z => [0 :: ST,1 :: ST,1::ST]
   frst z = 0 => YS(x +-> sncndnre(k,x,deriv z,-1),3)
   error "ELFUTS:sncndn: constant coefficient should be 0"
   sn(x,k) == series sncndn.(coefficients x,k).1
\[ cn(x,k) = \text{series sncndn.}(\text{coefficients } x,k).2 \]
\[ dn(x,k) = \text{series sncndn.}(\text{coefficients } x,k).3 \]

\[
\text{package EQ2 EquationFunctions2} \\
\text{--- EquationFunctions2.input ---} \\
)set break resume 
)sys rm -f EquationFunctions2.output 
)spool EquationFunctions2.output 
)set message test on 
)set message auto off 
)clear all 

--S 1 of 1 
)show EquationFunctions2 
---R 
---R EquationFunctions2(S: Type,R: Type) is a package constructor 
---R Abbreviation for EquationFunctions2 is EQ2 
---R This constructor is exposed in this frame. 
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for EQ2 
---R 
---R---------------------------------------- Operations ---------------------------------------- 
---R map : ((S -> R),Equation(S)) -> Equation(R) 
---R 
---E 1 

)spool 
)lisp (bye) 

\[
\text{--- EquationFunctions2.help ---} 
\]
This package provides operations for mapping the sides of equations.

See Also:
- )show EquationFunctions2

---

**EquationFunctions2 (EQ2)**

Exports:
- map

--- package EQ2 EquationFunctions2 ---

)abbrev package EQ2 EquationFunctions2
++ Date Last Updated: June 3, 1991
++ Description:
++ This package provides operations for mapping the sides of equations.

EquationFunctions2(S: Type, R: Type): with
  map: (S -> R , Equation S) -> Equation R
    ++ map(f, eq) returns an equation where f is applied to the sides of eq
  == add
    map(fn, eqn) == equation(fn lhs eqn, fn rhs eqn)
package ERROR ErrorFunctions

--- ErrorFunctions.input ---

)set break resume
)sys rm -f ErrorFunctions.output
)spool ErrorFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ErrorFunctions
--R
--R ErrorFunctions is a package constructor
--R Abbreviation for ErrorFunctions is ERROR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ERROR
--R
--R------------------------------------------ Operations ------------------------------------------
--R error : String -> Exit
--R error : (String,String) -> Exit
--R error : (String,List(String)) -> Exit
--E 1

)spool
)lisp (bye)

---

--- ErrorFunctions.help ---

====================================================================
ErrorFunctions examples
====================================================================

ErrorFunctions implements error functions callable from the system
interpreter. Typically, these functions would be called in user
functions. The simple forms of the functions take one argument which is either a string (an error message) or a list of strings which all together make up a message. The list can contain formatting codes (see below). The more sophisticated versions takes two arguments where the first argument is the name of the function from which the error was invoked and the second argument is either a string or a list of strings, as above. When you use the one argument version in an interpreter function, the system will automatically insert the name of the function as the new first argument. Thus in the user interpreter function

\[
f \ x \ \Rightarrow \ \text{if } \ x \ < \ 0 \ \text{then error "negative argument" else x}
\]

the call to error will actually be of the form

\[
\text{error("f","negative argument")}
\]

because the interpreter will have created a new first argument.

Formatting codes: error messages may contain the following formatting codes (they should either start or end a string or else have blanks around them):

- `%l` start a new line
- `%ceon` start centering message lines
- `%ceoff` stop centering message lines
- `%rjon` start displaying lines "ragged left"
- `%rjoff` stop displaying lines "ragged left"
- `%i` indent following lines 3 additional spaces
- `%u` unindent following lines 3 additional spaces
- `%xN` insert N blanks (eg, `%x10` inserts 10 blanks)

Examples:

1. error "Whoops, you made a %l %ceon big %ceoff %l mistake!"
2. error ["Whoops, you made a", "%l %ceon ", "big", "%d %ceoff %l", "mistake!"]

See Also:

- `)show ErrorFunctions`
ErrorFunctions (ERROR)

Exports:
error

--- package ERROR ErrorFunctions ---

)abbrev package ERROR ErrorFunctions
++ Author: Robert S. Sutor
++ Date Created: 29 May 1990
++ Date Last Updated: 29 May 1990
++ Description:
++ ErrorFunctions implements error functions callable from the system
++ interpreter. Typically, these functions would be called in user
++ functions. The simple forms of the functions take one argument
++ which is either a string (an error message) or a list of strings
++ which all together make up a message. The list can contain
++ formatting codes (see below). The more sophisticated versions takes
++ two arguments where the first argument is the name of the function
++ from which the error was invoked and the second argument is either a
++ string or a list of strings, as above. When you use the one
++ argument version in an interpreter function, the system will
++ automatically insert the name of the function as the new first
++ argument. Thus in the user interpreter function
++ \( f(x) = \begin{cases} \text{if } x < 0 \text{ then error "negative argument" else } x \end{cases} \)
++ the call to error will actually be of the form
++ \( \text{error("f","negative argument")} \)
++ because the interpreter will have created a new first argument.
++ Formatting codes: error messages may contain the following
++ formatting codes (they should either start or end a string or
++ else have blanks around them):
++ \( \%l \) start a new line
++ \( \%ceon \) start centering message lines
++ \( \%ceoff \) stop centering message lines
++ \( \%rjon \) start displaying lines "ragged left"
++ \( \%rjoff \) stop displaying lines "ragged left"
ErrorFunctions() : Exports == Implementation where
  Exports ==> with
    error: String -> Exit
    ++ error(msg) displays error message msg and terminates.
    error: List String -> Exit
    ++ error(lmsg) displays error message lmsg and terminates.
    error: (String,String) -> Exit
    ++ error(nam,msg) displays error message msg preceded by a
    ++ message containing the name nam of the function in which
    ++ the error is contained.
    error: (String,List String) -> Exit
    ++ error(nam,lmsg) displays error messages lmsg preceded by a
    ++ message containing the name nam of the function in which
    ++ the error is contained.

  Implementation ==> add

  prefix1 : String := "Error signalled from user code: %l ":
  prefix2 : String := "Error signalled from user code in function ":

  doit(s : String) : Exit ==
    throwKeyedMsg(s,nil$(List String))$Lisp
    -- there are no objects of type Exit, so we'll fake one,
    -- knowing we will never get to this step anyway.
    "exit" pretend Exit

  error(s : String) : Exit ==
    doit concat [prefix1,s]

  error(l : List String) : Exit ==
    s : String := prefix1
    for x in l repeat s := concat [s," ",x]
    doit s

  error(fn : String,s : String) : Exit ==
    doit concat [prefix2,fn," ":,%l ",",s]

  error(fn : String, l : List String) : Exit ==
    s : String := concat [prefix2,fn," ":,%l"
    for x in l repeat s := concat [s," ",x]
    doit s
package GBEUCLID EuclideanGroebnerBasisPackage

)set break resume
/sys rm -f EuclideanGroebnerBasisPackage.output
/spool EuclideanGroebnerBasisPackage.output
/set message test on
/set message auto off
/clear all

--S 1 of 25
a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
--R
//--R 2
//--R (1) 3y x + 2y x + y + 9x + 5x - 3
//--R Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 1

--S 2 of 25
a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
--R
//--R 3
//--R (2) 2y x - y x - y + 6x - 2x - 3x + 3
//--R Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 2

--S 3 of 25
a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
--R
//--R 3
//--R (3) y x + y x + 3x + 2x
//--R Type: DistributedMultivariatePolynomial([y,x],Integer)
--E 3
\[
an := \{a_1, a_2, a_3\}\\
(4)\\
\begin{align*}
&[3y^2 x + 2y x^2 + y^2 + 9x + 5x^2 - 3, \quad 2y^2 x - y^2 x - y + 6x^2 - 2x - 3x^2 + 3, \\
&3y^2 - 2y x^2 + 3x - 2]\\
\text{Type: List(DistributedMultivariatePolynomial([y, x], Integer))}
\end{align*}
\]

\[
\text{euclideanGroebner}(\text{an})\\
(5)\\
\begin{align*}
&[y x - y + x + 3, 2y^2 + 2x - 3x - 6, 2x^2 - 5x - 5x]\\
\text{Type: List(DistributedMultivariatePolynomial([y, x], Integer))}
\end{align*}
\]

\[
\text{euclideanGroebner}(\text{an}, \text{"redcrit"})\\
\begin{align*}
&\text{reduced Critpair - Polynom} : \\
&2y x^2 - y x^2 - y - 6x^2 - 3x + 3\\
\end{align*}
\]
--R
--R
--R 3 2
--R - 4x + 10x + 10x
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2
--R 2y + 2x - 3x - 6
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 2x + 5x + 5x
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R
--R
THE GROEBNER BASIS over EUCLIDEAN DOMAIN

\[ (6) \quad [y \cdot x - y \cdot x + 3, 2y + 2x - 3x - 6, 2y - 5x] \]

Type: List(DistributedMultivariatePolynomial([y,x],Integer))
There are

Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

(7) \[y x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x\]

Type: List(DistributedMultivariatePolynomial([y, x], Integer))

you choose option -info-

abbrev. for the following information strings are
--R  ci => Leading monomial for critpair calculation
--R  tci => Number of terms of polynomial i
--R  cj => Leading monomial for critpair calculation
--R  tcj => Number of terms of polynomial j
--R  c  => Leading monomial of critpair polynomial
--R  tc  => Number of terms of critpair polynomial
--R  rc => Leading monomial of redcritpair polynomial
--R  trc => Number of terms of redcritpair polynomial
--R  tF => Number of polynomials in reduction list F
--R  tD => Number of critpairs still to do

--R
--R

--R  3 3 2 2
--R  [ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]
--R
--R
--R  reduced Critpair - Polynom :
--R
--R  y x - y + x + 3
--R
--R

--R  2 2
--R  [ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x ,tc= 4,rc= y x ,trc= 4,tH= 1,tD= 3]
--R
--R
--R  reduced Critpair - Polynom :
--R
--R

--R  2
--R  4y + 4x - 6x - 12
--R
--R

--R  2
--R  [ci= y x ,tci= 6,cj= y x ,tcj= 4,c= y x ,tc= 5,rc= y ,trc= 4,tH= 2,tD= 3]
--R
--R
--R
--R  reduced Critpair - Polynom :
--R
--R

--R  3 2
--R  - 4x + 10x + 10x
reduced Critpair - Polynom :

2
2y + 2x - 3x - 6

reduced Critpair - Polynom :

2
2
[[ci= y x, tci= 6, cj= y x, tcj= 4, c= y x, tc= 5, rc= y, trc= 4, tH= 3, tD= 4]]

reduced Critpair - Polynom :

0

reduced Critpair - Polynom :

[[ci= y, tci= 4, cj= y, tcj= 4, c= 0, tc= 0, rc= 0, trc= 0, tH= 3, tD= 3]]

reduced Critpair - Polynom :

3
3
[[ci= y x, tci= 4, cj= y, tcj= 4, c= y, tc= 5, rc= x, trc= 3, tH= 3, tD= 3]]
reduced Critpair - Polynom :

3 3
[[ci = x , tci = 3, cj = x , tcj = 3, c = 0, tcr = 0, trc = 0, th = 3, td = 2]]

reduced Critpair - Polynom :

3 2
[[ci = y x , tci = 4, cj = y x, tcj = 4, c = y x , tc = 3, tcr = 0, trc = 0, th = 3, td = 1]]

reduced Critpair - Polynom :

3 2
[[ci = y, tci = 4, cj = x , tcj = 3, c = y x , tc = 5, tcr = 0, trc = 0, th = 3, td = 0]]

There are

3
Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

(8) \[ y x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x \]

Type: List(DistributedMultivariatePolynomial([y, x], Integer))

b1: HDMP([y, x], INT) := (9*x**2 + 5*x - 3) + y*(3*x**2 + 2*x + 1)

(9) \[ 3y x + 2y x + 9x + y + 5x - 3 \]

Type: HomogeneousDistributedMultivariatePolynomial([y, x], Integer)
b2: HDMP([y,x], INT) := (6*x**3 - 2*x**2 - 3*x + 3) + y*(2*x**3 - x - 1)

b3: HDMP([y,x], INT) := (3*x**3 + 2*x**2) + y*(x**3 + x**2)

bn := [b1, b2, b3]

euclideanGroebner(bn)

euclideanGroebner(bn, "redcrit")
reduced Critpair - Polynom :
 2
 4x + 4y - 6x - 12
 2
 6x + 2y - 3x - 6
 0
 2
 2y + 5y + 8x + 3
 0
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R 2 2
--R (14) [2y - 5y - 8x - 3, y x - y + x + 3, 2x + 2y - 3x - 6]
--R Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))
--E 14

--S 15 of 25
euclideanGroebner(bn,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
--R rc => Leading monomial of redcritpair polynomial
--R trc => Number of terms of redcritpair polynomial
--R tF => Number of polynomials in reduction list F
--R tD => Number of critpairs still to do
--R
--R
--R
--R 3 3 2 2
--R [[ci= y x , tci= 7, cj= y x , tcj= 4, c= y x , tc= 6, rc= y x, trc= 6, tH= 3, tD= 3]]
--R
--R
--R 2 2
--R [[ci= y x , tci= 6, cj= y x , tcj= 6, c= y x , tc= 4, rc= y x, trc= 4, tH= 1, tD= 3]]
--R
--R
--R 2
--R [[ci= y x , tci= 6, cj= y x , tcj= 4, c= y x, tc= 5, rc= x, trc= 4, tH= 2, tD= 3]]
--R
--R
--R 2
--R [[ci= y x , tci= 6, cj= y x, tcj= 4, c= y x, tc= 5, rc= x, trc= 4, tH= 2, tD= 3]]
There are 3 Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

\[ 2 \]
\[ (15) \quad [2y - 5y - 8x - 3, y x - y + x + 3, 2x + 2y - 3x - 6] \]
Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))

reduced Critpair - Polynom:

\[ 2 \]
\[ (15) \quad [-2y x - y x - 6x - y - 3x + 3] \]

you choose option -info-

abbrev. for the following information strings are

- ci => Leading monomial for critpair calculation
- tci => Number of terms of polynomial i
- cj => Leading monomial for critpair calculation
- tcj => Number of terms of polynomial j
- c => Leading monomial of critpair polynomial
- tc => Number of terms of critpair polynomial
---R rc => Leading monomial of redcritpair polynomial
---R trc => Number of terms of redcritpair polynomial
---R tF => Number of polynomials in reduction list F
---R tD => Number of critpairs still to do
---R
---R
---R
---R
---R
---R
---R
---R 3 3 2 2
---R [[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]
---R
---R
---R reduced Critpair - Polynom :
---R
---R y x - y + x + 3
---R
---R
---R
---R
---R 2 2
---R [[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x ,tc= 4,rc= y x ,trc= 4,tH= 1,tD= 3]]
---R
---R
---R reduced Critpair - Polynom :
---R
---R
---R
---R
---R
---R
---R 2
---R 4x + 4y - 6x - 12
---R
---R
---R
---R
---R
---R 2
---R [[ci= y x ,tci= 6,cj= y x ,tcj= 4,c= y x ,tc= 5,rc= x ,trc= 4,tH= 2,tD= 3]]
---R
---R
---R reduced Critpair - Polynom :
---R
---R
---R
---R
---R
---R
---R 2
---R 2x + 2y - 3x - 6
---R
---R
---R
---R
---R
---R 2
---R [[ci= y x ,tci= 6,cj= y x ,tcj= 4,c= y x ,tc= 5,rc= x ,trc= 4,tH= 2,tD= 3]]
---R
reduced Critpair - Polynom:

\[- \frac{2}{2} (x^2) - 2y + 5y + 8x + 3\]

reduced Critpair - Polynom:

\[- \frac{2}{2} (x^2) + 5y + 8x + 3\]

reduced Critpair - Polynom:

\[- \frac{2}{2} (x^2) + 5y + 8x + 3\]

reduced Critpair - Polynom:

\[3 \frac{2}{2} (x^2) \]
There are 3 Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

\begin{align*}
\text{(16)} & & [2y - 5y - 8x - 3, y\ x - y + x + 3, 2x + 2y - 3x - 6] \\
\text{Type: List(HomogeneousDistributedMultivariatePolynomial([y,x],Integer))}
\end{align*}

\begin{align*}
c1: \text{GDMP([y,x],INT,DIRPROD(2,NNI))} & :& (9\ x^2 + 5\ x - 3) + y(3\ x^2 + 2\ x + 1) \\
\text{Type: GeneralDistributedMultivariatePolynomial([y,x],Integer,...)
\end{align*}

\begin{align*}
c2: \text{GDMP([y,x],INT,DIRPROD(2,NNI))} & :& (6\ x^3 - 2\ x^2 - 3\ x + 3) + y(2\ x^3 - x - 1) \\
\text{Type: GeneralDistributedMultivariatePolynomial([y,x],Integer,...)
\end{align*}

\begin{align*}
c3: \text{GDMP([y,x],INT,DIRPROD(2,NNI))} & :& (3\ x^3 + 2\ x^2) + y(x^3 + x^2) \\
\text{Type: GeneralDistributedMultivariatePolynomial([y,x],Integer,...)
\end{align*}

\begin{align*}
cn:=\{c1,c2,c3\} & :& [3y \ x + 2y \ x + y + 9x + 5x - 3, 2y \ x - y \ x - y + 6x - 2x - 3x + 3, \\
& & 3y \ x + y \ x + 3x + 2x ] \\
\text{Type: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...)
\end{align*}
euclideanGroebner(cn)
--R
--R
--R 2 3 2
--R (21) \([y \ x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x]\)
--RType: List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...
--E 21

--S 22 of 25
euclideanGroebner(cn,"redcrit")
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 2 2
--R - 2y x - y x - y - 6x - 3x + 3
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R y x - y + x + 3
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R 2
--R 4y + 4x - 6x - 12
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R
--R
--R
--R 2
--R 2\,y + 2\,x - 3\,x - 6
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 2x + 5x + 5x
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R
--R
--R
--R
--R
--R THE GROEBNER BASIS over EUCLIDEAN DOMAIN
--R
--R (22) [y\,x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x]
--R Type: List(GeneralDistributedMultivariatePolynomial([y,x], Integer,...
--E 22
euclideanGroebner(cn,"info")

you choose option -info-

abbrev. for the following information strings are

ci => Leading monomial for critpair calculation

tci => Number of terms of polynomial i

cj => Leading monomial for critpair calculation

tcj => Number of terms of polynomial j

c => Leading monomial of critpair polynomial

tc => Number of terms of critpair polynomial

cj => Leading monomial of redcritpair polynomial

trc => Number of terms of redcritpair polynomial

tF => Number of polynomials in reduction list F

tD => Number of critpairs still to do

[[ci= y x ,tci= 7,cj= y x ,tcj= 4,c= y x ,tc= 6,rc= y x ,trc= 6,tH= 3,tD= 3]]

[[ci= y x ,tci= 6,cj= y x ,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 1,tD= 3]]

[[ci= y x ,tci= 6,cj= y x,tcj= 6,c= y x,tc= 4,rc= y x,trc= 4,tH= 2,tD= 3]]

[[ci= y x ,tci= 6,cj= y x,tcj= 6,c= y x,tc= 5,rc= y x,trc= 3,tH= 3,tD= 3]]

[[ci= y x ,tci= 6,cj= y x,tcj= 6,c= y x,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]

[[ci= y ,tci= 4,cj= y ,tcj= 4,c= y ,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]

[[ci= y ,tci= 4,cj= y ,tcj= 4,c= y ,tc= 5,rc= x ,trc= 3,tH= 3,tD= 3]]

[[ci= x ,tci= 3,cj= x ,tcj= 3,c= 0,tc= 0,rc= 0,trc= 0,tH= 3,tD= 3]]
There are 3 Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

\( \begin{align*}
\text{Type: } & \text{List(GeneralDistributedMultivariatePolynomial([y,x],Integer,...) of 23 of 25) } \\
\text{euclideanGroebner(cn,"info","redcrit")}
\end{align*} \)

reduced Critpair - Polynom :

\[-2y x - y x - y - 6x - 3x + 3\]

you choose option -info-

abbrev. for the following information strings are

- ci => Leading monomial for critpair calculation
- tci => Number of terms of polynomial i
- cj => Leading monomial for critpair calculation
- tcj => Number of terms of polynomial j
- c => Leading monomial of critpair polynomial
- tc => Number of terms of critpair polynomial
- rc => Leading monomial of redcritpair polynomial
- trc => Number of terms of redcritpair polynomial
- tF => Number of polynomials in reduction list F
- tD => Number of critpairs still to do
reduced Critpair - Polynom:

\[ 3 \quad 3 \quad 2 \quad 2 \]

\[ [[ci= y \ x , tci= 7, cj= y \ x , tcj= 4, c= y \ x , tc= 6, rc= y \ x , trc= 6, tH= 3, tD= 3]] \]

reduced Critpair - Polynom:

\[ y \ x - y + x + 3 \]

reduced Critpair - Polynom:

\[ 2 \quad 2 \]

\[ [[ci= y \ x , tci= 6, cj= y \ x , tcj= 6, c= y \ x , tc= 4, rc= y \ x , trc= 4, tH= 1, tD= 3]] \]

reduced Critpair - Polynom:

\[ 2 \]

\[ 4y + 4x - 6x - 12 \]

reduced Critpair - Polynom:

\[ 2 \]

\[ [[ci= y \ x , tci= 6, cj= y \ x , tcj= 6, c= y \ x , tc= 5, rc= y , trc= 4, tH= 2, tD= 3]] \]

reduced Critpair - Polynom:

\[ 3 \quad 2 \]

\[ - 4x + 10x + 10x \]

reduced Critpair - Polynom:

\[ 3 \]

\[ [[ci= y \ x , tci= 4, cj= y , tcj= 4, c= y , tc= 5, rc= x , trc= 3, tH= 3, tD= 3]] \]

reduced Critpair - Polynom:

\[ 2 \]
--R 2y + 2x - 3x - 6
--R
--R
--R
--R 2
--R [[ci= y x , tci= 6, cj= y x, tcj= 4, c= y x, tc= 5, rc= y, trc= 4, tH= 3, tD= 4]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R [[ci= y, tci= 4, cj= y, tcj= 4, c= y x, tc= 5, rc= y, trc= 4, tH= 3, tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 3 2
--R - 2x + 5x + 5x
--R
--R
--R
--R
--R [ci= y x, tci= 4, cj= y, tcj= 4, c= y x, tc= 5, rc= y, trc= 3, tH= 3, tD= 3]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
--R
--R
--R 3 3
--R [[ci= x, tci= 3, cj= x, tcj= 3, c= 0, tc= 0, rc= 0, trc= 0, tH= 3, tD= 2]]
--R
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 0
--R
reduced Critpair - Polynom :

There are

Groebner Basis Polynomials.

THE GROEBNER BASIS over EUCLIDEAN DOMAIN

(24) \[ y x - y + x + 3, 2y + 2x - 3x - 6, 2x - 5x - 5x \]

Operations

---E 25

)spool
)lisp (bye)
EuclideanGroebnerBasisPackage computes groebner bases for polynomial ideals over euclidean domains. The basic computation provides a distinguished set of generators for these ideals. This basis allows an easy test for membership: the operation euclideanNormalForm returns zero on ideal members. The string "info" and "redcrit" can be given as additional args to provide incremental information during the computation.

If "info" is given, a computational summary is given for each s-polynomial. If "redcrit" is given, the reduced critical pairs are printed.

The term ordering is determined by the polynomial type used.
Suggested types include

* DistributedMultivariatePolynomial
* HomogeneousDistributedMultivariatePolynomial
* GeneralDistributedMultivariatePolynomial

Example to call euclideanGroebner:

```plaintext
a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
an:=[a1,a2,a3]
euclideanGroebner(an)
```

This will return the weak euclidean Groebner basis set.
All reductions are total reductions.

You can get more information by providing a second argument.
To get the reduced critical pairs do:

```plaintext
euclideanGroebner(an,"redcrit")
```

You can get other information by calling:

```plaintext
euclideanGroebner(an,"info")
```

which returns:

- \( ci \Rightarrow \) Leading monomial for critpair calculation
- \( tci \Rightarrow \) Number of terms of polynomial i
- \( cj \Rightarrow \) Leading monomial for critpair calculation
- \( tcj \Rightarrow \) Number of terms of polynomial j
c => Leading monomial of critpair polynomial
tc => Number of terms of critpair polynomial
rc => Leading monomial of redcritpair polynomial
trc => Number of terms of redcritpair polynomial
tH => Number of polynomials in reduction list H
tD => Number of critpairs still to do

The three argument form returns all of the information:

\[ \text{euclideanGroebner}(\text{an}, \text{"info"}, \text{"redcrit"}) \]

The term ordering is determined by the polynomial type used.
Suggested types include
- DistributedMultivariatePolynomial
- HomogeneousDistributedMultivariatePolynomial
- GeneralDistributedMultivariatePolynomial

See Also:
- \text{\texttt{)}display\texttt{operations euclideanGroebner}}
- \text{\texttt{)}show\texttt{EuclideanGroebnerBasisPackage}}
- \text{\texttt{)}show\texttt{DistributedMultivariatePolynomial}}
- \text{\texttt{)}show\texttt{HomogeneousDistributedMultivariatePolynomial}}
- \text{\texttt{)}show\texttt{GeneralDistributedMultivariatePolynomial}}
- \text{\texttt{)}show\texttt{GroebnerPackage}}

---

**EuclideanGroebnerBasisPackage (GBEUCLID)**

Exports:
- euclideanGroebner
- euclideanNormalForm
--- package GBEUCLID EuclideanGroebnerBasisPackage ---

)abbrev package GBEUCLID EuclideanGroebnerBasisPackage
++ Authors: Gebauer, Moeller
++ Date Created: 12-1-86
++ Date Last Updated: 2-28-91
++ Description:
++ \spadtype{EuclideanGroebnerBasisPackage} computes groebner
++ bases for polynomial ideals over euclidean domains.
++ The basic computation provides
++ a distinguished set of generators for these ideals.
++ This basis allows an easy test for membership: the operation
++ "info" and "redcrit" can be given as additional args to provide
++ incremental information during the computation. If "info" is given,
++ a computational summary is given for each s-polynomial. If "redcrit"
++ is given, the reduced critical pairs are printed. The term ordering
++ is determined by the polynomial type used. Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

EuclideanGroebnerBasisPackage(Dom, Expon, VarSet, Dpol): T == C where

Dom: EuclideanDomain
Expon: OrderedAbelianMonoidSup
VarSet: OrderedSet
Dpol: PolynomialCategory(Dom, Expon, VarSet)

T== with

   euclideanNormalForm: (Dpol, List(Dpol) ) -> Dpol
   ++ euclideanNormalForm(poly,gb) reduces the polynomial poly modulo the
   ++ precomputed groebner basis gb giving a canonical representative
   ++ of the residue class.
   euclideanGroebner: List(Dpol) -> List(Dpol)
   ++ euclideanGroebner(lp) computes a groebner basis for a polynomial
   ++ ideal over a euclidean domain generated by the list of polys lp.
   ++
   ++X a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
   ++X a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
   ++X a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
   ++X an:=[a1,a2,a3]
   ++X euclideanGroebner(an)

   euclideanGroebner: (List(Dpol), String) -> List(Dpol)
   ++ euclideanGroebner(lp, infoflag) computes a groebner basis
   ++ for a polynomial ideal over a euclidean domain
   ++ generated by the list of polynomials lp.
   ++ During computation, additional information is printed out
if infoflag is given as
either "info" (for summary information) or
"redcrit" (for reduced critical pairs)

X a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
X a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
X a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
X an:=[a1,a2,a3]

euclideanGroebner(an,"redcrit")
euclideanGroebner(an,"info")

euclideanGroebner: (List(Dpol), String, String ) -> List(Dpol)
euclideanGroebner(lp, "info", "redcrit") computes a groebner basis
for a polynomial ideal generated by the list of polynomials lp.
If the second argument is "info",
a summary is given of the critical pairs.
If the third argument is "redcrit", critical pairs are printed.

X a1:DMP([y,x],INT):= (9*x**2 + 5*x - 3)+ y*(3*x**2 + 2*x + 1)
X a2:DMP([y,x],INT):= (6*x**3 - 2*x**2 - 3*x +3) + y*(2*x**3 - x - 1)
X a3:DMP([y,x],INT):= (3*x**3 + 2*x**2) + y*(x**3 + x**2)
an:=[a1,a2,a3]
euclideanGroebner(an,"info","redcrit")

C== add
Ex ==> OutputForm
lc ==> leadingCoefficient
red ==> reductum

import OutputForm

------ Definition list of critPair
------ lcmfij is now lcm of headterm of poli and polj
------ lcmcij is now lcm of of lc poli and lc polj

critPair ==>Record(lcmfij: Expon, lcmcij: Dom, poli:Dpol, polj: Dpol )
Prinp == Record( ci:Dpol,tcj:Integer,cj:Dpol,tcj:Integer,c:Dpol,tc:Integer,rc:Dpol,trc:Integer,tH:Integer,tD:Integer)

------ Definition of intermediate functions

strongGbasis: (List(Dpol), Integer, Integer) -> List(Dpol)
eminGbasis: List(Dpol) -> List(Dpol)
ecritT: (critPair ) -> Boolean
ecritM: (Expon, Dom, Expon, Dom) -> Boolean
ecritB: (Expon, Dom, Expon, Dom, Expon, Dom) -> Boolean
ecritinH: (Dpol, List(Dpol)) -> Boolean
ecritBonD: (Dpol, List(critPair)) -> List(critPair)
ecritMTondd1:(List(critPair)) -> List(critPair)
ecritMondd1:(Expon, Dom, List(critPair)) -> List(critPair)
crithdelH: (Dpol, List(Dpol)) -> List(Dpol)
eupdatF: (Dpol, List(Dpol) ) -> List(Dpol)
updatH: (Dpol, List(Dpol), List(Dpol)) -> List(Dpol)
sortin: (Dpol, List(Dpol) ) -> List(Dpol)
eRed: (Dpol, List(Dpol)) -> Dpol
esPol: (Dpol, List(Dpol)) -> Dpol
esPol: (critPair) -> Dpol
updatD: (List(critPair), List(critPair)) -> List(critPair)
lepol: Dpol -> Integer
prinshINFO : Dpol -> Void
prinINFO: (critPair, Dpol, Dpol,Integer,Integer,Integer) -> Integer
prinpolINFO: List(Dpol) -> Void
prinb: Integer -> Void

------ MAIN ALGORITHM GROEBNER ------------------------
euclideanGroebner( Pol: List(Dpol) ) ==
  eminGbasis(strongGbasis(Pol,0,0))
euclideanGroebner( Pol: List(Dpol), xx1: String) ==
  xx1 = "redcrit" =>
    eminGbasis(strongGbasis(Pol,1,0))
  xx1 = "info" =>
    eminGbasis(strongGbasis(Pol,2,1))
  print(" "::Ex)
  print("WARNING: options are - redcrit and/or info - "::Ex)
  print("you didn’t type them correct"::Ex)
  print("please try again":Ex)
  print(" "::Ex)
  []
euclideanGroebner( Pol: List(Dpol), xx1: String, xx2: String) ==
  (xx1 = "redcrit" and xx2 = "info") or
  (xx1 = "info" and xx2 = "redcrit") =>
    eminGbasis(strongGbasis(Pol,1,1))
  xx1 = "redcrit" and xx2 = "redcrit" =>
    eminGbasis(strongGbasis(Pol,1,0))
  xx1 = "info" and xx2 = "info" =>
    eminGbasis(strongGbasis(Pol,2,1))
  print(" "::Ex)
  print("WARNING: options are - redcrit and/or info - "::Ex)
  print("you didn’t type them correct"::Ex)
  print("please try again":Ex)
  print(" "::Ex)
  []

------ calculate basis

strongGbasis(Pol: List(Dpol),xx1: Integer, xx2: Integer ) ==
  ddi, D : List(critPair)
--------- create D and Pol

Pol1:= sort((z1:Dpol,z2:Dpol):Boolean +-> (degree z1 > degree z2) or
          ((degree z1 = degree z2 ) and
            sizeLess?(leadingCoefficient z2,leadingCoefficient z1)),
          Pol)
Pol:= [first(Pol1)]
H:= Pol
Pol1:= rest(Pol1)
D:= nil
while ~null Pol1 repeat
  h:= first(Pol1)
  Pol1:= rest(Pol1)
  en:= degree(h)
  lch:= lc h
  dd1:=
    [sup(degree(x), en), lcm(leadingCoefficient x, lch), x, h]$critPair
    for x in Pol
  D:= updatD(
    ecritMtondd1(
      sort(
        (z1:critPair,z2:critPair):Boolean+->
          (z1.lcmfij < z2.lcmfij) or
          (( z1.lcmfij = z2.lcmfij ) and
            ( sizeLess?(z1.lcmciij,z2.lcmciij)) ), dd1)),
        ecritBonD(h,D))
  Pol:= cons(h, eupdatF(h, Pol))
((en = degree(first(H))) and
 (leadingCoefficient(h) = leadingCoefficient(first(H)) ) ) =>
  " go to top of while 
H:= updatH(h,H,crithdelH(h,H),[h])
H:= sort((z1,z2) +-> (degree z1 > degree z2) or
          ((degree z1 = degree z2 ) and
            sizeLess?(leadingCoefficient z2,leadingCoefficient z1)), H)
D:= sort((z1,z2) +-> (z1.lcmfij < z2.lcmfij) or
          (( z1.lcmfij = z2.lcmfij ) and
            ( sizeLess?(z1.lcmciij,z2.lcmciij)) ) ,D)
xx:= xx2
-------- loop

while ~null D repeat
  DO:= first D
  ep:=esPol(D0)
  D:= rest(D)
  eh:= ecredPol(eRed(ep,H,H),H)
  if xx1 = 1 then
    prinshINFO(eh)
  eh = 0 =>
    if xx2 = 1 then
ala:= prindINFO(D0, ep, eh, #H, #D, xx)
xx:= 2

" go to top of while "

eh := unitCanonical eh
e:= degree(eh)
leh:= lc eh
dd1:=
[[sup(degree(x), e), lcm(leadingCoefficient x, leh), x, eh]$critPair
for x in Pol]

D:= updatD(
ecritMTondd1(
sort((z1,z2) +-> (z1.lcmfij < z2.lcmfij) or
(( z1.lcmfij = z2.lcmfij ) and
( sizeLess?(z1.lcmcij,z2.lcmcij) ) ), dd1)),
ecritBonD(eh,D))

Pol:= cons(eh,eupdatF(eh,Pol))
^ecrithinH(eh,H) or
((e = degree(first(H))) and
(leadingCoefficient(eh) = leadingCoefficient(first(H)) ) ) =>
if xx2 = 1 then
ala:= prindINFO(D0, ep, eh, #H, #D, xx)
xx:= 2

" go to top of while "

H:= updatH(eh,H,crithdelH(eh,H),[eh])
H:= sort((z1,z2)+-> (degree z1 > degree z2) or
((degree z1 = degree z2 ) and
sizeLess?(leadingCoefficient z2,leadingCoefficient z1)), H)
if xx2 = 1 then
ala:= prindINFO(D0, ep, eh, #H, #D, xx)
xx:= 2

" go to top of while "

if xx2 = 1 then
prinpolINFO(Pol)
print(" THE GROEBNER BASIS over EUCLIDEAN DOMAIN":::Ex)
if xx1 = 1 and xx2 ^= 1 then
print(" THE GROEBNER BASIS over EUCLIDEAN DOMAIN":::Ex)

H

---------------------------

--- erase multiple of e in D2 using crit M

ecritMondd1(e: Expon, c: Dom, D2: List(critPair)) ==
null D2 => nil
x:= first(D2)
ecritM(e,c, x.lcmfij, lcm(leadingCoefficient(x.poli),
leadingCoefficient(x.poli)))
=> ecritMondd1(e, c, rest(D2))
cons(x, ecritMondd1(e, c, rest(D2)))
ecredPol(h: Dpol, F: List(Dpol)) ==
  h0 := 0
  null F => h
  while h ^= 0 repeat
    h0 := h0 + monomial(leadingCoefficient(h), degree(h))
    h := eRed(red(h), F, F)
  h0

--- reduce dd1 using crit T and crit M
ecritMTondd1(dd1: List(critPair)) ==
  null dd1 => nil
  f1 := first(dd1)
  s1 := #(dd1)
  cT1 := ecritT(f1)
  s1 = 1 and cT1 => nil
  s1 = 1 => dd1
  e1 := f1.lcmfij
  r1 := rest(dd1)
  e2 := first(r1)
  e1 = f2.lcmfij and f1.lcmcij = f2.lcmcij =>
    cT1 => ecritMTondd1(cons(f1, rest(r1)))
    ecritMTondd1(r1)
  dd1 := ecritMondd1(e1, f1.lcmcij, r1)
  cT1 => ecritMTondd1(dd1)
  cons(f1, ecritMTondd1(dd1))

--- erase elements in D fullfilling crit B
ecritBonD(h: Dpol, D: List(critPair)) ==
  null D => nil
  x := first(D)
  x1 := x.poli
  x2 := x.polj
  ecritB(degree(h), leadingCoefficient(h),
         degree(x1), leadingCoefficient(x1),
         degree(x2), leadingCoefficient(x2)) =>
    ecritBonD(h, rest(D))
  cons(x, ecritBonD(h, rest(D)))

--- concat F and h and erase multiples of h in F
eupdatF(h: Dpol, F: List(Dpol)) ==
null F => nil
f1:= first(F)
ecritM(degree h, leadingCoefficient(h), degree f1, leadingCoefficient(f1))
    => eupdatF(h, rest(F))
cons(f1, eupdatF(h, rest(F)))

-------------------------------
--- concat H and h and erase multiples of h in H
updatH(h: Dpol, H: List(Dpol), Hh: List(Dpol), Hhh: List(Dpol)) ==
null H => append(Hh, Hhh)
h1:= first(H)
hlcm:= sup(degree(h1), degree(h))
plc:= extendedEuclidean(leadingCoefficient(h), leadingCoefficient(h1))
hp:= monomial(plc.coef1, subtractIfCan(hlcm, degree(h))::Expon) * h +
    monomial(plc.coef2, subtractIfCan(hlcm, degree(h1))::Expon) * h1
(ecritHinHp(hp, Hh) and ecrithHinHp(hp, Hhh)) =>
    hpp:= append(rest(H), Hh)
    hp:= ecredPol(eRed(hp, hpp, hpp), hpp)
updatH(h, rest(H), crithdelH(hp, Hh), cons(hp, crithdelH(hp, Hhh)))
updatH(h, rest(H), Hh, Hhh)

--------------------------------------------------
---- delete elements in cons(h, H)
crithdelH(h: Dpol, H: List(Dpol)) ==
null H => nil
h1:= first(H)
dh1:= degree h1
dh:= degree h
ecritM(dh, lc h, dh1, lc h1) => crithdelH(h, rest(H))
dh1 = sup(dh, dh1) =>
    plc:= extendedEuclidean( lc h1, lc h)
    cons(plc.coef1 * h1 + monomial(plc.coef2, subtractIfCan(dh1, dh)::Expon) * h,
        crithdelH(h, rest(H)))
    cons(h1, crithdelH(h, rest(H)))
eminGbasis(F: List(Dpol)) ==
null F => nil
newbas := eminGbasis rest F
cons(ecredPol( first(F), newbas), newbas)

---------------------------------------------
--- does h belong to H
ecrithinH(h: Dpol, H: List(Dpol)) ==
null H => true
h1:= first(H)
ecritM(degree h1, lc h1, degree h, lc h) => false
ecrithinH(h, rest(H))
--- calculate euclidean S-polynomial of a critical pair

esPol(p: critPair) ==
  Tij := p.lcmfij
  fi := p.poli
  fj := p.polj
  lij := lcm(leadingCoefficient(fi), leadingCoefficient(fj))
  red(fi)*monomial((lij exquo leadingCoefficient(fi))::Dom,
                   subtractIfCan(Tij, degree fi)::Expon) -
  red(fj)*monomial((lij exquo leadingCoefficient(fj))::Dom,
                   subtractIfCan(Tij, degree fj)::Expon)

--- euclidean reduction mod F

eRed(s: Dpol, H: List(Dpol), Hh: List(Dpol)) ==
  ( s = 0 or null H ) => s
  f1 := first(H)
  ds := degree s
  lf1 := leadingCoefficient(f1)
  ls := leadingCoefficient(s)
  e := Union(Expon, "failed")
  (((e := subtractIfCan(ds, degree f1)) case "failed" ) or sizeLess?(ls, lf1) ) =>
    eRed(s, rest(H), Hh)
  sdf1 := divide(ls, lf1)
  q1 := sdf1.quotient
  sdf1.remainder = 0 =>
    eRed(red(s) - monomial(q1,e)*reductum(f1), Hh, Hh)
  eRed(s -(monomial(q1, e)*f1), rest(H), Hh)

--- crit T true, if e1 and e2 are disjoint

ecritT(p: critPair) ==
  pi := p.poli
  pj := p.polj
  ci := lc pi
  cj := lc pj
  (p.lcmfij = degree pi + degree pj) and (p.lcmcij = ci*cj)

--- crit M - true, if lcm#2 multiple of lcm#1

ecritM(e1: Expon, c1: Dom, e2: Expon, c2: Dom) ==
  en: Union(Expon, "failed")
\((\text{en:=subtractIfCan(e2, e1)} \text{ case "failed"}) \text{ or} \) 
\((\text{c2 exquo ci} \text{ case "failed"}) \Rightarrow \text{false} \) 
\text{true} 

------------------------

--- crit B - true, if eik is a multiple of eh and eik =equal \(\text{lcm(eh,ei)}\) and eik =equal \(\text{lcm(eh,ek)}\)

\text{ecritB(eh:Expon, ch: Dom, ei:Expon, ci: Dom, ek:Expon, ck: Dom)} ==
\text{eik:= sup(ei, ek)} 
\text{cik:= lcm(ci, ck)}
\text{ecritM(eh, ch, eik, cik) and}
\text{^ecritM(eik, cik, sup(ei, eh), lcm(ci, ch)) and}
\text{^ecritM(eik, cik, sup(ek, eh), lcm(ck, ch))}

------------------------

--- reduce p1 mod lp

\text{euclideanNormalForm(p1: Dpol, lp: List(Dpol)) ==}
\text{eRed(p1, lp, lp)}

------------------------

--- insert element in sorted list

\text{sortin(p1: Dpol, lp: List(Dpol))==}
null lp => \[p1\]
f1:= first(lp)
elf1:= degree(f1)
ep1:= degree(p1)
\((\text{elf1 < ep1}) \text{ or } ((\text{elf1 = ep1}) \text{ and})
\text{sizeLess?(leadingCoefficient(f1),leadingCoefficient(p1)))} \Rightarrow
\text{cons(f1,sortin(p1, rest(lp))})
\text{cons(p1,lp)}

\text{updatD(D1: List(critPair), D2: List(critPair)) ==}
null D1 => D2
null D2 => D1
dl1:= first(D1)
dl2:= first(D2)
\((\text{d11.lcmfij < d12.lcmfij}) \Rightarrow \text{cons(d11, updatD(D1.rest, D2))}
\text{cons(d12, updatD(D1, D2.rest))}

---- calculate number of terms of polynomial

\text{lepol(p1:Dpol)==}
n: Integer
n:= 0
\text{while p1 ~= 0 repeat}
n := n + 1
p1 := red(p1)
n
---- print blank lines

prinb(n: Integer) ==
for i in 1..n repeat messagePrint(" ")

---- print reduced critpair polynom

prinshINFO(h: Dpol) ==
prinb(2)
messagePrint(" reduced Critpair - Polynom :")
prinb(2)
print(h::Ex)
prinb(2)

-------------------------------

---- print info string

prindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1: Integer, i2: Integer, n: Integer) ==
ll: List Prinp
a: Dom
cpi := cp.poli
cpj := cp.polj
if n = 1 then
prinb(1)
messagePrint("you choose option -info- ")
messagePrint("abbrev. for the following information strings are")
messagePrint(" ci => Leading monomial for critpair calculation")
messagePrint(" tc i => Number of terms of polynomial i")
messagePrint(" cj => Leading monomial for critpair calculation")
messagePrint(" tc j => Number of terms of polynomial j")
messagePrint(" c => Leading monomial of critpair polynomial")
messagePrint(" tc => Number of terms of critpair polynomial")
messagePrint(" rc => Leading monomial of redcritpair polynomial")
messagePrint(" trc => Number of terms of redcritpair polynomial")
messagePrint(" tF => Number of polynomials in reduction list F")
messagePrint(" tD => Number of critpairs still to do")
prinb(4)
n := 2
prinb(1)
a := 1
ph = 0 =>
ps = 0 =>
ll := [[monomial(a, degree(cpi)), lepol(cpi), monomial(a, degree(cpj)), lepol(cpj), ps, 0, ph, 0, i1, i2]]
print(ll::Ex)
prinb(1)
n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
lepol(ps), ph,0,i1,i2]$Prinp]
print(ll::Ex)
prinb(1)
n
ll:= [[monomial(a,degree(cpi)),lepol(cpi),
monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),
lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2]$Prinp]
print(ll::Ex)
prinb(1)

-----------------------------

---- print the groebner basis polynomials

prinpolINFO(pl: List(Dpol))==
n:Integer
n:= #pl
prinb(1)
n = 1 =>
  print(" There is 1 Groebner Basis Polynomial "::Ex)
  prinb(2)
print(" There are "::Ex)
prinb(1)
print(n::Ex)
prinb(1)
print(" Groebner Basis Polynomials. "::Ex)
prinb(2)
package EVALCYC EvaluateCycleIndicators

— EvaluateCycleIndicators.input —

)set break resume
)sys rm -f EvaluateCycleIndicators.output
)spool EvaluateCycleIndicators.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show EvaluateCycleIndicators
--R
--R EvaluateCycleIndicators(F: Algebra(Fraction(Integer))) is a package constructor
--R Abbreviation for EvaluateCycleIndicators is EVALCYC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EVALCYC
--R
--R------------------------------- Operations --------------------------------
--R eval : ((Integer -> F),SymmetricPolynomial(Fraction(Integer))) -> F
--R
--E 1

)spool
)lisp (bye)

——

— EvaluateCycleIndicators.help —

====================================================================
EvaluateCycleIndicators examples
====================================================================

This package is to be used in conjuction with the CycleIndicators package.
It provides an evaluation function for SymmetricPolynomials.

See Also:
  o )show EvaluateCycleIndicators
EvaluateCycleIndicators (EVALCYC)

Exports:
  eval

—— package EVALCYC EvaluateCycleIndicators ——

)abbrev package EVALCYC EvaluateCycleIndicators
++ Author: William H. Burge
++ Date Created: 1986
++ Date Last Updated: Feb 1992
++ Description:
  ++ This package is to be used in conjunction with the CycleIndicators package.
  ++ It provides an evaluation function for SymmetricPolynomials.

EvaluateCycleIndicators(F):T==C where
  F:Algebra Fraction Integer
  I==>Integer
  L==>List
  SPOL==>SymmetricPolynomial
  RN==>Fraction Integer
  PR==>Polynomial(RN)
  PTN==>Partition()
  lc ==> leadingCoefficient
  red ==> reductum
  T== with
    eval:((I->F),SPOL RN)->F
      ++\spad{eval(f,s)} evaluates the cycle index s by applying
      ++ the function f to each integer in a monomial partition,
      ++ forms their product and sums the results over all monomials.
  C== add
    evp:((I->F),PTN)->F
    fn:I->F
    pt:PTN
    spol:SPOL RN
    i:I
    evp(fn, pt)== _*/[fn i for i in pt::(L I)]
eval(fn,spol)==
if spol=0
  then 0
else ((lc spol)* evp(fn,degree spol)) + eval(fn,red spol)

---

"EVALCYC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EVALCYC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"EVALCYC" -> "PFECAT"

---

package ESCONT ExpertSystemContinuityPackage

---

\)
\)
\)set break resume
\)sys rm -f ExpertSystemContinuityPackage.output
\)spool ExpertSystemContinuityPackage.output
\)set message test on
\)set message auto off
\)clear all

--S 1 of 1
\)
\)show ExpertSystemContinuityPackage

--R
--R ExpertSystemContinuityPackage is a package constructor
--R Abbreviation for ExpertSystemContinuityPackage is ESCONT
--R This constructor is exposed in this frame.
--R Issue \)edit bookvol10.4.pamphlet to see algebra source code for ESCONT
--R
--R------------------------------------------------ operations --------------------------------
--R df2st : DoubleFloat -> String
--R functionIsFracPolynomial? : Record(var: Symbol,fn: Expression(DoubleFloat),range: Segment(OrderedCompletion(DoubleFloat)),abserr: DoubleFloat,relerr: DoubleFloat) -> Boolean
--R gethi : Segment(OrderedCompletion(DoubleFloat)) -> DoubleFloat
--R getlo : Segment(OrderedCompletion(DoubleFloat)) -> DoubleFloat
--R ldf2lst : List(DoubleFloat) -> List(String)
--R polynomialZeros : (Polynomial(Fraction(Integer)),Symbol,Segment(OrderedCompletion(DoubleFloat))) -> List(DoubleFloat)
--R problemPoints : (Expression(DoubleFloat),Symbol,Segment(OrderedCompletion(DoubleFloat))) -> List(DoubleFloat)
--R sdf2lst : Stream(DoubleFloat) -> List(String)
--R singularitiesOf : (Expression(DoubleFloat),List(Symbol),Segment(OrderedCompletion(DoubleFloat))) -> Stream(DoubleFloat)
--R singularitiesOf : (Vector(Expression(DoubleFloat)),List(Symbol),Segment(OrderedCompletion(DoubleFloat))) -> Stream(DoubleFloat)
--R zerosOf : (Expression(DoubleFloat),List(Symbol),Segment(OrderedCompletion(DoubleFloat))) -> Stream(DoubleFloat)
--R
--E 1

)spool
)lisp (bye)

---

— ExpertSystemContinuityPackage.help —

====================================================================
ExpertSystemContinuityPackage examples
====================================================================

ExpertSystemContinuityPackage is a package of functions for the use of
domains belonging to the category NumericalIntegration.

See Also:
o )show ExpertSystemContinuityPackage

---

ExpertSystemContinuityPackage (ESCONT)

Exports:
df2st functionIsFracPolynomial? gethi getlo ldf2lst
polynomialZeros problemPoints sdf2lst singularitiesOf singularitiesOf
zerosOf

— package ESCONT ExpertSystemContinuityPackage —
abbrev package ESCONT ExpertSystemContinuityPackage
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: June 1995
++ Description:
++ ExpertSystemContinuityPackage is a package of functions for the use of
++ domains belonging to the category \axiomType{NumericalIntegration}.

ExpertSystemContinuityPackage(): E == I where
   EF2  ==> ExpressionFunctions2
   FI   ==> Fraction Integer
   EFI  ==> Expression Fraction Integer
   PFI  ==> Polynomial Fraction Integer
   DF   ==> DoubleFloat
   LDF  ==> List DoubleFloat
   EDF  ==> Expression DoubleFloat
   VEDF ==> Vector Expression DoubleFloat
   SDF  ==> Stream DoubleFloat
   SS   ==> Stream String
   EEAFD ==> Equation Expression DoubleFloat
   LEDF ==> List Expression Double Float
   KEDF ==> Kernel Expression DoubleFloat
   LKEFDF ==> List Kernel Expression DoubleFloat
   PDF  ==> Polynomial DoubleFloat
   FPDF ==> Fraction Polynomial DoubleFloat
   OCFDF ==> OrderedCompletion DoubleFloat
   SOCDF ==> Segment OrderedCompletion DoubleFloat
   NIA  ==> Record(var:Symbol,fn:EDF,range:SOCDF,abserr:DF,relerr:DF)
   UP   ==> UnivariatePolynomial
   BO   ==> BasicOperator
   RS   ==> Record(zeros: SDF,ones: SDF,singularities: SDF)

E == with
   getlo : SOCDF -> DF
      ++ getlo(u) gets the \axiomType{DoubleFloat} equivalent of
      ++ the first endpoint of the range \axiom{u}
   gethi : SOCDF -> DF
      ++ gethi(u) gets the \axiomType{DoubleFloat} equivalent of
      ++ the second endpoint of the range \axiom{u}
   functionIsFracPolynomial?: NIA -> Boolean
      ++ functionIsFracPolynomial?(args) tests whether the function
      ++ can be retracted to \axiomType{Fraction(Polynomial(DoubleFloat))}
   problemPoints:(EDF,Symbol,SOCDF) -> List DF
      ++ problemPoints(f,var,range) returns a list of possible problem points
      ++ by looking at the zeros of the denominator of the function \spad{f}
      ++ if it can be retracted to \axiomType{Polynomial(DoubleFloat)}.
   zerosOf:(EDF,List Symbol,SOCDF) -> SDF
      ++ zerosOf(e,vars,range) returns a list of points
      ++ \axiomType{DoubleFloat} at which a NAG fortran version of \spad{e}
++ will most likely produce an error.
singularitiesOf: (EDF, List Symbol, SOCDF) -> SDF
  ++ singularitiesOf(e, vars, range) returns a list of points
  ++ (\text{axiomType\{Doublefloat\}}) at which a NAG fortran
  ++ version of spad\{e\} will most likely produce
  ++ an error. This includes those points which evaluate to 0/0.
singularitiesOf: (Vector EDF, List Symbol, SOCDF) -> SDF
  ++ singularitiesOf(v, vars, range) returns a list of points
  ++ (\text{axiomType\{Doublefloat\}}) at which a NAG fortran
  ++ version of spad\{v\} will most likely produce
  ++ an error. This includes those points which evaluate to 0/0.
polynomialZeros: (PFI, Symbol, SOCDF) -> LDF
  ++ polynomialZeros(fn, var, range) calculates the real zeros of the
  ++ polynomial which are contained in the given interval. It returns
  ++ a list of points (\text{axiomType\{Doublefloat\}}) for which the univariate
  ++ polynomial spad\{fn\} is zero.
df2st: DF -> String
  ++ df2st(n) coerces a \text{axiomType\{DoubleFloat\}} to \text{axiomType\{String\}}
ldf2lst:LDF -> List String
  ++ ldf2lst(ln) coerces a List of \text{axiomType\{DoubleFloat\}} to
  ++ \text{axiomType\{List\}(\text{axiomType\{String\})
sdf2lst:SDF -> List String
  ++ sdf2lst(ln) coerces a Stream of \text{axiomType\{DoubleFloat\}} to
  ++ \text{axiomType\{List\}(\text{axiomType\{String\})

I ==> ExpertSystemToolsPackage add
import ExpertSystemToolsPackage

functionIsPolynomial?(args:NIA):Boolean ==
  -- tests whether the function can be retracted to a polynomial
  (retractIfCan(args.fn)@Union(PDF,"failed"))$EDF case PDF

isPolynomial?(f:EDF):Boolean ==
  -- tests whether the function can be retracted to a polynomial
  (retractIfCan(f)@Union(PDF,"failed"))$EDF case PDF

isConstant?(f:EDF):Boolean ==
  -- tests whether the function can be retracted to a constant (DoubleFloat)
  (retractIfCan(f)@Union(DF,"failed"))$EDF case DF

denominatorIsPolynomial?(args:NIA):Boolean ==
  -- tests if the denominator can be retracted to polynomial
  a:= copy args
  a.fn:=denominator(args.fn)
  (functionIsPolynomial?(a))$Boolean

denIsPolynomial?(f:EDF):Boolean ==
  -- tests if the denominator can be retracted to polynomial
  (isPolynomial?(denominator f))$Boolean
listInRange(l:LDF,range:SOCDF):LDF ==
    -- returns a list with only those elements internal to the range range
    [t for t in l | in?(t,range)]

loseUntil(l:SDF,a:DF):SDF ==
    empty?(l)$SDF => l
    f := first(l)$SDF
    (abs(f) <= abs(a)) => loseUntil(rest(l)$SDF,a)
    l

retainUntil(l:SDF,a:DF,b:DF,flag:Boolean):SDF ==
    empty?(l)$SDF => l
    f := first(l)$SDF
    (in?(f)$ExpertSystemContinuityPackage1(a,b)) =>
        concat(f,retainUntil(rest(l),a,b,false))
    flag => empty()$SDF
    retainUntil(rest(l),a,b,true)

streamInRange(l:SDF,range:SOCDF):SDF ==
    -- returns a stream with only those elements internal to the range range
    a := getlo(range := dfRange(range))
    b := gethi(range)
    explicitlyFinite?(l) =>
        select(in?($ExpertSystemContinuityPackage1(a,b),l)$SDF
    negative?(a*b) => retainUntil(l,a,b,false)
    negative?(a) =>
        l := loseUntil(l,b)
        retainUntil(l,a,b,false)
    l := loseUntil(l,a)
    retainUntil(l,a,b,false)

getStream(n:Symbol,s:String):SDF ==
    import RS
    entry?(n,bfKeys()$BasicFunctions)$(List(Symbol)) =>
        c := bfEntry(n)$BasicFunctions
        (s = "zeros")@Boolean => c.zeros
        (s = "singularities")@Boolean => c.singularities
        (s = "ones")@Boolean => c.ones
        empty()$SDF

polynomialZeros(fn:PFI,var:Symbol,range:SOCDF):LDF ==
    up := unmakeSUP(univariate(fn)$PFI)$UP(var,FI)
    range := dfRange(range)
    r:Record(left:FI,right:FI) := [df2fi(getlo(range)), df2fi(gethi(range))]
    ans:List(Record(left:FI,right:FI)) :=
        realZeros(up,r,1/1000000000000000000)$RealZeroPackageQ(UP(var,FI))
    listInRange(dflist(ans),range)

functionIsFracPolynomial?(args:NIA):Boolean ==
-- tests whether the function can be retracted to a fraction
-- where both numerator and denominator are polynomial
(retractIfCan(args.fn)@Union(FPDF,"failed"))$EDF case FPDF

problemPoints(f:EDF,var:Symbol,range:SOCDF):LDF ==
(denIsPolynomial?(f))@Boolean =>
c := retract(edf2efi(denominator(f)))@PFI
polynomialZeros(c,var,range)
empty()$LDF

zerosOf(e:EDF,vars:List Symbol,range:SOCDF):SDF ==
(u := isQuotient(e)) case EDF =>
singularitiesOf(u,vars,range)
k := kernels(e)$EDF
((nk := # k) = 0)@Boolean => empty()$SDF -- constant found.
(nk = 1)@Boolean => -- single expression found.
ker := first(k)$LKEDF
n := name(operator(ker)$KEDF)$BO
entry?(n,vars) => -- polynomial found.
c := retract(edf2efi(e))@PFI
coerce(polynomialZeros(c,n,range))$SDF
a := first(argument(ker)$KEDF)$LEDF
(not (n = log :: Symbol)$Boolean) and ((w := isPlus a) case LEDF) =>
var:Symbol := first(variables(a))
c:EDF := w.2
c1:EDF := w.1
-- entry?(c1,[b::EDF for b in vars]) and (one?(# vars)) =>
entry?(c1,[b::EDF for b in vars]) and ((# vars) = 1) =>
c2:DF := edf2df c
c3 := c2 :: OCDF
varEqn := equation(varEdf,c1-c)$EEDF
range2 := (lo(range)+c3)..(hi(range)+c3)
s := zerosOf(subst(e,varEqn)$EDF,vars,range2)
st := map(t1 +-> t1-c2,s)$StreamFunctions2(DF,DF)
streamInRange(st,range)
zerosOf(a,vars,range)
(t := isPlus(e)$EDF) case LEDF => -- constant + expression
# t > 2 => empty()$SDF
entry?(a,[b::EDF for b in vars]) => -- finds entries like sqrt(x)
ost := getStream(n,\"ones\")
o := edf2df(second(t)$LEDFF)
--
one?(o) or one?(-o) => -- is it like (f(x) -/+ 1)
(o = 1) or (-o = 1) => -- is it like (f(x) -/+ 1)
ost := map(t2 +-> -t2/o,st)$StreamFunctions2(DF,DF)
streamInRange(st,range)
empty()$SDF
empty()$SDF
entry?(a,[b::EDF for b in vars]) => -- finds entries like sqrt(x)
ost := getStream(n,\"zeros\")
\texttt{streamInRange(st,range)}
\begin{align*}
& (n = \text{\texttt{tan}} :: \text{Symbol})@\text{Boolean} \Rightarrow \\
& \text{concat}([\text{zerosOf}(a,\text{vars},\text{range}),\text{singularitiesOf}(a,\text{vars},\text{range})]) \\
& (n = \text{\texttt{sin}} :: \text{Symbol})@\text{Boolean} \Rightarrow \\
& \text{concat}([\text{zerosOf}(a,\text{vars},\text{range}),\text{singularitiesOf}(a,\text{vars},\text{range})]) \\
& \text{empty()}$SDF
\end{align*}

\texttt{(t := \text{isPlus}(e)$EDF) \text{ case LEDF} \Rightarrow \text{empty()}$SDF -- INCOMPLETE!!!}
\texttt{(v := \text{isTimes}(e)$EDF) \text{ case LEDF} \Rightarrow \\
\text{concat}([\text{zerosOf}(u,\text{vars},\text{range}) \text{ for } u \text{ in } v]) \\
\text{empty()}$SDF

\texttt{singularitiesOf}(e:EDF,\text{vars}:\text{List Symbol},\text{range}:SOCDF)$SDF ==
\begin{align*}
& (u := \text{isQuotient}(e)) \text{ case EDF} \Rightarrow \\
& \text{zerosOf}(u,\text{vars},\text{range}) \\
& (t := \text{isPlus } e) \text{ case LEDF} \Rightarrow \\
& \text{concat}([\text{singularitiesOf}(u,\text{vars},\text{range}) \text{ for } u \text{ in } t]) \\
& (v := \text{isTimes } e) \text{ case LEDF} \Rightarrow \\
& \text{concat}([\text{singularitiesOf}(u,\text{vars},\text{range}) \text{ for } u \text{ in } v]) \\
& (k := \text{mainKernel } e) \text{ case KEDF} \Rightarrow \\
& n := \text{name(operator } k) \\
& \text{entry?(n,vars) } \Rightarrow \text{coerce(problemPoints(e,n,range))}$SDF \\
& a:EDF := (\text{argument } k).1 \\
& (\text{not } (n = \text{\texttt{log}} :: \text{Symbol}@\text{Boolean}) \text{ and } ((w := \text{isPlus } a) \text{ case LEDF}) \Rightarrow \\
& \text{var}:\text{Symbol} := \text{first(variables}(a)) \\
& c:EDF := w.2 \\
& c1:EDF := w.1 \\
& \text{-- entry?(c1,[]}b::EDF \text{ for } b \text{ in } \text{vars}{]} \text{ and } (\text{one?}(# \text{vars})) \Rightarrow \\
& \text{entry?(c1,[]}b::EDF \text{ for } b \text{ in } \text{vars}{]} \text{ and } ((# \text{vars}) = 1) \Rightarrow \\
& c2:DF := \text{edf2df } c \\
& c3 := c2 :: \text{OCDF} \\
& \text{varEqn} := \text{equation}(\text{varEdf},c1-c)$EEDF \\
& \text{range2} := (\text{lo}(\text{range})+c3)\ldots(\text{hi}(\text{range})+c3) \\
& s := \text{singularitiesOf}(\text{subst}(e,\text{varEqn})$EDF,\text{vars},\text{range2}) \\
& \text{st} := \text{map}(t3 \rightarrow t3-c2,a)$\text{StreamFunctions2}(DF,DF) \\
& \text{streamInRange(st,range)} \\
& \text{singularitiesOf}(a,\text{vars},\text{range}) \\
& \text{entry?}(a,[]}b::EDF \text{ for } b \text{ in } \text{vars}{]} \Rightarrow \\
& \text{st} := \text{getStream}(n,\text{"singularities"}) \\
& \text{streamInRange(st,range)} \\
& (n = \text{\texttt{log}} :: \text{Symbol}@\text{Boolean} \Rightarrow \\
& \text{concat}([\text{zerosOf}(a,\text{vars},\text{range}),\text{singularitiesOf}(a,\text{vars},\text{range})]) \\
& \text{empty()}$SDF
\end{align*}

\texttt{singularitiesOf}(v:VEDF,\text{vars}:\text{List Symbol},\text{range}:SOCDF)$SDF ==
\begin{align*}
& \text{ls} := [\text{singularitiesOf}(u,\text{vars},\text{range}) \text{ for } u \text{ in } \text{entries}(v)\text{$VEDF}] \\
& \text{concat}(\text{ls})$SDF
package ESCONT1 ExpertSystemContinuityPackage1

---

--- ESCONT.dotabb ---

"ESCONT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESCONT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"ESCONT" -> "ACFS"

---

package ESCONT1 ExpertSystemContinuityPackage1

--- ExpertSystemContinuityPackage1.input ---

)set break resume
)sys rm -f ExpertSystemContinuityPackage1.output
)spool ExpertSystemContinuityPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpertSystemContinuityPackage1

--R
--R ExpertSystemContinuityPackage1(A: DoubleFloat,B: DoubleFloat) is a package constructor
--R Abbreviation for ExpertSystemContinuityPackage1 is ESCONT1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ESCONT1

--R------------------ Operations ----------------------------
--R in?: DoubleFloat -> Boolean
--R
--E 1

)spool
)lisp (bye)

---

--- ExpertSystemContinuityPackage1.help ---

====================================================================
ExpertSystemContinuityPackage1 examples
====================================================================
ExpertSystemContinuityPackage1 exports a function to check range inclusion

See Also:
  o )show ExpertSystemContinuityPackage1

==

ExpertSystemContinuityPackage1 (ESCONT1)

Exports:
in?

— package ECONT1 ExpertSystemContinuityPackage1 —

)abbrev package ECONT1 ExpertSystemContinuityPackage1
++ Author: Brian Dupee
++ Date Created: May 1994
++ Date Last Updated: June 1995
++ Description:
  ++ ExpertSystemContinuityPackage1 exports a function to check range inclusion

ExpertSystemContinuityPackage1(A:DF,B:DF): E == I where
  EF2  ==> ExpressionFunctions2
  FI   ==> Fraction Integer
  EFI  ==> Expression Fraction Integer
  PFI  ==> Polynomial Fraction Integer
  DF   ==> DoubleFloat
  LDF  ==> List DoubleFloat
  EDF  ==> Expression DoubleFloat
  VEDF ==> Vector Expression DoubleFloat
  SDF  ==> Stream DoubleFloat
  SS   ==> Stream String
PACKAGE ESTOOLS EXPERTSYSTEMTOOLSPACKAGE

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEDF</td>
<td>Equation Expression DoubleFloat</td>
</tr>
<tr>
<td>LEDF</td>
<td>List Expression DoubleFloat</td>
</tr>
<tr>
<td>KEDF</td>
<td>Kernel Expression DoubleFloat</td>
</tr>
<tr>
<td>LKEDF</td>
<td>List Kernel Expression DoubleFloat</td>
</tr>
<tr>
<td>PDF</td>
<td>Polynomial DoubleFloat</td>
</tr>
<tr>
<td>FPDF</td>
<td>Fraction Polynomial DoubleFloat</td>
</tr>
<tr>
<td>OCDF</td>
<td>OrderedCompletion DoubleFloat</td>
</tr>
<tr>
<td>SCDF</td>
<td>Segment OrderedCompletion DoubleFloat</td>
</tr>
<tr>
<td>NIA</td>
<td>Record(var:Symbol,fn:EDF,range:SCDF,abserr:DF,relerr:DF)</td>
</tr>
<tr>
<td>UP</td>
<td>UnivariatePolynomial</td>
</tr>
<tr>
<td>BO</td>
<td>BasicOperator</td>
</tr>
<tr>
<td>RS</td>
<td>Record(zeros: SDF,ones: SDF,singularities: SDF)</td>
</tr>
</tbody>
</table>

E => with

in?:DF -> Boolean
++ in?(p) tests whether point p is internal to the range \spad{A..B}

I => add

in?(p:DF):Boolean ==
a:Boolean := (p < B)$DF
b:Boolean := (A < p)$DF
(a and b)@Boolean

---

-- ESCONT1.dotabb --

"ESCONT1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESCONT1"]
"Package" [color="#FF4488"]
"ESCONT1" -> "Package"

---

package ESTOOLS ExpertSystemToolsPackage

-- ExpertSystemToolsPackage.input --

)set break resume
)sys rm -f ExpertSystemToolsPackage.output
)spool ExpertSystemToolsPackage.output
)set message test on
)set message auto off
>clear all

--S 1 of 1

)show ExpertSystemToolsPackage

--R

--R ExpertSystemToolsPackage is a package constructor

--R Abbreviation for ExpertSystemToolsPackage is ESTOOLS

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ESTOOLS

--)--------------------------------------------------------------- Operations -----------------------------------
--R concat : (Result,Result) -> Result  concat : List(Result) -> Result
--R df2mf : DoubleFloat -> MachineFloat  df2st : DoubleFloat -> String
--R f2df : Float -> DoubleFloat  f2st : Float -> String
--R outputMeasure : Float -> String

--)------------------------------------------------------------------- Operations -----------------------------------
--R concat : (Result,Result) -> Result
--R df2mf : DoubleFloat -> MachineFloat
--R df2st : DoubleFloat -> String

--)-------------------------------------------------------------------------------- Operations -----------------------------------
--R att2Result : Record(endPointContinuity: Union(continuous: Continuous at the end points,lowerSingular: There is a ... range is infinite,bothInfinite: Both top and bottom points are infinite,notEvaluated: Range not yet evaluated)) -> Result

--)-------------------------------------------------------------------------------- Operations -----------------------------------
--R df2ef : DoubleFloat -> Expression(Float)

--)-------------------------------------------------------------------------------- Operations -----------------------------------
--R df2fi : DoubleFloat -> Fraction(Integer)

--)-------------------------------------------------------------------------------- Operations -----------------------------------
--R dfRange : Segment(OrderedCompletion(DoubleFloat)) -> Segment(OrderedCompletion(DoubleFloat))

--)-------------------------------------------------------------------------------- Operations -----------------------------------
--R df2efi : Expression(DoubleFloat) -> Expression(Float)

--)-------------------------------------------------------------------------------- Operations -----------------------------------
--R dfRange : Segment(OrderedCompletion(DoubleFloat)) -> Segment(OrderedCompletion(DoubleFloat))

--E 1

)spool

)lisp (bye)
---

— ExpertSystemToolsPackage.help —

================================================================================

ExpertSystemToolsPackage examples
================================================================================

ExpertSystemToolsPackage contains some useful functions for use by the computational agents of numerical solvers.

See Also:
  o )show ExpertSystemToolsPackage

---

ExpertSystemToolsPackage (ESTOOLS)

Exports:

att2Result  concat  convert  dflist  dfRange
df2ef       df2fi    df2mf    df2st   edf2df
edf2ef      edf2efi  edf2fi   ef2edf  expenseOfEvaluation
fi2df        f2df     f2st     gethi   getlo
iflist2Result in?    isQuotient ldf2lst  ldf2vmf
mat          measure2Result measure2Result gethi  ldf2vmf
outputMeasure pdf2df  pdf2ef  sdf2lst  socf2ocdf
vedf2vef

— package ESTOOLS ExpertSystemToolsPackage —

)abbrev package ESTOOLS ExpertSystemToolsPackage
++ Author: Brian Dupee
++ Date Created: May 1994
\textbf{CHAPTER 6. CHAPTER E}

++ Date Last Updated: July 1996
++ Description: 
++ \texttt{\textbackslash axiom{ExpertSystemToolsPackage}} contains some useful functions for use 
++ by the computational agents of numerical solvers.

\texttt{ExpertSystemToolsPackage():E == I where}
\begin{itemize}
  \item \texttt{LEDF ==> List Expression DoubleFloat}
  \item \texttt{KEDF ==> Kernel Expression DoubleFloat}
  \item \texttt{LKEDF ==> List Kernel Expression DoubleFloat}
  \item \texttt{VEDF ==> Vector Expression DoubleFloat}
  \item \texttt{VEF ==> Vector Expression Float}
  \item \texttt{MF ==> MachineFloat}
  \item \texttt{EFI ==> Expression Fraction Integer}
  \item \texttt{MDF ==> Matrix DoubleFloat}
  \item \texttt{LDF ==> List DoubleFloat}
  \item \texttt{PDF ==> Polynomial DoubleFloat}
  \item \texttt{EDF ==> Expression DoubleFloat}
  \item \texttt{EF ==> Expression Float}
  \item \texttt{DF ==> DoubleFloat}
  \item \texttt{F ==> Float}
  \item \texttt{ST ==> String}
  \item \texttt{LST ==> List String}
  \item \texttt{SS ==> Stream String}
  \item \texttt{FI ==> Fraction Integer}
  \item \texttt{R ==> Ring}
  \item \texttt{OR ==> OrderedRing}
  \item \texttt{ON ==> Record(additions:INT,multiplications:INT,exponentiations:INT,functionCalls:INT)}
  \item \texttt{RVE ==> Record(val:EDF,exponent:INT)}
  \item \texttt{BO ==> BasicOperator}
  \item \texttt{OFCF ==> OrderedCompletion Float}
  \item \texttt{OCDF ==> OrderedCompletion DoubleFloat}
  \item \texttt{SOCF ==> Segment OrderedCompletion Float}
  \item \texttt{SOCDF ==> Segment OrderedCompletion DoubleFloat}
  \item \texttt{Measure ==> Record(measure:F, name:String, explanations:List String)}
  \item \texttt{Measure2 ==> Record(measure:F, name:String, explanations:List String, extra:Result)}
  \item \texttt{CTYPE ==> Union(continuous: "Continuous at the end points",}
  \begin{itemize}
    \item \texttt{lowerSingular: "There is a singularity at the lower end point",}
    \item \texttt{upperSingular: "There is a singularity at the upper end point",}
    \item \texttt{bothSingular: "There are singularities at both end points",}
    \item \texttt{notEvaluated: "End point continuity not yet evaluated"}
  \end{itemize}
  \item \texttt{RTYPE ==> Union(finite: "The range is finite",}
  \begin{itemize}
    \item \texttt{lowerInfinite: "The bottom of range is infinite",}
    \item \texttt{upperInfinite: "The top of range is infinite",}
    \item \texttt{bothInfinite: "Both top and bottom points are infinite",}
\end{itemize}
\end{itemize}
notEvaluated: "Range not yet evaluated")

STYPE ==> Union(str:SDF, notEvaluated:"Internal singularities not yet evaluated")

ATT ==> Record(endPointContinuity:CTYPE, singularitiesStream:STYPE, range:RTYPE)


E with

f2df:F -> DF
  ++ f2df(f) is a function to convert a \axiomType{Float} to a
  ++ \axiomType{DoubleFloat}

ef2edf:EF -> EDF
  ++ ef2edf(f) is a function to convert an \axiomType{Expression Float}
  ++ to an \axiomType{Expression DoubleFloat}

ocf2ocdf: OCF -> OCDF
  ++ ocf2ocdf(a) is a function to convert an \axiomType{OrderedCompletion}
  ++ Float to an \axiomType{OrderedCompletion DoubleFloat}

socf2socdf: SOCF -> SOCDF
  ++ socf2socdf(a) is a function to convert a \axiomType{Segment OrderedCompletion}
  ++ Float to an \axiomType{Segment OrderedCompletion DoubleFloat}

convert: List SOCF -> List SOCDF
  ++ convert(l) is a function to convert a \axiomType{Segment OrderedCompletion Float}
  ++ to a \axiomType{Segment OrderedCompletion DoubleFloat}

df2fi :DF -> FI
  ++ df2fi(n) is a function to convert a \axiomType{DoubleFloat} to a
  ++ \axiomType{Fraction Integer}

edf2fi :EDF -> FI
  ++ edf2fi(n) maps \axiomType{Expression DoubleFloat} to
  ++ \axiomType{Fraction Integer}
  ++ It is an error if n is not coercible to Fraction Integer

edf2df :EDF -> DF
  ++ edf2df(n) maps \axiomType{Expression DoubleFloat} to
  ++ \axiomType{DoubleFloat}
  ++ It is an error if \spad{n} is not coercible to DoubleFloat

isQuotient:EDF -> Union(EDF,"failed")
  ++ isQuotient(expr) returns the quotient part of the input
  ++ expression or \spad{"failed"} if the expression is not of that form.

expenseOfEvaluation:VEDF -> F
  ++ expenseOfEvaluation(o) gives an approximation of the cost of
  ++ evaluating a list of expressions in terms of the number of basic
  ++ operations.
  ++ < 0.3 inexpensive ; 0.5 neutral ; > 0.7 very expensive
  ++ 400 'operation units' -> 0.75
  ++ 200 'operation units' -> 0.5
  ++ 83 'operation units' -> 0.25
  ++ ** = 4 units , function calls = 10 units.

numberOfOperations:VEDF -> ON
  ++ numberOfOperations(ode) counts additions, multiplications,
  ++ exponentiations and function calls in the input set of expressions.

edf2efi:EDF -> EFI
++ edf2efi(e) coerces \texttt{Expression DoubleFloat} into
++ \texttt{Expression Fraction Integer}
dfRange:SOCDF \rightarrow SOCDF
++ \texttt{dfRange(r)} converts a range including
++ \inputbitmap{htbmdir/plusminus.bitmap} \texttt{\infty}
++ to \texttt{Expression DoubleFloat} equivalents.
dflist:List(Record(left:F, right:F)) \rightarrow LDF
++ \texttt{dflist(l)} returns a list of \texttt{Expression DoubleFloat} equivalents of list \texttt{l}
df2mf:DF \rightarrow MF
++ \texttt{df2mf(n)} coerces a \texttt{Expression DoubleFloat} to \texttt{Expression MachineFloat}
ldf2vmf:LDF \rightarrow VMF
++ \texttt{ldf2vmf(l)} coerces a \texttt{List Expression DoubleFloat} to
++ \texttt{List Expression MachineFloat}
edf2ef:EDF \rightarrow EF
++ \texttt{edf2ef(e)} maps \texttt{Expression DoubleFloat} to
++ \texttt{Expression Float}
vedf2vef:VEDF \rightarrow VEF
++ \texttt{vedf2vef(v)} maps \texttt{Vector Expression DoubleFloat} to
++ \texttt{Vector Expression Float}
in?:DF,SOCDF \rightarrow Boolean
++ \texttt{in?(p,range)} tests whether point \texttt{p} is internal to the
++ range \texttt{range}
df2st:DF \rightarrow ST
++ \texttt{df2st(n)} coerces a \texttt{Expression DoubleFloat} to \texttt{Expression String}
f2st:F \rightarrow ST
++ \texttt{f2st(n)} coerces a \texttt{Expression Float} to \texttt{Expression String}
ldf2lst:LDF \rightarrow LST
++ \texttt{ldf2lst(l)} coerces a \texttt{List Expression DoubleFloat} to \texttt{List Expression String}
sdf2lst:SDF \rightarrow LST
++ \texttt{sdf2lst(ln)} coerces a \texttt{Stream Expression DoubleFloat} to \texttt{String}
getlo : SOCDF \rightarrow DF
++ \texttt{getlo(u)} gets the \texttt{Expression DoubleFloat} equivalent of
++ the first endpoint of the range \texttt{\spad{u}}
gethi : SOCDF \rightarrow DF
++ \texttt{gethi(u)} gets the \texttt{Expression DoubleFloat} equivalent of
++ the second endpoint of the range \texttt{\spad{u}}
concat:(Result,Result) \rightarrow Result
++ \texttt{concat(a,b)} adds two aggregates of type \texttt{Result}.
concat:(List Result) \rightarrow Result
++ \texttt{concat(l)} concatenates a list of aggregates of type \texttt{Result}.
outputMeasure:F \rightarrow ST
++ \texttt{outputMeasure(n)} rounds \texttt{\spad{n}} to 3 decimal places and outputs
++ it as a string
measure2Result:Measure \rightarrow Result
++ \texttt{measure2Result(m)} converts a measure record into a \texttt{Result}
measure2Result:Measure2 \rightarrow Result
++ \texttt{measure2Result(m)} converts a measure record into a \texttt{Result}
att2Result:ATT \rightarrow Result
++ \texttt{att2Result(m)} converts a attributes record into a \texttt{Result}
iflist2Result:IFV \rightarrow Result
++ iflist2Result(m) converts a attributes record into a \axiomType{Result}

pdf2ef:PDF -> EF
++ pdf2ef(p) coerces a \axiomType{Polynomial DoubleFloat} to
++ \axiomType{Expression Float}

pdf2df:PDF -> DF
++ pdf2df(p) coerces a \axiomType{Polynomial DoubleFloat} to
++ \axiomType{DoubleFloat}. It is an error if \axiom{p} is not
++ retractable to DoubleFloat.

df2ef:DF -> EF
++ df2ef(a) coerces a \axiomType{DoubleFloat} to \axiomType{Expression Float}

fi2df:FI -> DF
++ fi2df(f) coerces a \axiomType{Fraction Integer} to \axiomType{DoubleFloat}

mat:(LDF,NNI) -> MDF
++ mat(a,n) constructs a one-dimensional matrix of a.

I ==> add

mat(a:LDF,n:NNI):MDF ==
  empty?(a)$LDF => zero(1,n)$MDF
  matrix(list([i for i in a for j in 1..n])$(List LDF))$MDF

f2df(f:F):DF == (convert(f)@DF)$F

ef2edf(f:EF):EDF == map(f2df,f)$EF2(F,DF)

fi2df(f:FI):DF == coerce(f)$DF

ocf2ocdf(a:OCF):OCDF ==
  finite? a => (f2df(retract(a)@F))::OCDF
  a pretend OCF

socf2socdf(a:SOCF):SOCDF ==
  segment(ocf2ocdf(lo a),ocf2ocdf(hi a))

convert(l:List SOCF):List SOCDF == [socf2socdf a for a in l]

pdf2df(p:PDF):DF == retract(p)$DF

df2ef(a:DF):EF ==
  b := convert(a)@Float
  coerce(b)$EF

pdf2ef(p:PDF):EF == df2ef(pdf2df(p))

edf2fi(m:EDF):FI == retract(retract(m)$DF)$FI

edf2df(m:EDF):DF == retract(m)$DF

df2fi(r:DF):FI == (retract(r)$FI)$DF
dfRange(r:SOCDF):SOCDF ==
  if infinite?(lo(r))$OCDF then r := -(max()$DF :: OCDF)..hi(r)$SOCDF
  if infinite?(hi(r))$OCDF then r := lo(r)$SOCDF..(max()$DF :: OCDF)
  r

dflist(l:List(Record(left:FI,right:FI))):LDF == [u.left :: DF for u in l]
edf2efi(f:EDF):EFI == map(df2fi,f)$EF2(DF,FI)
df2st(n:DF):String == (convert((convert(n)@Float)$DF)@ST)$Float
f2st(n:F):String == (convert(n)@ST)$Float
ldf2lst(ln:LDF):LST == [df2st f for f in ln]
sdf2lst(ln:SDF):LST ==
  explicitlyFinite? ln =>
  m := map(df2st,ln)$StreamFunctions2(DF,ST)
  if index?(20,m)$SS then
    split!(m,20)
  m := concat(m,".......")
  m := complete(m)$SS
  entries(m)$SS
  empty()$LST

df2mf(n:DF):MF == (df2fi(n))::MF
ldf2vmf(l:LDF):VMF ==
  m := [df2mf(n) for n in l]
  vector(m)$VMF
edf2ef(e:EDF):EF == map(convert$DF,e)$EF2(DF,Float)
vedf2vef(vedf:VEDF):VEF == vector([edf2ef e for e in members(vedf)])
getlo(u:SOCDF):DF == retract(lo(u))@DF
gethi(u:SOCDF):DF == retract(hi(u))@DF
in?(p:DF,range:SOCDF):Boolean ==
  top := gethi(range)
  bottom := getlo(range)
  a:Boolean := (p < top)$DF
  b:Boolean := (p > bottom)$DF
  (a and b)$Boolean
isQuotient(expr:EDF):Union(EDF,"failed") ==
  (k := mainKernel expr) case KEDF =>
    (expr = inv(f := k :: KEDF :: EDF)$EDF)$EDF => f
    one?(numerator expr) => denominator expr
(numerator expr) = 1 => denominator expr
"failed"
"failed"

\[
\text{numberOfOperations1}(\text{fn: EDF}, \text{numbersSoFar: ON}) = \begin{cases} \\
\text{(u := isQuotient(fn)) case EDF =>} \\
\text{numbersSoFar := numberOfOperations1(u, numbersSoFar)} \\
\text{(p := isPlus(fn)) case LEDF =>} \\
p := \text{coerce(p)@LEDF} \\
np := \#p \\
\text{numbersSoFar.additions := (numbersSoFar.additions)+np-1} \\
\text{for i in 1..np repeat} \\
\text{numbersSoFar := numberOfOperations1(p.i, numbersSoFar)} \\
\text{numbersSoFar} \\
\text{(t := isTimes(fn)) case LEDF =>} \\
t := \text{coerce(t)@LEDF} \\
tn := \#t \\
\text{numbersSoFar.multiplications := (numbersSoFar.multiplications)+nt-1} \\
\text{for i in 1..nt repeat} \\
\text{numbersSoFar := numberOfOperations1(t.i, numbersSoFar)} \\
\text{numbersSoFar} \\
\text{if (e := isPower(fn)) case RVE then} \\
e := \text{coerce(e)@RVE} \\
e.exponent>1 => \\
\text{numbersSoFar.exponentiations := inc(numbersSoFar.exponentiations)} \\
\text{numbersSoFar := numberOfOperations1(e.val, numbersSoFar)} \\
\text{lk := kernels(fn)} \\
\#lk = 1 => -- #lk = 0 => constant found (no further action) \\
k := \text{first(lk)@LKEDF} \\
n := \text{name(operator(k))@KEDF}$BO \\
\text{entry?(n, variables(fn)@EDF)$LS => numbersSoFar} -- solo variable found \\
a := \text{first(argument(k)@KEDF)$LEDF} \\
\text{numbersSoFar.functionCalls := inc(numbersSoFar.functionCalls)$INT} \\
\text{numbersSoFar := numberOfOperations1(a, numbersSoFar)} \\
\text{numbersSoFar} \\
\end{cases}
\]

\[
\text{numberOfOperations(ode: VEDF): ON} = \begin{cases} \\
n: ON := [0, 0, 0, 0] \\
\text{for i in 1..#ode repeat} \\
n: ON := numberOfOperations1(ode.i, n) \\
n \\
\end{cases}
\]

\[
\text{expenseOfEvaluation(o: VEDF): F} = \begin{cases} \\
n: ON := numberOfOperations(o) \\
a := \text{ln.additions} \\
m := \text{ln.multiplications} \\
e := \text{ln.exponentiations} \\
f := 10*\text{ln.functionCalls} \\
n := (a + m + 4*e + 10*e) \\
(1.0 - \exp((-n:: F/288.0)$F)
\end{cases}
\]
concat(a: Result, b: Result): Result ==
members0fa := (members(a)@List(Record(key: Symbol, entry: Any)))
members0fb := (members(b)@List(Record(key: Symbol, entry: Any)))
allMembers:=
  concat(members0fa, members0fb)$List(Record(key: Symbol, entry: Any))
construct(allMembers)

concat(l: List Result): Result ==
import List Result
empty? l => empty()$Result
f := first l
if empty?(r := rest l) then
  f
else
  concat(f, concat r)

outputMeasure(m:F): ST ==
  fl: Float := round(m*(f := 1000.0))/f
  convert(fl)@ST

measure2Result(m: Measure): Result ==
  mm := coercem.measure$AnyFunctions1(Float)
  mnr: Record(key: Symbol, entry: Any) := [bestMeasure@Symbol, mm]
  mn := coercem.name$AnyFunctions1(ST)
  mnr: Record(key: Symbol, entry: Any) := [nameOfRoutine@Symbol, mn]
  me := coercem.explanations$AnyFunctions1(List String)
  mer: Record(key: Symbol, entry: Any) := [allMeasures@Symbol, me]
  mr := construct([mmr, mnr, mer])$Result
  met := coerce(mr)$AnyFunctions1(Result)
  meth: Record(key: Symbol, entry: Any):=[method@Symbol, met]
  construct([meth])$Result

measure2Result(m: Measure2): Result ==
  mm := coercem.measure$AnyFunctions1(Float)
  mnr: Record(key: Symbol, entry: Any) := [bestMeasure@Symbol, mm]
  mn := coercem.name$AnyFunctions1(ST)
  mnr: Record(key: Symbol, entry: Any) := [nameOfRoutine@Symbol, mn]
  me := coercem.explanations$AnyFunctions1(List String)
  mer: Record(key: Symbol, entry: Any) := [allMeasures@Symbol, me]
  mx := coercem.extra$AnyFunctions1(Result)
  mxr: Record(key: Symbol, entry: Any) := [other@Symbol, mx]
  mr := construct([mmr, mnr, mer, mxr])$Result
  met := coercemr$AnyFunctions1(Result)
  meth: Record(key: Symbol, entry: Any):=[method@Symbol, met]
  construct([meth])$Result

att2Result(att: ATT): Result ==
aepc := coercem endPointContinuity$AnyFunctions1(CTYPE)
ar := coercem range$AnyFunctions1(RTYPE)
as := coerce(att.singularitiesStream)$AnyFunctions1(STYPE)
aa:List Any := [aepc,ar,as]
aaa := coerce(aa)$AnyFunctions1(List Any)
aar:Record(key:Symbol,entry:Any) := [attributes@Symbol,aaa]
construct([aar])$Result

iflist2Result(ifv:IFV):Result ==
ifvs:List String :=
[concat(["stiffness: ",outputMeasure(ifv.stiffness)]),
 concat(["stability: ",outputMeasure(ifv.stability)]),
 concat(["expense: ",outputMeasure(ifv.expense)]),
 concat(["accuracy: ",outputMeasure(ifv.accuracy)]),
 concat(["intermediateResults: ",outputMeasure(ifv.intermediateResults)])]
ifa:= coerce(ifvs)$AnyFunctions1(List String)
ifr:Record(key:Symbol,entry:Any) := [intensityFunctions@Symbol,ifa]
construct([ifr])$Result

package ESTOOLS1 ExpertSystemToolsPackage1

— ExpertSystemToolsPackage1.input —

)set break resume
)sys rm -f ExpertSystemToolsPackage1.output
)spool ExpertSystemToolsPackage1.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show ExpertSystemToolsPackage1
--R
--R ExpertSystemToolsPackage1(R1: OrderedRing) is a package constructor
--R Abbreviation for ExpertSystemToolsPackage1 is ESTOOLS1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ESTOOLS1
--R
--R------------------------------------ Operations ------------------------------------
--R neglist : List(R1) -> List(R1)
--R
--E 1

)spool
)lisp (bye)

---

| ExpertSystemToolsPackage1.help |
---

===============================================
ExpertSystemToolsPackage1 examples
===============================================

ExpertSystemToolsPackage1 contains some useful functions for use by the computational agents of Ordinary Differential Equation solvers.

See Also:
o )show ExpertSystemToolsPackage1

---

ExpertSystemToolsPackage1 (ESTOOLS1)

Exports:
neglist

--- package ESTOOLS1 ExpertSystemToolsPackage1 ---
)abbrev package ESTOOLS1 ExpertSystemToolsPackage1
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: February 1995
++ Description:
++ \texttt{ExpertSystemToolsPackage1} contains some useful functions for use
++ by the computational agents of Ordinary Differential Equation solvers.

\begin{verbatim}
ExpertSystemToolsPackage1(R1: OR): E == I where
    OR ==> OrderedRing
    E ==> with
        neglist: List R1 -> List R1
        ++ neglist(l) returns only the negative elements of the list \texttt{l}
    I ==> add
        neglist(l: List R1): List R1 == [u for u in l | negative?(u)$R1]
\end{verbatim}

— ESTOOLS1.dotabb —

"ESTOOLS1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS1"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ESTOOLS1" -> "OAGROUP"

— package ESTOOLS2 ExpertSystemToolsPackage2 —

)set break resume
)sys rm -f ExpertSystemToolsPackage2.output
)spool ExpertSystemToolsPackage2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpertSystemToolsPackage2
--R
--R ExpertSystemToolsPackage2(R1: Ring, R2: Ring) is a package constructor
--R Abbreviation for ExpertSystemToolsPackage2 is ESTOOLS2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ESTOOLS2
--R
--R----------------------------- Operations -----------------------------
--R map : ((R1 -> R2),Matrix(R1)) -> Matrix(R2)
--R
--E 1

)spool
)lisp (bye)

— ExpertSystemToolsPackage2.help —

====================================================================
ExpertSystemToolsPackage2 examples
====================================================================

ExpertSystemToolsPackage2 contains some useful functions for use
by the computational agents of Ordinary Differential Equation solvers.

See Also:
o )show ExpertSystemToolsPackage2

— — — — —

ExpertSystemToolsPackage2 (ESTOOLS2)

Exports:
map

— package ESTOOLS2 ExpertSystemToolsPackage2 —

)abbrev package ESTOOLS2 ExpertSystemToolsPackage2
++ Author: Brian Dupee
++ Date Created: February 1995
++ Date Last Updated: July 1996
++ Description:
++ \texttt{\texttt{\texttt{a\texttt{xiom}\{ExpertSystemToolsPackage2\}}} contains some useful functions for use
++ by the computational agents of Ordinary Differential Equation solvers.

ExpertSystemToolsPackage2(R1:R,R2:R): E == I where
  R => Ring
  E => with
    map:(R1->R2,Matrix R1) -> Matrix R2
    ++ map(f,m) applies a mapping f:R1 -> R2 onto a matrix
    ++ \spad{m} in R1 returning a matrix in R2
  I => add
    map(f:R1->R2,m:Matrix R1):Matrix R2 ==
    matrix(([f u for u in v] for v in listOfLists(m)$(Matrix R1))$(Matrix R2)

———

| ESTOOLS2.dotabb |

"ESTOOLS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ESTOOLS2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"ESTOOLS2" -> "LMODULE"
"ESTOOLS2" -> "SGROUP"

———

package EXPR2 ExpressionFunctions2

—— ExpressionFunctions2.input ——

)set break resume
)sys rm -f ExpressionFunctions2.output
)spool ExpressionFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionFunctions2
--R
--R ExpressionFunctions2(R: OrderedSet,S: OrderedSet) is a package constructor
---R Abbreviation for ExpressionFunctions2 is EXPR2
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXPR2
---R
---R----------------------------------- Operations -----------------------------------
---R map : ((R -> S),Expression(R)) -> Expression(S)
---R
---E 1

)spool
)lisp (bye)

---

— ExpressionFunctions2.help —

================================================================================
ExpressionFunctions2 examples
================================================================================

Lifting of maps to Expressions.

See Also:
o )show ExpressionFunctions2

---

ExpressionFunctions2 (EXPR2)

--- package EXPR2 ExpressionFunctions2 ---
package EXPRSOL ExpressionSolve

Bugs

seriesSolve(sin f x / cos x, f, x, [1])$EXPRSOL(INT, EXPR INT, UFPS EXPR INT, UFPS SUPEXPR EXPR INT)

returns

(((0 . 1) 0 . 1) NonNullStream #<compiled-function STREAM;generate;M$;62!0> . UNPRINTABLE)

but
works. This is probably due to missing "/" in UFPS.

I'd really like to be able to specify a function that works for all domains in a category. For example, \( x \rightarrow y(x)^2 + \sin x + x \) should work for \( \text{EXPR INT} \) as well as for \( \text{UTS INT} \), both being domains having \( \text{TranscendentalFunctionCategory} \).

--- ExpressionSolve.input ---

)set break resume
)sys rm -f ExpressionSolve.output
)spool ExpressionSolve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionSolve
--R
--R ExpressionSolve(R: Join(OrderedSet,IntegralDomain,ConvertibleTo(InputForm)),F: FunctionSpace(R),UTSF: UnivariateTaylorSeriesCategory(F),UTSSUPF: UnivariateTaylorSeriesCategory(SparseUnivariatePolynomialExpressions(F))) is a package constructor
--R Abbreviation for ExpressionSolve is EXPRSOL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXPRSOL
--R
--R------------------------------- Operations --------------------------------
--R replaceDiffs : (F,BasicOperator,Symbol) -> F
--R seriesSolve : (F,BasicOperator,Symbol,List(F)) -> UTSF
--R

)spool
)lisp (bye)

--- ExpressionSolve.help ---

====================================================================
ExpressionSolve examples
====================================================================

This package has no description

See Also:
ExpressionSolve (EXPRSOL)

Exports:
  replaceDiffs  seriesSolve

— package EXPRSOL ExpressionSolve —

)abbrev package EXPRSOL ExpressionSolve
++ Description:
++ This package has no description

ExpressionSolve(R, F, UTSF, UTSSUPF): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
  F: FunctionSpace R
  UTSF: UnivariateTaylorSeriesCategory F
  SUP ==> SparseUnivariatePolynomialExpressions
  UTSSUPF: UnivariateTaylorSeriesCategory SUP F
  OP ==> BasicOperator
  SY ==> Symbol
  NNI ==> NonNegativeInteger
  MKF ==> MakeBinaryCompiledFunction(F, UTSSUPF, UTSSUPF, UTSSUPF)

Exports == with

  seriesSolve: (F, OP, SY, List F) -> UTSF
  replaceDiffs: (F, OP, Symbol) -> F

Implementation == add
\getchunk{implementation: EXPRSOL ExpressionSolve}
The general method is to transform the given expression into a form which can then be compiled. There is currently no other way in Axiom to transform an expression into a function.

We need to replace the differentiation operator by the corresponding function in the power series category, and make composition explicit. Furthermore, we need to replace the variable by the corresponding variable in the power series. It turns out that the compiler doesn’t find the right definition of monomial(1,1). Thus we introduce it as a second argument. In fact, maybe that’s even cleaner. Also, we need to tell the compiler that kernels that are independent of the main variable should be coerced to elements of the coefficient ring, since it will complain otherwise.

I cannot find an example for this behaviour right now. However, if I do use the coerce, the following fails:

\[
\text{seriesSolve}(h \cdot x^{-1-x} \cdot h(x \cdot x), h, x, [1])
\]

---

---

\text{getchunk\{implementation: EXPRSOL ExpressionSolve\}}

\text{opelt := operator("elt"::Symbol)\$OP}
\text{opdiff := operator("D"::Symbol)\$OP}
\text{opcoerce := operator("coerce"::Symbol)\$OP}

\text{-- replaceDiffs: (F, OP, Symbol) -> F}
\text{replaceDiffs (expr, op, sy) ==}
\text{lk := kernels expr}
\text{for k in lk repeat}
\text{-- if freeOf?(coerce k, sy) then}
\text{-- expr := subst(expr, [k], [opcoerce [coerce k]])}
\text{if is?\(k\), op) then}
\text{arg := first argument k}
\text{if arg = sy::F}
\text{then expr := subst(expr, [k], [(name op)::F])}
\text{else expr := subst(expr, [k], [opelt [(name op)::F,
\text{replaceDiffs(arg, op, sy)]])}}

\text{-- => "iterate"}

\text{if is?\(k\), %diff) then}
\text{args := argument k}
\text{differentiand :=
\text{replaceDiffs(subst(args.1, args.2 = args.3), op, sy)}}
\text{expr := subst(expr, [k], [opdiff differentiand])}

\text{-- => "iterate"}
package ES1 ExpressionSpaceFunctions1

expr

seriesSolve(expr, op, sy, l) ==
ex := replaceDiffs(expr, op, sy)
f := compiledFunction(ex, name op, sy)$MKF
seriesSolve(x+->f(x, monomial(1,1)$UTSSUPF), l) $\text{$\text{TaylorSolve}$}(F, UTSF, UTSSUPF)

---

package ES1 ExpressionSpaceFunctions1
CHAPTER 6. CHAPTER E

---

ExpressionSpaceFunctions1.help

====================================================================
ExpressionSpaceFunctions1 examples
====================================================================

This package allows a map from any expression space into any object
to be lifted to a kernel over the expression set, using a given
property of the operator of the kernel.

See Also:
o )show ExpressionSpaceFunctions1

---

ExpressionSpaceFunctions1 (ES1)

Exports:
map

--- package ES1 ExpressionSpaceFunctions1 ---

)abbrev package ES1 ExpressionSpaceFunctions1
++ Author: Manuel Bronstein
++ Date Created: 23 March 1988
++ Date Last Updated: 19 April 1991
++ Description:
++ This package allows a map from any expression space into any object
++ to be lifted to a kernel over the expression set, using a given
++ property of the operator of the kernel.
-- should not be exposed
ExpressionSpaceFunctions1(F:ExpressionSpace, S:Type): with
map: (F -> S, String, Kernel F) -> S
++ map(f, p, k) uses the property p of the operator
++ of k, in order to lift f and apply it to k.

== add
-- prop contains an evaluation function List S -> S
map(F2S, prop, k) ==
args := [F2S x for x in argument k]$List(S)
(p := property(operator k, prop)) case None =>
  ((p::None) pretend (List S -> S)) args
error "Operator does not have required property"

— ES1.dotabb —

"ES1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ES1"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"ES1" -> "EVALAB"

— ExpressionSpaceFunctions2.input —

)set break resume
)sys rm -f ExpressionSpaceFunctions2.output
)spool ExpressionSpaceFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionSpaceFunctions2
--R
--R ExpressionSpaceFunctions2(E: ExpressionSpace,F: ExpressionSpace) is a package constructor
--R Abbreviation for ExpressionSpaceFunctions2 is ES2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ES2
--R
--R------------------------------- Operations --------------------------------
This package allows a mapping $E \to F$ to be lifted to a kernel over $E$; this lifting can fail if the operator of the kernel cannot be applied in $F$. Do not use this package with $E = F$, since this may drop some properties of the operators.

See Also:
- `)show ExpressionSpaceFunctions2`
++ Author: Manuel Bronstein  
++ Date Created: 23 March 1988  
++ Date Last Updated: 19 April 1991  
++ Description:  
++ This package allows a mapping \( E \rightarrow F \) to be lifted to a kernel over \( E \);  
++ This lifting can fail if the operator of the kernel cannot be applied  
++ in \( F \); Do not use this package with \( E = F \), since this may  
++ drop some properties of the operators.

\[
\text{ExpressionSpaceFunctions2}(E:\text{ExpressionSpace}, \ F:\text{ExpressionSpace}): \text{with}
\]

\[
\text{map}: \ (E \rightarrow F, \ \text{Kernel \ E}) \rightarrow F
\]

++ \( \text{map}(f, k) \) returns \( \text{g = op}(f(a1), \ldots, f(an)) \) where  
++ \( \text{spad}(k = \text{op}(a1, \ldots, an)) \).

== add  
\[
\text{map}(f, k) ==
\]

(\text{operator}(\text{operator} k)\$F) \ [f \ \text{for} \ x \ \text{in} \ \text{argument} \ k]\$\text{List}(F)

——

— ES2.dotabb —

"ES2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ES2"]  
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]  
"ES2" \rightarrow "EVALAB"

——

package EXPRODE ExpressionSpaceODESolver

—— ExpressionSpaceODESolver.input ——

)set break resume
)sys rm -f ExpressionSpaceODESolver.output
)spool ExpressionSpaceODESolver.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionSpaceODESolver

--R  
--R ExpressionSpaceODESolver(\text{R: \ Join(OrderedSet,IntegralDomain,ConvertibleTo(InputForm))}, \ F:\ \text{FunctionSpace}(\text{R}))  
--R Abbreviation for ExpressionSpaceODESolver is \text{EXPRODE}  
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXPRODE
--R
--R---------------------------------------- Operations ----------------------------------------
--R seriesSolve : (Equation(F),BasicOperator,Equation(F),Equation(F)) -> Any
--R seriesSolve : (Equation(F),BasicOperator,Equation(F),List(F)) -> Any
--R seriesSolve : (List(Equation(F)),List(BasicOperator),Equation(F),List(Equation(F))) -> Any
--R seriesSolve : (List(Equation(F)),List(BasicOperator),Equation(F),List(F)) -> Any
--R seriesSolve : (List(F),List(BasicOperator),Equation(F),List(Equation(F))) -> Any
--R seriesSolve : (Equation(F),BasicOperator,Equation(F),F) -> Any
--R seriesSolve : (F,BasicOperator,Equation(F),Equation(F)) -> Any
--R seriesSolve : (F,BasicOperator,Equation(F),List(F)) -> Any
--R
--E 1

)spool
)lisp (bye)

—— ExpressionSpaceODESolver.help ——

====================================================================
ExpressionSpaceODESolver examples
====================================================================

Taylor series solutions of explicit ODE's;

See Also:
  o )show ExpressionSpaceODESolver
ExpressionSpaceODESolver (EXPRODE)

Exports:
seriesSolve

-- package EXPRODE ExpressionSpaceODESolver --

)abbrev package EXPRODE ExpressionSpaceODESolver
++ Author: Manuel Bronstein
++ Date Created: 5 Mar 1990
++ Date Last Updated: 30 September 1993
++ Description:
++ Taylor series solutions of explicit ODE’s;

ExpressionSpaceODESolver(R, F): Exports == Implementation where
  R: Join(OrderedSet, IntegralDomain,ConvertibleTo InputForm)
  F: FunctionSpace R
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
OP ==> BasicOperator
SY ==> Symbol
UTS ==> UnivariateTaylorSeries(F, x, center)
MKF ==> MakeUnaryCompiledFunction(F, UTS, UTS)
MKL ==> MakeUnaryCompiledFunction(F, List UTS, UTS)
A1 ==> AnyFunctions1(UTS)
AL1 ==> AnyFunctions1(List UTS)
EQ ==> Equation F
ODE ==> UnivariateTaylorSeriesODESolver(F, UTS)

Exports ==> with
  seriesSolve: (EQ, OP, EQ, EQ) -> Any
    + seriesSolve(eq,y,x=a, y a = b) returns a Taylor series solution
      + of eq around x = a with initial condition \spad{y(a) = b}.
    + Note that eq must be of the form
      + \spad{f(x, y x) y'(x) + g(x, y x) = h(x, y x)}.
  seriesSolve: (EQ, OP, EQ, List F) -> Any
CHAPTER 6. CHAPTER E

++ seriesSolve(eq, y, x=a, [b0, ..., b(n-1)]) returns a Taylor series
++ solution of eq around \spad{x = a} with initial conditions
++ \spad{y(a) = b0}, \spad{y'(a) = b1},
++ \spad{y''(a) = b2}, ..., \spad{y(n-1)(a) = b(n-1)}
++ eq must be of the form
++ \spad{f(x, y x, y'(x), ..., y(n-1)(x)) y(n)(x) +
++ g(x, y x, y'(x), ..., y(n-1)(x)) = h(x, y x, y'(x), ..., y(n-1)(x))}

seriesSolve: (List EQ, List OP, EQ, List EQ) -> Any
++ returns a taylor series solution of \spad{\{eq1, ..., eqn\}} around
++ \spad{x = a} with initial conditions \spad{yi(a) = bi}.
++ Note that eqi must be of the form
++ \spad{fi(x, y1 x, y2 x, ..., yn x) y1'(x) +
++ gi(x, y1 x, y2 x, ..., yn x) = h(x, y1 x, y2 x, ..., yn x)}.

seriesSolve: (List EQ, List OP, EQ, List F) -> Any
++ is equivalent to
++ \spad{seriesSolve([eq1, ..., eqn], [y1, ..., yn], x = a, [y1 a = b1, ..., yn a = bn])}

seriesSolve: (List F, List OP, EQ, List F) -> Any
++ is equivalent to
++ \spad{seriesSolve([eq1=0, ..., eqn=0], [y1, ..., yn], x = a, [y1 a = b1, ..., yn a = bn])}

seriesSolve: (List F, List OP, EQ, List EQ) -> Any
++ is equivalent to
++ \spad{seriesSolve([eq1=0, ..., eqn=0], [y1, ..., yn], x = a, [y1 a = b1, ..., yn a = bn])}

Implementation ==> add
checkCompat: (OP, EQ, EQ) -> F
checkOrder1: (F, OP, K, SY, F) -> F
checkOrderN: (F, OP, K, SY, F, NonNegativeInteger) -> F
checkSystem: (F, List K, List F) -> F
div2exquo : F -> F
smp2exquo : P -> F
k2exquo : K -> F
diffRhs : (F, F) -> F
diffRhsK : (K, F) -> F
findCompat : (F, List EQ) -> F
findEq : (K, SY, List F) -> F
localInteger: F -> F

opelt := operator("elt"::Symbol)$OP
--opex := operator("exquo"::Symbol)$OP
opex := operator("fixedPointExquo"::Symbol)$OP
opint := operator("integer"::Symbol)$OP

Rint? := R has IntegerNumberSystem

localInteger n == (Rint? => n; opint n)
diffRhs(f, g) == diffRhsK(retract(f)@K, g)
k2exquo k ==
  is?(op := operator k, "%diff"::Symbol) =>
    error "Improper differential equation"
  kernel(op, [div2exquo f for f in argument k]$List(F))
smp2exquo p ==
  map(k2exquo,x+->x::F,p)$_PolynomialCategoryLifting(IndexedExponents K,K, R, P, F)
div2exquo f ==
  -- one?(d := denom f) => f
  ((d := denom f) = 1) => f
  opex(smp2exquo numer f, smp2exquo d)

-- if g is of the form a * k + b, then return -b/a
diffRhsK(k, g) ==
  h := univariate(g, k)
  (degree(numer h) <= 1) and ground? denom h =>
    - coefficient(numer h, 0) / coefficient(numer h, 1)
  error "Improper differential equation"
checkCompat(y, eqx, eqy) ==
  lhs(eqy) =$F y(rhs eqx) => rhs eqy
  error "Improper initial value"
findCompat(yx, 1) ==
  for eq in l repeat
    yx =$F lhs eq => return rhs eq
  error "Improper initial value"
findEq(k, x, sys) ==
  k := retract(differentiate(k::F, x))@K
  for eq in sys repeat
    member?(k, kernels eq) => return eq
error "Improper differential equation"

checkOrder1(diffeq, y, yx, x, sy) ==
  div2exquo subst(diffRhs(differentiate(yx::F,x),diffeq),[yx],[sy])

checkOrderN(diffeq, y, yx, x, sy, n) ==
  zero? n => error "No initial value(s) given"
  m := (minIndex(l := retract(f := yx::F)@K)$List(K)))::F
  lv := [opelt(sy, localInteger m)]$List(F)
  for i in 2..n repeat
    l := concat(retract(f := differentiate(f, x))@K, l)
    lv := concat(opelt(sy, localInteger(m := m + 1)), lv)
  div2exquo subst(diffRhs(differentiate(f, x), diffeq), 1, lv)

checkSystem(diffeq, yx, lv) ==
  for k in kernels diffeq repeat
    is?(x, "%diff":SY) =>
      return div2exquo subst(diffRhsK(k, diffeq), yx, lv)
    0

seriesSolve(l:List EQ, y:List OP, eqx:EQ, eqy:List EQ) ==
  seriesSolve([lhs deq - rhs deq for deq in l]$List(F), y, eqx, eqy)

seriesSolve(l:List EQ, y:List OP, eqx:EQ, y0:List F) ==
  seriesSolve([lhs deq - rhs deq for deq in l]$List(F), y, eqx, y0)

seriesSolve(l:List F, ly:List OP, eqx:EQ, eqy:List EQ) ==
  seriesSolve(l, ly, eqx,
    [findCompat(y rhs eqx, eqy) for y in ly]$List(F))

seriesSolve(diffeq:EQ, y:OP, eqx:EQ, eqy:EQ) ==
  seriesSolve(lhs diffeq - rhs diffeq, y, eqx, eqy)

seriesSolve(diffeq:EQ, y:OP, eqx:EQ, y0:F) ==
  seriesSolve(lhs diffeq - rhs diffeq, y, eqx, y0)

seriesSolve(diffeq:EQ, y:OP, eqx:EQ, y0:List F) ==
  seriesSolve(lhs diffeq - rhs diffeq, y, eqx, y0)

seriesSolve(diffeq:EQ, y:OP, eqx:EQ, eqy:EQ) ==
  seriesSolve(diffeq, y, eqx, checkCompat(y, eqx, eqy))

seriesSolve(diffeq:F, y:OP, eqx:EQ, y0:F) ==
  x := symbolIfCan(retract(lhs eqx)$OK)::SY
  sy := name y
  yx := retract(y lhs eqx)$OK
  f := checkOrder1(diffeq, y, yx, x, sy::F)
  center := rhs eqx
  coerce(ode1((compiledFunction(f, sy)$MKF, y0)$ODE)$A1}
seriesSolve(diffeq:F, y:OP, eqx:EQ, y0:List F) ==
x := symbolIfCan(retract(lhs eqx)@K)::SY
sy := new()$SY
yx := retract(y lhs eqx)@K
f := checkOrderN(diffeq, y, yx, x, sy::F, #y0)
center := rhs eqx
coerce(ode(compiledFunction(f, sy)$MKL, y0)$ODE)$A1

seriesSolve(sys:List F, ly:List OP, eqx:EQ, l0:List F) ==
x := symbolIfCan(kx := retract(lhs eqx)@K)::SY
fsy := (sy := new()$SY)::F
m := (minIndex(l0) - 1)::F
yx := concat(kx, [retract(y lhs eqx)@K for y in ly]$List(K))
lelt := [opelt(fsy, localInteger(m := m+1)) for k in yx]$List(F)
sys := [findEq(k, x, sys) for k in rest yx]
l := [checkSystem(eq, yx, lelt) for eq in sys]$List(F)
center := rhs eqx
coerce(mpsode(l0,[compiledFunction(f,sy)$MKL for f in l])$ODE)$A1

package OMEXPR ExpressionToOpenMath

ExpressionToOpenMath.input

)clear all

--S 1 of 1
)show ExpressionToOpenMath

--R
--R ExpressionToOpenMath(R: Join(OpenMath,OrderedSet,Ring)) is a package constructor
---R Abbreviation for ExpressionToOpenMath is OMEXPR
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for OMEXPR
---R
---R--------------------------------- Operations ---------------------------------
---R OMwrite : Expression(R) -> String
---R OMwrite : (Expression(R),Boolean) -> String
---R OMwrite : (OpenMathDevice,Expression(R)) -> Void
---R OMwrite : (OpenMathDevice,Expression(R),Boolean) -> Void
---R
---E 1

)spool
)lisp (bye)

ExpressionToOpenMath (OMEXPR)

ExpressionToOpenMath provides support for converting objects of type Expression into OpenMath.

See Also:
  o )show ExpressionToOpenMath

ExpressionToOpenMath examples
ExpressionToOpenMath examples
Exports:
OMwrite

-- package OMEXPR ExpressionToOpenMath --

)abbrev package OMEXPR ExpressionToOpenMath
++ Author: Mike Dewar & Vilya Harvey
++ Description:
++ \texttt{ExpressionToOpenMath} provides support for
++ converting objects of type \texttt{Expression} into OpenMath.

ExpressionToOpenMath(R: Join(OpenMath, OrderedSet, Ring)): with
OMwrite : Expression R -> String
OMwrite : (Expression R, Boolean) -> String
OMwrite : (OpenMathDevice, Expression R) -> Void
OMwrite : (OpenMathDevice, Expression R, Boolean) -> Void
== add
import Expression R
SymInfo ==> Record(cd:String, name:String)
import SymInfo
import Record(key: Symbol, entry: SymInfo)
import AssociationList(Symbol, SymInfo)
import OMENC

-----------------------------
-- Local translation tables.
-----------------------------

nullaryFunctionAList : AssociationList(Symbol, SymInfo) := construct [_.
[pi, ["nums1", "pi"]]]

unaryFunctionAList : AssociationList(Symbol, SymInfo) := construct [_.
[exp, ["transc1", "exp"]],_.
[log, ["transc1", "ln"]],_.
[sin, ["transc1", "sin"]],_.
[cos, ["transc1", "cos"]],_.
[tan, ["transc1", "tan"]],_.
[cot, ["transc1", "cot"]],_.
[sec, ["transc1", "sec"]],_.
[csc, ["transc1", "csc"]],_.
[asin, ["transc1", "arcsin"]],_.
[acos, ["transc1", "arccos"]],_.
[atan, ["transc1", "arctan"]],_.
[acot, ["transc1", "arccot"]],_.
[asec, ["transc1", "arcsec"]],_.
[acsc, ["transc1", "arccsc"]],_.
[sinh, ["transc1", "sinh"]],_.
[cosh, ["transc1", "cosh"]],_.
[tanh, ["transc1", "tanh"]],_.
]
[coth, ["transc1", "coth"]],
[sech, ["transc1", "sech"]],
[csch, ["transc1", "csch"]],
[asinh, ["transc1", "arcsinh"]],
[acosh, ["transc1", "arccosh"]],
[atanh, ["transc1", "arctanh"]],
[acoth, ["transc1", "arccoth"]],
[asech, ["transc1", "arcsech"]],
[acsch, ["transc1", "arccsch"]],
[factorial, ["integer1", "factorial"]],
[abs, ["arith1", "abs"]]

-- Still need the following unary functions:
-- digamma
-- Gamma
-- airyAi
-- airyBi
-- erf
-- Ei
-- Si
-- Ci
-- li
-- dilog

-- Still need the following binary functions:
-- Gamma(a, x)
-- Beta(x, y)
-- polygamma(k, x)
-- besselI(v, x)
-- besselY(v, x)
-- besselJ(v, x)
-- besselK(v, x)
-- permutation(n, m)
-- summation(x:%, n:Symbol) : as opposed to "definite" sum
-- product(x:%, n:Symbol) : ditto

-------------------------------
-- Forward declarations.
-------------------------------

outputOMExpr : (OpenMathDevice, Expression R) -> Void

-------------------------------
-- Local helper functions
-------------------------------

outputOMArith1(dev: OpenMathDevice, sym: String, args: List Expression R): Void ==
OMputApp(dev)
OMputSymbol(dev, "arith1", sym)
for arg in args repeat
outputOMLambda(dev: OpenMathDevice, ex: Expression R, var: Expression R): Void ==
OMputBind(dev)
OMputSymbol(dev, "fns1", "lambda")
OMputBVar(dev)
OMwrite(dev, var, false)
OMputEndBVar(dev)
OMwrite(dev, ex, false)
OMputEndBind(dev)

outputOMInterval(dev: OpenMathDevice, lo: Expression R, hi: Expression R): Void ==
OMputApp(dev)
OMputSymbol(dev, "interval1", "interval")
OMwrite(dev, lo, false)
OMwrite(dev, hi, false)
OMputEndApp(dev)

outputOMIntInterval(dev: OpenMathDevice, lo: Expression R, hi: Expression R): Void ==
OMputApp(dev)
OMputSymbol(dev, "interval1", "integer__interval")
OMwrite(dev, lo, false)
OMwrite(dev, hi, false)
OMputEndApp(dev)

outputOMBinomial(dev: OpenMathDevice, args: List Expression R): Void ==
not #args=2 => error "Wrong number of arguments to binomial"
OMputApp(dev)
OMputSymbol(dev, "combinat1", "binomial")
for arg in args repeat
  OMwrite(dev, arg, false)
OMputEndApp(dev)

outputOMPower(dev: OpenMathDevice, args: List Expression R): Void ==
not #args=2 => error "Wrong number of arguments to power"
outputOMArith1(dev, "power", args)

outputOMDefsum(dev: OpenMathDevice, args: List Expression R): Void ==
#args ^= 5 => error "Unexpected number of arguments to a defsum"
OMputApp(dev)
OMputSymbol(dev, "arith1", "sum")
outputOMIntInterval(dev, args.4, args.5)
outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
OMputEndApp(dev)

outputOMDefprod(dev: OpenMathDevice, args: List Expression R): Void ==
#args ^= 5 => error "Unexpected number of arguments to a defprod"
OMputApp(dev)
OMputSymbol(dev, "arith1", "product")
outputOMIntInterval(dev, args.4, args.5)
outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
OMputEndApp(dev)

outputOMDefint(dev: OpenMathDevice, args: List Expression R): Void ==
#args ^= 5 => error "Unexpected number of arguments to a defint"
OMputApp(dev)
OMputSymbol(dev, "calculus1", "defint")
outputOMInterval(dev, args.4, args.5)
outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
OMputEndApp(dev)

outputOMInt(dev: OpenMathDevice, args: List Expression R): Void ==
#args ^= 3 => error "Unexpected number of arguments to a defint"
OMputApp(dev)
OMputSymbol(dev, "calculus1", "int")
outputOMLambda(dev, eval(args.1, args.2, args.3), args.3)
OMputEndApp(dev)

outputOMFunction(dev: OpenMathDevice, op: Symbol, args: List Expression R): Void ==
nargs := #args
zero? nargs =>
omOp: Union(SymInfo, "failed") := search(op, nullaryFunctionAList)
omOp case "failed" =>
error concat ["No OpenMath definition for nullary function ", coerce op]
OMputSymbol(dev, omOp.cd, omOp.name)
-- one? nargs =>
(nargs = 1) =>
omOp: Union(SymInfo, "failed") := search(op, unaryFunctionAList)
omOp case "failed" =>
error concat ["No OpenMath definition for unary function ", coerce op]
OMputApp(dev)
OMputSymbol(dev, omOp.cd, omOp.name)
for arg in args repeat
OMwrite(dev, arg, false)
OMputEndApp(dev)
-- Most of the binary operators cannot be handled trivialy like the
-- unary ones since they have bound variables of one kind or another.
-- The special functions should be straightforward, but we don’t have
-- a CD for them yet :-)
op = %defint => outputOMDefint(dev, args)
op = integral => outputOMInt(dev, args)
op = %defsum => outputOMDefsum(dev, args)
op = %defprod => outputOMDefprod(dev, args)
op = %power => outputOMPower(dev, args)
op = binomial => outputOMBinomial(dev, args)
error concat ["No OpenMath definition for function ", string op]

outputOMExpr(dev: OpenMathDevice, ex: Expression R): Void ==
ground? ex => OMwrite(dev, ground ex, false)
not((v := retractIfCan(ex)@Union(Symbol,"failed")) case "failed") =>
OMputVariable(dev, v)
not((v := isPlus ex) case "failed") => outputOMArith1(dev, "plus", w)
not((v := isTimes ex) case "failed") => outputOMArith1(dev, "times", w)
--not((y := isMult ex) case "failed") =>
-- outputOMArith("times", [OMwrite(y.coef)$Integer,
-- OMwrite(coerce y.var)])
-- At the time of writing we don’t need both isExpt and isPower
-- here but they may be relevant when we integrate this stuff into
-- the main Expression code. Note that if we don’t check that
-- the exponent is non-trivial we get thrown into an infinite recursion.
not (((x := isExpt ex) case "failed") or (x.exponent = 1)) =>
not (((z := isPower ex) case "failed") or (z.exponent = 1)) =>
outputOMPower(dev, [ z.val, z.exponent::Expression R ])
--OMputApp(dev)
--OMputSymbol(dev, "arith1", "power")
--outputOMExpr(dev, z.val)
--OMputInteger(dev, z.exponent)
--OMputEndApp(dev)
-- TODO: add error handling code here...

OMwrite(ex: Expression R): String ==
s: String := ""
sp := OM_-STRINGTOSTRINGPTR(s)$Lisp
dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML())
OMputObject(dev)
outputOMExpr(dev, ex)
OMputEndObject(dev)
OMclose(dev)
s := OM_-STRINGPTRTOSTRING(sp)$Lisp pretend String
s

OMwrite(ex: Expression R, wholeObj: Boolean): String ==
s: String := ""
sp := OM_STRINGTOSTRINGPTR(s)$Lisp
dev: OpenMathDevice := OMopenString(sp pretend String, OMencodingXML())
if wholeObj then
  OMputObject(dev)
outputOMExpr(dev, ex)
if wholeObj then
  OMputEndObject(dev)
OMclose(dev)
s := OM_STRINGPTRTOSTRING(sp)$Lisp pretend String
s

OMwrite(dev: OpenMathDevice, ex: Expression R): Void ==
  OMputObject(dev)
outputOMExpr(dev, ex)
OMputEndObject(dev)

OMwrite(dev: OpenMathDevice, ex: Expression R, wholeObj: Boolean): Void ==
  if wholeObj then
    OMputObject(dev)
  outputOMExpr(dev, ex)
  if wholeObj then
    OMputEndObject(dev)

package EXPR2UPS ExpressionToUnivariatePowerSeries

— ExpressionToUnivariatePowerSeries.input —

)set break resume
)sys rm -f ExpressionToUnivariatePowerSeries.output
)spool ExpressionToUnivariatePowerSeries.output
)set message test on
)set message auto off
)clear all
This package provides functions to convert functional expressions to power series.

See Also:
- )show ExpressionToUnivariatePowerSeries
ExpressionToUnivariatePowerSeries (EXPR2UPS)

Exports:
laurent puiseux series taylor

Abbrev package EXPR2UPS ExpressionToUnivariatePowerSeries
++ Author: Clifton J. Williamson
++ Date Created: 9 May 1989
++ Date Last Updated: 20 September 1993
++ Description:
++ This package provides functions to convert functional expressions
++ to power series.

ExpressionToUnivariatePowerSeries(R,FE): Exports == Implementation where
 R : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
     LinearityExplicitRingOver Integer)
 FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
     FunctionSpace R)

EQ  => Equation
I   => Integer
NNI => NonNegativeInteger
RN  => Fraction Integer
SY  => Symbol
UTS => UnivariateTaylorSeries
ULS => UnivariateLaurentSeries
UPXS => UnivariatePuiseuxSeries
GSER => GeneralUnivariatePowerSeries
EFULS => ElementaryFunctionsUnivariateLaurentSeries
EFUPXS => ElementaryFunctionsUnivariatePuiseuxSeries
FS2UPS => FunctionSpaceToUnivariatePowerSeries
Prob => Record(func:String,prob:String)
ANY1 => AnyFunctions1

Exports => with
taylor: SY -> Any
++ \spad{taylor(x)} returns x viewed as a Taylor series.
taylor: FE -> Any
++ \spad{taylor(f)} returns a Taylor expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable.
taylor: (FE,NNI) -> Any
++ \spad{taylor(f,n)} returns a Taylor expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least n.
taylor: (FE,EQ FE) -> Any
++ \spad{taylor(f,x = a)} expands the expression f as a Taylor series
++ in powers of \spad{(x - a)}.
taylor: (FE,EQ FE,NNI) -> Any
++ \spad{taylor(f,x = a)} expands the expression f as a Taylor series
++ in powers of \spad{(x - a)}; terms will be computed up to order
++ at least n.

laurent: SY -> Any
++ \spad{laurent(x)} returns x viewed as a Laurent series.
laurent: FE -> Any
++ \spad{laurent(f)} returns a Laurent expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable.
laurent: (FE,I) -> Any
++ \spad{laurent(f,n)} returns a Laurent expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least n.
laurent: (FE,EQ FE) -> Any
++ \spad{laurent(f,x = a)} expands the expression f as a Laurent series
++ in powers of \spad{(x - a)}.
laurent: (FE,EQ FE,I) -> Any
++ \spad{laurent(f,x = a,n)} expands the expression f as a Laurent
++ series in powers of \spad{(x - a)}; terms will be computed up to order
++ at least n.
puisseux: SY -> Any
++ \spad{puiseux(x)} returns x viewed as a Puiseux series.
puisseux: FE -> Any
++ \spad{puiseux(f)} returns a Puiseux expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable.
puisseux: (FE,RN) -> Any
++ \spad{puiseux(f,n)} returns a Puiseux expansion of the expression f.
++ Note that f should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least n.
puisseux: (FE,EQ FE) -> Any
++ \spad{puiseux(f,x = a)} expands the expression f as a Puiseux series
++ in powers of \( \spad{(x - a)} \).

puiseux: (FE, EQ FE, RN) \rightarrow \text{Any}
++ \( \spad{\text{puiseux}(f, x = a, n)} \) expands the expression \( f \) as a Puiseux
++ series in powers of \( \spad{(x - a)} \); terms will be computed up to order
++ at least \( n \).

series: SY \rightarrow \text{Any}
++ \( \spad{\text{series}(x)} \) returns \( x \) viewed as a series.

series: FE \rightarrow \text{Any}
++ \( \spad{\text{series}(f)} \) returns a series expansion of the expression \( f \).
++ Note that \( f \) should have only one variable; the series will be
++ expanded in powers of that variable.

series: (FE, RN) \rightarrow \text{Any}
++ \( \spad{\text{series}(f, n)} \) returns a series expansion of the expression \( f \).
++ Note that \( f \) should have only one variable; the series will be
++ expanded in powers of that variable and terms will be computed
++ up to order at least \( n \).

series: (FE, EQ FE) \rightarrow \text{Any}
++ \( \spad{\text{series}(f, x = a)} \) expands the expression \( f \) as a series
++ in powers of \( (x - a) \).

series: (FE, EQ FE, RN) \rightarrow \text{Any}
++ \( \spad{\text{series}(f, x = a, n)} \) expands the expression \( f \) as a series
++ in powers of \( (x - a) \); terms will be computed up to order
++ at least \( n \).

Implementation \( \Rightarrow \) add
performSubstitution: (FE, SY, FE) \rightarrow \text{FE}
performSubstitution(fcn, x, a) ==
  zero? a => fcn
  xFE := x :: FE
  eval(fcn, xFE = xFE + a)

iTaylor: (FE, SY, FE) \rightarrow \text{Any}
iTaylor(fcn, x, a) ==
  pack := FS2UPS(R, FE, I, ULS(FE, x, a),
               EFULS(FE, UTS(FE, x, a), ULS(FE, x, a)), x)
  ans := exprToUPS(fcn, false, "just do it")$pack
  ans case %problem =>
    ans.%problem.prob = "essential singularity" =>
      error "No Taylor expansion: essential singularity"
    ans.%problem.func = "log" =>
      error "No Taylor expansion: logarithmic singularity"
    ans.%problem.func = "nth root" =>
      error "No Taylor expansion: fractional powers in expansion"
      error "No Taylor expansion"
  uls := ans.%series
  (uts := taylorIfCan uls) case "failed" =>
    error "No Taylor expansion: pole"
  any1 := ANY1(UTS(FE, x, a))
  coerce(uts :: UTS(FE, x, a))$any1
taylor(x:SY) ==
  uts := UTS(FE,x,0$FE); any1 := ANY1(uts)
  coerce(monomial(1,1)$uts)$any1

taylor(fcn:FE) ==
  null(vars := variables fcn) =>
    error "taylor: expression has no variables"
  not null rest vars =>
    error "taylor: expression has more than one variable"
  taylor(fcn,(first(vars) :: FE) = 0)

taylor(fcn:FE,n:NNI) ==
  null(vars := variables fcn) =>
    error "taylor: expression has no variables"
  not null rest vars =>
    error "taylor: expression has more than one variable"
  x := first vars
  uts := UTS(FE,x,0$FE); any1 := ANY1(uts)
  series := retract(taylor(fcn,(x :: FE) = 0))$any1
  coerce(extend(series,n))$any1

taylor(fcn:FE,eq:EQ FE) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  iTaylor(performSubstitution(fcn,x,a),x,a)

taylor(fcn,eq,n) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  any1 := ANY1(UTS(FE,x,a))
  series := retract(iTaylor(performSubstitution(fcn,x,a),x,a))$any1
  coerce(extend(series,n))$any1

iLaurent: (FE,SY,FE) -> Any
iLaurent(fcn,x,a) ==
  pack := FS2UPS(R,FE,I,ULS(FE,x,a),-
    EFULS(FE,UTS(FE,x,a),ULS(FE,x,a)),x)
  ans := exprToUPS(fcn,false,"just do it")$pack
  ans case %problem =>
    ans.%problem.prob = "essential singularity" =>
      error "No Laurent expansion: essential singularity"
    ans.%problem.func = "log" =>
      error "No Laurent expansion: logarithmic singularity"
    ans.%problem.func = "nth root" =>
      error "No Laurent expansion: fractional powers in expansion"
    error "No Laurent expansion"
  any1 := ANY1(ULS(FE,x,a))
coerce(ans.%series)$any1

laurent(x:SY) ==
uls := ULS(FE,x,0$FE); any1 := ANY1(uls)
coerce(monomial(1,1)$uls)$any1

laurent(fcn:FE) ==
null(vars := variables fcn) =>
   error "laurent: expression has no variables"
not null rest vars =>
   error "laurent: expression has more than one variable"
laurent(fcn,(first(vars) :: FE) = 0)

laurent(fcn:FE,n:1) ==
null(vars := variables fcn) =>
   error "laurent: expression has no variables"
not null rest vars =>
   error "laurent: expression has more than one variable"
x := first vars
uls := ULS(FE,x,0$FE); any1 := ANY1(uls)
series := retract(laurent(fcn,(x :: FE) = 0))$any1
coerce(extend(series,n))$any1

laurent(fcn:FE,eq:EQ FE) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
   error "taylor: left hand side must be a variable"
x := xx :: SY; a := rhs eq
iLaurent(performSubstitution(fcn,x,a),x,a)

laurent(fcn,eq,n) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
   error "taylor: left hand side must be a variable"
x := xx :: SY; a := rhs eq
any1 := ANY1(ULS(FE,x,a))
series := retract(iLaurent(performSubstitution(fcn,x,a),x,a))$any1
coerce(extend(series,n))$any1

iPuiseux: (FE,SY,FE) -> Any
iPuiseux(fcn,x,a) ==
pack := FS2UPS(R,FE,RN,UPXS(FE,x,a),_
EFUPXS(FE,ULS(FE,x,a),UPXS(FE,x,a),_
EFULS(FE,UTS(FE,x,a),ULS(FE,x,a))),x)
ans := exprToUPS(fcn,false,"just do it")$pack
ans case %problem =>
an.sproblem.prob = "essential singularity" =>
   error "No Puiseux expansion: essential singularity"
an.sproblem.func = "log" =>
   error "No Puiseux expansion: logarithmic singularity"
error "No Puiseux expansion"
any1 := ANY1(UPXS(FE,x,a))
coerce(ans.%series)$any1

puiseux(x:SY) ==
    upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
    coerce(monomial(1,1)$upxs)$any1

puiseux(fcn:FE) ==
    null(vars := variables fcn) =>
        error "puiseux: expression has no variables"
    not null rest vars =>
        error "puiseux: expression has more than one variable"
    puiseux(fcn,(first(vars) :: FE) = 0)

puiseux(fcn:FE,n:RN) ==
    null(vars := variables fcn) =>
        error "puiseux: expression has no variables"
    not null rest vars =>
        error "puiseux: expression has more than one variable"
    x := first vars
    upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
    series := retract(puiseux(fcn,(x :: FE) = 0))$any1
    coerce(extend(series,n))$any1

puiseux(fcn:FE,eq:EQ FE) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
        error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    iPuiseux(performSubstitution(fcn,x,a),x,a)

puiseux(fcn,eq,n) ==
    (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
        error "taylor: left hand side must be a variable"
    x := xx :: SY; a := rhs eq
    any1 := ANY1(UPXS(FE,x,a))
    series := retract(iPuiseux(performSubstitution(fcn,x,a),x,a))$any1
    coerce(extend(series,n))$any1

iSeries: (FE,SY,FE) -> Any
iSeries(fcn,x,a) ==
    pack := FS2UPS(R,FE,RN,UPXS(FE,x,a), _
        EFUPXS(FE,ULS(FE,x,a),UPXS(FE,x,a), _
        EFULS(FE,UTS(FE,x,a),ULS(FE,x,a))),x)
    ans := exprToUPS(fcn,false,"just do it")$pack
    ans case %problem =>
        ansG := exprToGenUPS(fcn,false,"just do it")$pack
        ansG case %problem =>
            ansG.%problem.prob = "essential singularity" =>
                error "No series expansion: essential singularity"
            error "No series expansion"
    anyone := ANY1(GSER(FE,x,a))
coerce((ansG.%series) :: GSER(FE,x,a))$any1
any1 := ANY1(UPXS(FE,x,a))
coerce(ans.%series)$any1

series(x:SY) ==
upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
coerce(monomial(1,1)$upxs)$any1

series(fcn:FE) ==
null(vars := variables fcn) =>
error "series: expression has no variables"
not null rest vars =>
error "series: expression has more than one variable"
series(fcn,(first(vars) :: FE) = 0)

series(fcn:FE,n:RN) ==
null(vars := variables fcn) =>
error "series: expression has no variables"
not null rest vars =>
error "series: expression has more than one variable"
x := first vars
upxs := UPXS(FE,x,0$FE); any1 := ANY1(upxs)
series := retract(series(fcn,(x :: FE) = 0))$any1
coerce(extend(series,n))$any1

series(fcn:FE,eq:EQ FE) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
error "taylor: left hand side must be a variable"
x := xx :: SY; a := rhs eq
iSeries(performSubstitution(fcn,x,a),x,a)

series(fcn,eq,n) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
error "taylor: left hand side must be a variable"
x := xx :: SY; a := rhs eq
any1 := ANY1(UPXS(FE,x,a))
series := retract(iSeries(performSubstitution(fcn,x,a),x,a))$any1
coerce(extend(series,n))$any1

——

— EXPR2UPS.dotabb —

"EXPR2UPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXPR2UPS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"EXPR2UPS" -> "ULSCCAT"
package EXPRTUBE ExpressionTubePlot

— ExpressionTubePlot.input —

)set break resume
)sys rm -f ExpressionTubePlot.output
)spool ExpressionTubePlot.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ExpressionTubePlot
--R
)--R ExpressionTubePlot  is a package constructor
)--R Abbreviation for ExpressionTubePlot is EXPRTUBE
)--R This constructor is not exposed in this frame.
)--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXPRTUBE
--)R
)--R-------------------------------------------- Operations --------------------------------------------
)--R constantToUnaryFunction : DoubleFloat -> (DoubleFloat -> DoubleFloat)
)--R tubePlot : (Expression(Integer),Expression(Integer),Expression(Integer),(DoubleFloat -> DoubleFloat),Segment(DoubleFloat),(DoubleFloat -> DoubleFloat),Integer) -> TubePlot(Plot3D)
)--R tubePlot : (Expression(Integer),Expression(Integer),Expression(Integer),(DoubleFloat -> DoubleFloat),Segment(DoubleFloat),DoubleFloat,Integer) -> TubePlot(Plot3D)
)--R tubePlot : (Expression(Integer),Expression(Integer),Expression(Integer),(DoubleFloat -> DoubleFloat),Segment(DoubleFloat),DoubleFloat,Integer,String) -> TubePlot(Plot3D)

--)E 1

)spool
)lisp (bye)

———

— ExpressionTubePlot.help —

==================================
ExpressionTubePlot examples
==================================

Package for constructing tubes around 3-dimensional parametric curves.

See Also:
o )show ExpressionTubePlot

———
ExpressionTubePlot (EXPRTUBE)

Exports:
- constantToUnaryFunction
- tubePlot

--- package EXPRTUBE ExpressionTubePlot ---

`expressionsTubePlot()`:: Exports == Implementation where

- B ==> Boolean
- I ==> Integer
- FE ==> Expression Integer
- SY ==> Symbol
- SF ==> DoubleFloat
- L ==> List
- S ==> String
- SEG ==> Segment
- F2F ==> MakeFloatCompiledFunction(FE)
- Pt ==> Point SF
- PLOT3 ==> Plot3D
- TUBE ==> TubePlot Plot3D

Exports == with

- constantToUnaryFunction: SF -> (SF -> SF)
  ++ constantToUnaryFunction(s) is a local function which takes the
  ++ value of s, which may be a function of a constant, and returns
  ++ a function which always returns the value \spad{DoubleFloat} s.
- tubePlot: (FE,FE,FE,SF -> SF,SEG SF,SF -> SF,I) -> TUBE
  ++ tubePlot(f,g,h,colorFcn,a..b,r,n) puts a tube of radius r(t) with
  ++ n points on each circle about the curve \spad{x = f(t)},
  ++ \spad{y = g(t)}, \spad{z = h(t)} for \spad{t in [a,b]}.```
++ The tube is considered to be open.
tubePlot: (FE,FE,FE,SF -> SF,SEG SF, SF -> SF,I,S) -> TUBE
++ tubePlot(f,g,h,colorFcn,a..b,r,n,s) puts a tube of radius \spad{r(t)}
++ with n points on each circle about the curve \spad{x = f(t)},
++ \spad{y = g(t)}, \spad{z = h(t)} for t in \spad{[a,b]}. If s = "closed", the tube is
++ considered to be closed; if s = "open", the tube is considered
++ to be open.
tubePlot: (FE,FE,FE,SE, SF -> SF,SEG SF, SF,SF,I,S) -> TUBE
++ tubePlot(f,g,h,colorFcn,a..b,r,n) puts a tube of radius r with
++ n points on each circle about the curve \spad{x = f(t)},
++ \spad{y = g(t)}, \spad{z = h(t)} for t in \spad{[a,b]}. The tube is considered to be open.
tubePlot: (FE,FE,FE,SF -> SF,SEG SF, SF,SF,I,S) -> TUBE
++ tubePlot(f,g,h,colorFcn,a..b,r,n,s) puts a tube of radius r with
++ n points on each circle about the curve \spad{x = f(t)},
++ \spad{y = g(t)}, \spad{z = h(t)} for t in \spad{[a,b]}. If s = "closed", the tube is
++ considered to be closed; if s = "open", the tube is considered
++ to be open.

Implementation ==> add
import Plot3D
import F2F
import TubePlotTools

--% variables
getVariable: (FE,FE,FE) -> SY
getVariable(x,y,z) ==
  varList1 := variables x
  varList2 := variables y
  varList3 := variables z
  (not (# varList1 <= 1)) or (not (# varList2 <= 1)) or _
  (not (# varList3 <= 1)) =>
    error "tubePlot: only one variable may be used"
null varList1 =>
  null varList2 =>
  null varList3 =>
    error "tubePlot: a variable must appear in functions"
  first varList3
  t2 := first varList2
  null varList3 => t2
  not (first varList3 = t2) =>
    error "tubePlot: only one variable may be used"
  t1 := first varList1
  null varList2 =>
  null varList3 => t1
  not (first varList3 = t1) =>
    error "tubePlot: only one variable may be used"
t1
  t2 := first varList2
null varList3 =>
  not (t1 = t2) =>
    error "tubePlot: only one variable may be used"
  t1
not (first varList3 = t1) or not (t2 = t1) =>
  error "tubePlot: only one variable may be used"
  t1

--% tubes: variable radius

tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
  tRange:SEG SF,radFcn:SF -> SF,n:I,string:S) ==
  -- check value of n
  n < 3 => error "tubePlot: n should be at least 3"
  -- check string
  flag : B :=
    string = "closed" => true
    string = "open" => false
    error "tubePlot: last argument should be open or closed"
  -- check variables
  t := getVariable(x,y,z)
  -- coordinate functions
  xFunc := makeFloatFunction(x,t)
  yFunc := makeFloatFunction(y,t)
  zFunc := makeFloatFunction(z,t)
  -- derivatives of coordinate functions
  xp := differentiate(x,t)
  yp := differentiate(y,t)
  zp := differentiate(z,t)
  -- derivative of arc length
  sp := sqrt(xp ** 2 + yp ** 2 + zp ** 2)
  -- coordinates of unit tangent vector
  Tx := xp/sp; Ty := yp/sp; Tz := zp/sp
  -- derivatives of coordinates of unit tangent vector
  Txp := differentiate(Tx,t)
  Typ := differentiate(Ty,t)
  Tzp := differentiate(Tz,t)
  -- K = curvature = length of curvature vector
  K := sqrt(Txp ** 2 + Typ ** 2 + Tzp ** 2)
  -- coordinates of principal normal vector
  Nx := Txp / K; Ny := Typ / K; Nz := Tzp / K
  -- functions SF->SF giving coordinates of principal normal vector
  NxFunc := makeFloatFunction(Nx,t);
  NyFunc := makeFloatFunction(Ny,t);
  NzFunc := makeFloatFunction(Nz,t);
  -- coordinates of binormal vector
  Ex := Ty * Nz - Tz * Ny
  By := Tz * Nx - Tx * Nz
Bz := Tx * Ny - Ty * Nx

functions SF -> SF giving coordinates of binormal vector

BxFunc := makeFloatFunction(Bx,t);
ByFunc := makeFloatFunction(By,t);
BzFunc := makeFloatFunction(Bz,t);

-- create Plot3D
parPlot := plot(xFunc,yFunc,zFunc,colorFcn,tRange)
tvals := first tValues parPlot
curvePts := first listBranches parPlot
cosSin := cosSinInfo n

loopList : L L Pt := nil()
while not null tvals repeat

-- note that tvals and curvePts have the same number of elements
tval := first tvals; tvals := rest tvals
ctr := first curvePts; curvePts := rest curvePts

pNormList : L SF :=
  [NxFunc tval,NyFunc tval,NzFunc tval,colorFcn tval]
pNorm : Pt := point pNormList
bNormList : L SF :=
  [BxFunc tval,ByFunc tval,BzFunc tval,colorFcn tval]
bNorm : Pt := point bNormList

lps := loopPoints(ctr,pNorm,bNorm,radFcn tval,cosSin)
loopList := cons(lps,loopList)
tube(parPlot,reverse_! loopList,flag)

tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
tRange:SEG SF,radFcn:SF -> SF,n:I) ==
tubePlot(x,y,z,colorFcn,tRange,radFcn,n,"open")

project: (SF,SF) -> SF
project(x,y) == x

constantToUnaryFunction x == s +-> project(x,s)

tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
tRange:SEG SF,rad:SF,n:I,s:S) ==
tubePlot(x,y,z,colorFcn,tRange,constantToUnaryFunction rad,n,s)

tubePlot(x:FE,y:FE,z:FE,colorFcn:SF -> SF,_
tRange:SEG SF,rad:SF,n:1) ==
tubePlot(x,y,z,colorFcn,tRange,rad,n,"open")
package EXP3D Export3D

--- Export3D.input ---

)set break resume
)sys rm -f Export3D.output
)spool Export3D.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Export3D
--R
--R Export3D is a package constructor
--R Abbreviation for Export3D is EXP3D
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for EXP3D
--R
--R-------------------------------- Operations --------------------------------
--R writeObj : (SubSpace(3,DoubleFloat),String) -> Void
--R
--E 1

)spool
)lisp (bye)

---

--- Export3D.help ---

====================================================================
Export3D examples
====================================================================

This package provides support for exporting SubSpace and ThreeSpace structures to files.

See Also:
Export3D (EXP3D)

Exports:
writeObj

--- package EXP3D Export3D ---

)abbrev package EXP3D Export3D
++ Author: Martin Baker
++ Date: June, 2010
++ Description:
++ This package provides support for exporting SubSpace and
++ ThreeSpace structures to files.

EF ==> Expression Float
SBF ==> SegmentBinding Float
DF ==> DoubleFloat
I ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
STR ==> String

Export3D(): with

  writeObj:(SubSpace(3,DoubleFloat),String) -> Void
    ++ writes 3D SubSpace to a file in Wavefront (.OBJ) format

== add
import List List NNI
-- return list of indexes
-- assumes subnodes are leaves containing index
faceIndex(subSp: SubSpace(3,DoubleFloat)):List NNI ==
   faceIndexList:List NNI := []
   for poly in children(subSp) repeat
      faceIndexList := cons(extractIndex(poly),faceIndexList)
   reverse faceIndexList

-- called if this component contains a single polygon
-- write out face information for Wavefront (.OBJ) 3D file format
-- one face per line, represented by list of vertex indexes
writePolygon(f1:TextFile,curves: List SubSpace(3,DoubleFloat)):Void ==
   faceIndexList:List NNI := []
   for curve in curves repeat
      faceIndexList := append(faceIndexList,faceIndex(curve))
   s:String := "f 
   for i in faceIndexList repeat
      s:=concat(s,string(i))$String
      s:=concat(s," ")$String
   writeLine!(f1,s)

-- called if this component contains a mesh, the mesh will be rendered
-- as quad polygons.
-- write out face information for Wavefront (.OBJ) 3D file format
-- one face per line, represented by list of vertex indexes
writeMesh(f1:TextFile,curves: List SubSpace(3,DoubleFloat)):Void ==
   meshIndexArray:List List NNI := []
   for curve in curves repeat
      meshIndexArray := cons(faceIndex(curve),meshIndexArray)
   meshIndexArray := reverse meshIndexArray
   rowLength := #meshIndexArray
   colLength := #(meshIndexArray.1)
   for i in 1..(rowLength-1) repeat
      for j in 1..(colLength-1) repeat
         s1:String := concat["row ",string(i)," col ",string(j)]
         writeLine!(f1,s1)
         s:String := concat["f ",string((meshIndexArray.i).j)," ",
            string((meshIndexArray.(i+1)).j)," ",
            string((meshIndexArray.(i+1)).(j+1))," ",
            string((meshIndexArray.i).(j+1))]
         writeLine!(f1,s)
   toString(d : DoubleFloat) : String ==
      unparse(convert(d)@InputForm)
-- this writes SubSpace geometry to Wavefront (.OBJ) 3D file format
-- requires SubSpace to contain 3 or 4 dimensional points over DoubleFloat
-- to export a function plot try:
-- writeObj(subSpace(makeObject(x*x-y*y,x=-1..1,y=-1..1)),"myfile.obj")
-- colour dimension is ignored
-- no normals or texture data is generated
writeObj(subSp: SubSpace(3,DoubleFloat), filename:String):Void ==
  f1:TextFile:=open(filename::FileName,"output")
  writeLine!(f1,"# mesh generated by axiom")
-- write vertex data
  verts := pointData(subSp)
  for v in verts repeat
    #v < 3 => error "Can't write OBJ file from 2D points"
  writeLine!(f1,concat(
    ["v ", toString(v.1), " ",toString(v.2), " ", toString(v.3)]]$String)
  for component in children(subSp) repeat
    curves := children(component)
    if #curves < 2 then
      sayTeX$Lisp "Can't write point or curve to OBJ file"
      --writeLine!(f1,"new component")
    if #curves > 1 then
      if numberOfChildren(curves.1) = 1 then writePolygon(f1,curves)
      if numberOfChildren(curves.1) > 1 then writeMesh(f1,curves)
  close! f1

    — EXP3D.dotabb —

"EXP3D" [color="#FF4488",href="bookvol10.4.pdf#nameddest=EXP3D"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"EXP3D" -> "STRING"

    — package E04AGNT e04AgentsPackage —

    — e04AgentsPackage.input —

)set break resume
)sys rm -f e04AgentsPackage.output
)spool e04AgentsPackage.output
)set message test on
)set message auto off
>clear all

--S 1 of 1
>show e04AgentsPackage

--R
--R e04AgentsPackage is a package constructor
--R Abbreviation for e04AgentsPackage is E04AGNT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for E04AGNT
--R
--R--------------------------------- Operations --------------------------------
--R changeNameToObjf : (Symbol,Result) -> Result
--R expenseOfEvaluation : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> Float
--R finiteBound : (List(OrderedCompletion(DoubleFloat)),DoubleFloat) -> List(DoubleFloat)
--R linear? : List(Expression(DoubleFloat)) -> Boolean
--R linear? : Expression(DoubleFloat) -> Boolean
--R linearMatrix : (List(Expression(DoubleFloat)),NonNegativeInteger) -> Matrix(DoubleFloat)
--R linearPart : List(Expression(DoubleFloat)) -> List(Expression(DoubleFloat))
--R nonLinearPart : List(Expression(DoubleFloat)) -> List(Expression(DoubleFloat))
--R optAttributes : Union(noa: Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: ... Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat))) -> List(String)
--R quadratic? : Expression(DoubleFloat) -> Boolean
--R simpleBounds? : List(Expression(DoubleFloat)) -> Boolean
--R sortConstraints : Record(fn: Expression(DoubleFloat),init: List(DoubleFloat),lb: List(OrderedCompletion(DoubleFloat)),ub: List(OrderedCompletion(DoubleFloat))) -> List(String)
--R splitLinear : Expression(DoubleFloat) -> Expression(DoubleFloat)
--R sumOfSquares : Expression(DoubleFloat) -> Union(Expression(DoubleFloat),"failed")
--R varList : (Expression(DoubleFloat),NonNegativeInteger) -> List(Symbol)
--R variables : Record(lfn: List(Expression(DoubleFloat)),init: List(DoubleFloat)) -> List(Symbol)
--R
--E 1

)spool
)lisp (bye)

——

— e04AgentsPackage.help —

====================================================================
e04AgentsPackage examples
====================================================================

e04AgentsPackage is a package of numerical agents to be used to
investigate attributes of an input function so as to decide the
measure of an appropriate numerical optimization routine.

See Also:
o )show e04AgentsPackage

——
e04AgentsPackage (E04AGNT)

Exports:
changeNameToObjf  expenseOfEvaluation  finiteBound  linear?  linearMatrix
linearPart  nonLinearPart  optAttributes  quadratic?  simpleBounds?
sortConstraints  splitLinear  sumOfSquares  varList  variables

| package E04AGNT e04AgentsPackage |

)abbrev package E04AGNT e04AgentsPackage
++ Author: Brian Dupee
++ Date Created: February 1996
++ Date Last Updated: June 1996
++ Description:
++ \texttt{\axiomType{e04AgentsPackage}} is a package of numerical agents to be used
++ to investigate attributes of an input function so as to decide the
++ \texttt{\axiomFun{measure}} of an appropriate numerical optimization routine.

e04AgentsPackage(): E == I where
MDF => Matrix DoubleFloat
VEDF => Vector Expression DoubleFloat
EDF => Expression DoubleFloat
EFI => Expression Fraction Integer
PFI => Polynomial Fraction Integer
FI => Fraction Integer
F => Float
DF => DoubleFloat
OCDF => OrderedCompletion DoubleFloat
LDCDF => List OrderedCompletion DoubleFloat
LEDF => List Expression DoubleFloat
PDF => Polynomial DoubleFloat
LDF => List DoubleFloat
INT => Integer
NNI => NonNegativeInteger
722

CHAPTER 6. CHAPTER E

\[ \text{LS} \mapsto \text{List Symbol} \]
\[ \text{EF2} \mapsto \text{ExpressionFunctions2} \]
\[ \text{NOA} \mapsto \text{Record(fn:EDF, init:LDF, lb:LOCDF, cf:LEDF, ub:LOCDF)} \]
\[ \text{LSA} \mapsto \text{Record(lfn:LEDF, init:LDF)} \]

\[ \text{E} \mapsto \text{with} \]
\[ \text{finiteBound}:(\text{LOCDF}, \text{DF}) \to \text{LDF} \]
\[ \quad \text{++ finiteBound}(l, b) \text{ replaces all instances of an infinite entry in} \]
\[ \quad \text{++ } \text{axiom}(l) \text{ by a finite entry } \text{axiom}(b) \text{ or } \text{axiom}(-b). \]
\[ \text{sortConstraints:NOA} \to \text{NOA} \]
\[ \quad \text{++ sortConstraints(args) uses a simple bubblesort on the list of} \]
\[ \quad \text{++ constraints using the degree of the expression on which to sort.} \]
\[ \quad \text{++ Of course, it must match the bounds to the constraints.} \]
\[ \text{sumOfSquares:EDF} \to \text{Union(EDF,"failed")} \]
\[ \quad \text{++ sumOfSquares}(f) \text{ returns either an expression for which the square is} \]
\[ \quad \text{++ the original function of "failed".} \]
\[ \text{splitLinear:EDF} \to \text{EDF} \]
\[ \quad \text{++ splitLinear}(f) \text{ splits the linear part from an expression which it} \]
\[ \quad \text{++ returns.} \]
\[ \text{simpleBounds?:LEDF} \to \text{Boolean} \]
\[ \quad \text{++ simpleBounds?}(l) \text{ returns true if the list of expressions } l \text{ are} \]
\[ \quad \text{++ simple.} \]
\[ \text{linear?:LEDF} \to \text{Boolean} \]
\[ \quad \text{++ linear?}(l) \text{ returns true if all the bounds } l \text{ are either linear or} \]
\[ \quad \text{++ simple.} \]
\[ \text{linear?:EDF} \to \text{Boolean} \]
\[ \quad \text{++ linear?}(e) \text{ tests if } \text{axiom}(e) \text{ is a linear function.} \]
\[ \text{linearMatrix:(LEDF, NNI)} \to \text{MDF} \]
\[ \quad \text{++ linearMatrix}(l,n) \text{ returns a matrix of coefficients of the linear} \]
\[ \quad \text{++ functions in } \text{axiom}(l). \text{ If } l \text{ is empty, the matrix has at least one} \]
\[ \quad \text{++ row.} \]
\[ \text{linearPart:LEDF} \to \text{LEDF} \]
\[ \quad \text{++ linearPart}(l) \text{ returns the list of linear functions of } \text{axiom}(l). \]
\[ \text{nonLinearPart:LEDF} \to \text{LEDF} \]
\[ \quad \text{++ nonLinearPart}(l) \text{ returns the list of non-linear functions of } l. \]
\[ \text{quadratic?:EDF} \to \text{Boolean} \]
\[ \quad \text{++ quadratic?}(e) \text{ tests if } \text{axiom}(e) \text{ is a quadratic function.} \]
\[ \text{variables:LSA} \to \text{LS} \]
\[ \quad \text{++ variables}(args) \text{ returns the list of variables in } \text{axiom}(\text{args.lfn}) \]
\[ \text{varList:(EDF,NNI)} \to \text{LS} \]
\[ \quad \text{++ varList}(e,n) \text{ returns a list of } \text{axiom}(n) \text{ indexed variables with name} \]
\[ \quad \text{++ as in } \text{axiom}(e). \]
\[ \text{changeNameToObjf:(Symbol,Result)} \to \text{Result} \]
\[ \quad \text{++ changeNameToObjf}(s,r) \text{ changes the name of item } \text{axiom}(s) \text{ in } \text{axiom}(r) \]
\[ \quad \text{++ to objf.} \]
\[ \text{expenseOfEvaluation:LSA} \to \text{F} \]
\[ \quad \text{++ expenseOfEvaluation}(o) \text{ returns the intensity value of the} \]
\[ \quad \text{++ cost of evaluating the input set of functions. This is in terms} \]
\[ \quad \text{++ of the number of ‘‘operational units’’. It returns a value} \]
\[ \quad \text{++ in the range [0,1].} \]
optAttributes:Union(noa:NOA,lsa:LSA) -> List String
  ++ optAttributes(o) is a function for supplying a list of attributes
  ++ of an optimization problem.

I ==> add

import ExpertSystemToolsPackage, ExpertSystemContinuityPackage

sumOfSquares2:EFI -> Union(EFI,"failed")
nonLinear?:EDF -> Boolean
finiteBound2:(OCDF,DF) -> DF
functionType:EDF -> String

finiteBound2(a:OCDF,b:DF):DF ==
  not finite?(a) =>
    positive?(a) => b
    -b
  retract(a)@DF

finiteBound(l:LOCDF,b:DF):LDF == [finiteBound2(i,b) for i in l]

sortConstraints(args:NOA):NOA ==
  Args := copy args
  c:LEDF := Args.cf
  l:LOCDF := Args.lb
  u:LOCDF := Args.ub
  m:INT := (# c) - 1
  n:INT := (# 1) - m
  for j in m..1 by -1 repeat
    for i in 1..j repeat
      s:EDF := c.i
      t:EDF := c.(i+1)
      if linear?(t) and (nonLinear?(s) or quadratic?(s)) then
        swap!(c,i,i+1)$LEDF
        swap!(l,n+i-1,n+i)$LOCDF
        swap!(u,n+i-1,n+i)$LOCDF
  Args

changeNameToObjf(s:Symbol,r:Result):Result ==
  a := remove!(s,r)$Result
  a case Any =>
    insert![objf@Symbol,a],r)$Result
  r
  r

sum(a:EDF,b:EDF):EDF == a+b

variables(args:LSA): LS == variables(reduce(sum,(args.lfn)))

sumOfSquares(f:EDF):Union(EDF,"failed") ==
e := edf2efi(f)
s:Union(EFI,"failed") := sumOfSquares2(e)
s case EFI =>
    map(fi2df,s)$EF2(FI,DF)
"failed"

sumOfSquares2(f:EFI):Union(EFI,"failed") ==
p := retractIfCan(f)@Union(PFI,"failed")
p case PFI =>
    r := squareFreePart(p)$PFI
    (p=r)$Boolean => "failed"
    tp := totalDegree(p)$PFI
    tr := totalDegree(r)$PFI
    t := tp quo tr
    found := false
    q := r
    for i in 2..t by 2 repeat
        s := q**2
        (s=p)$Boolean =>
            found := true
            leave
        q := r**i
    if found then
        q :: EFI
    else
        "failed"
"failed"

splitLinear(f:EDF):EDF ==
out := 0$EDF
(l := isPlus(f)$EDF) case LEDF =>
    for i in l repeat
        if not quadratic? i then
            out := out + i
    out
out

edf2pdf(f:EDF):PDF == (retract(f)@PDF)$EDF

varList(e:EDF,n:NNI):LS ==
s := name(first(variables(edf2pdf(e))$PDF)$LS)$Symbol
[subscript(s,[t::OutputForm]) for t in expand([1..n])$Segment(Integer)]

functionType(f:EDF):String ==
n := #(variables(f))$EDF
p := (retractIfCan(f)@Union(PDF,"failed"))$EDF
p case PDF =>
    d := totalDegree(p)$PDF
--    one?(n*d) => "simple"
    (n*d) = 1 => "simple"
-- one?(d) => "linear"
(d = 1) => "linear"
(d=2)@Boolean => "quadratic"
"non-linear"
"non-linear"

simpleBounds?(l: LDF):Boolean ==
a := true
for e in l repeat
    not (functionType(e) = "simple")@Boolean =>
        a := false
    leave
a

dimple?(e:DF):Boolean == (functionType(e) = "simple")@Boolean
linear?(e:DF):Boolean == (functionType(e) = "linear")@Boolean
quadratic?(e:DF):Boolean == (functionType(e) = "quadratic")@Boolean
nonLinear?(e:DF):Boolean == (functionType(e) = "non-linear")@Boolean

linear?(l: LDF):Boolean ==
a := true
for e in l repeat
    s := functionType(e)
    (s = "quadratic")@Boolean or (s = "non-linear")@Boolean =>
        a := false
    leave
a

simplePart(l: LDF):LDF == [i for i in l | simple?(i)]

linearPart(l: LDF):LDF == [i for i in l | linear?(i)]

nonLinearPart(l: LDF):LDF ==
    [i for i in l | not linear?(i) and not simple?(i)]

linearMatrix(l: LDF, n:NNI):MDF ==
    empty?(l) => mat([],n)
    L := linearPart l
    M := zero(max(1, # L)$NNI,n)$MDF
    vars := varList(first(l)$LDF,n)
    row:INT := 1
    for a in L repeat
        for j in monomials(edf2pdf(a))$PDF repeat
            col:INT := 1
            for c in vars repeat
                if ((first(variables(j)$PDF)$LS)=c)@Boolean then
                    M(row,col) := first(coefficients(j)$PDF)$LDF
col := col + 1
row := row + 1

expenseOfEvaluation(o: LSA): F ==
expenseOfEvaluation(vector(copy o.lfn)$VEDF)

optAttributes(o: Union(noa: NOA, lsa: LSA)): List String ==
o case noa =>
n := o.noa
s1: String := "The object function is " functionType(n.fn)
if empty?(n.lb) then
  s2: String := "There are no bounds on the variables"
else
  s2: String := "There are simple bounds on the variables"
c := n.cf
if empty?(c) then
  s3: String := "There are no constraint functions"
else
t := #(c)
  lin := #(linearPart(c))
  nonlin := #(nonLinearPart(c))
  s3: String := "There are " string(lin)$String " linear and "
  string(nonlin)$String " non-linear constraints"
[s1, s2, s3]
l := o.lsa
s: String := "non-linear"
if linear?(l.lfn) then
  s := "linear"
["The object functions are " s]
Chapter 7

Chapter F

package FACTFUNC FactoredFunctions

— FactoredFunctions.input —

)set break resume
)sys rm -f FactoredFunctions.output
)spool FactoredFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FactoredFunctions
--R
--R FactoredFunctions(M: IntegralDomain) is a package constructor
--R Abbreviation for FactoredFunctions is FACTFUNC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FACTFUNC
--R
--R------------------------------------- Operations -----------------------------------
--R log : Factored(M) -> List(Record(coef: NonNegativeInteger,logand: M))
--R nthRoot : (Factored(M),NonNegativeInteger) -> Record(exponent: NonNegativeInteger,coef: M,radicand: List(M))
--E 1

)spool
)lisp (bye)

———
— FactoredFunctions.help —

FactoredFunctions examples

Computes various functions on factored arguments.

See Also:
- )show FactoredFunctions

———

FactoredFunctions (FACTFUNC)

Exports:
  log  nthRoot

——— package FACTFUNC FactoredFunctions ——

)abbrev package FACTFUNC FactoredFunctions
++ Author: Manuel Bronstein
++ Date Created: 2 Feb 1988
++ Date Last Updated: 25 Jun 1990
++ Description:
++ computes various functions on factored arguments.
-- not visible to the user

FactoredFunctions(M: IntegralDomain): Exports == Implementation where
  N ==> NonNegativeInteger

Exports ==> with
  nthRoot: (Factored M,N) -> Record(exponent:N,coef:M,radicand:List M)
++ nthRoot(f, n) returns \(\{p, r, [r_1, \ldots, r_m]\}\) such that
++ the nth-root of f is equal to \(r * \text{pth-root}(r_1 * \ldots * r_m)\),
++ where \(r_1, \ldots, r_m\) are distinct factors of f,
++ each of which has an exponent smaller than p in f.

log : Factored M -> List Record(coef:N, logand:M)
++ log(f) returns \(\{(a_1, b_1), \ldots, (a_m, b_m)\}\) such that
++ the logarithm of f is equal to \(a_1*\log(b_1) + \ldots + a_m*\log(b_m)\).

Implementation ==> add
nthRoot(ff, n) ==
  coeff:M := 1
  -- radi:List(M) := (one? unit ff => empty(); [unit ff])
radi:List(M) := (((unit ff) = 1) => empty(); [unit ff])
  if
  d:N :=
    empty? radi => gcd(concat(n, \[t.exponent::N for t in lf\])::N)
    1
  n := n quo d
  for term in lf repeat
    qr := divide(term.exponent::N quo d, n)
    coeff := coeff * term.factor ** qr.quotient
    not zero?(qr.remainder) =>
      radi := concat_!(radi, term.factor ** qr.remainder)
  [n, coeff, radi]

log ff ==
  ans := unit ff
  concat([1, unit ff],
    [[term.exponent::N, term.factor] for term in factors ff])

package FR2 FactoredFunctions2
\texttt{)} \texttt{sys rm -f FactoredFunctions2.output} \\
\texttt{)} \texttt{spool FactoredFunctions2.output} \\
\texttt{)} \texttt{set message test on} \\
\texttt{)} \texttt{set message auto off} \\
\texttt{)} \texttt{clear all} \\
\texttt{)} \texttt{\textbf{--S 1 of 7}} \\
\texttt{double(x) == x + x} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--E 1}} \\
\texttt{)} \texttt{\textbf{--S 2 of 7}} \\
\texttt{f := factor(720)} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--R 4 2}} \\
\texttt{)} \texttt{\textbf{--R (2) 2 3 5}} \\
\texttt{)} \texttt{\textbf{--R Type: Factored(Integer)}} \\
\texttt{)} \texttt{\textbf{--E 2}} \\
\texttt{)} \texttt{\textbf{--S 3 of 7}} \\
\texttt{map(double,f)} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--R Compiling function double with type Integer -> Integer}} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--R 4 2}} \\
\texttt{)} \texttt{\textbf{--R (3) 2 4 6 10}} \\
\texttt{)} \texttt{\textbf{--R Type: Factored(Integer)}} \\
\texttt{)} \texttt{\textbf{--E 3}} \\
\texttt{)} \texttt{\textbf{--S 4 of 7}} \\
\texttt{makePoly(b) == x + b} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--R Type: Void}} \\
\texttt{)} \texttt{\textbf{--E 4}} \\
\texttt{)} \texttt{\textbf{--S 5 of 7}} \\
\texttt{g := map(makePoly,f)} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--R Compiling function makePoly with type Integer -> Polynomial(Integer)}} \\
\texttt{)} \texttt{\textbf{--R}} \\
\texttt{)} \texttt{\textbf{--R 4 2}} \\
\texttt{)} \texttt{\textbf{--R (5) (x + 1)(x + 2) (x + 3) (x + 5)}} \\
\texttt{)} \texttt{\textbf{--R Type: Factored(Polynomial(Integer))}} \\
\texttt{)} \texttt{\textbf{--E 5}} \\
\texttt{)} \texttt{\textbf{--S 6 of 7}} \\
\texttt{nthFlag(g,1)}
FactoredFunctions2 examples

FactoredFunctions2 contains functions that involve factored objects whose underlying domains may not be the same. For example, map might be used to coerce an object of type Factored(Integer) to Factored(Complex(Integer)).

The FactoredFunctions2 package implements one operation, map, for applying an operation to every base in a factored object and to the unit.

double(x) == x + x

Type: Void

f := factor(720)
  4  2
  2  3  5

Type: Factored Integer

Actually, the map operation used in this example comes from Factored itself, since double takes an integer argument and returns an integer result.
map(double,f)
4 2
2 4 6 10
Type: Factored Integer

If we want to use an operation that returns an object that has a type
different from the operation's argument, the map in Factored cannot be
used and we use the one in FactoredFunctions2.

makePoly(b) == x + b
4 2
(x + 1)(x + 2) (x + 3) (x + 5)
Type: Factored Polynomial Integer

In fact, the "2" in the name of the package means that we might
be using factored objects of two different types.

g := map(makePoly,f)

It is important to note that both versions of map destroy any information
known about the bases (the fact that they are prime, for instance).

The flags for each base are set to "nil" in the object returned by map.

nthFlag(g,1)
"nil"
Type: Union("nil",...)

See Also:
o )help Factored
o )show FactoredFunctions2
FactoredFunctions2 (FR2)

Exports:
map

— package FR2 FactoredFunctions2 —

)abbrev package FR2 FactoredFunctions2
++ Author: Robert S. Sutor
++ Date Created: 1987
++ Description:
++ \spadtype{FactoredFunctions2} contains functions that involve
++ factored objects whose underlying domains may not be the same.
++ For example, \spadfun{map} might be used to coerce an object of
++ type \spadtype{Factored(Integer)} to
++ \spadtype{Factored(Complex(Integer))}.

FactoredFunctions2(R, S):Exports == Implementation where
R: IntegralDomain
S: IntegralDomain

Exports ==> with
map: (R -> S, Factored R) -> Factored S
++ map(fn,u) is used to apply the function \userfun{fn} to every
++ factor of \spadvar{u}. The new factored object will have all its
++ information flags set to "nil". This function is used, for
++ example, to coerce every factor base to another type.

Implementation ==> add
map(func, f) ==
    func(unit f) *
    _#/nilFactor(func(g.factor), g.exponent) for g in factors f]
package FRUTIL FactoredFunctionUtilities

--- FactoredFunctionUtilities.input ---

)set break resume
/sys rm -f FactoredFunctionUtilities.output
/spool FactoredFunctionUtilities.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FactoredFunctionUtilities
--R
--R FactoredFunctionUtilities(R: IntegralDomain) is a package constructor
--R Abbreviation for FactoredFunctionUtilities is FRUTIL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FRUTIL
--R
--R-------------------------------------- Operations --------------------------------------
--R mergeFactors : (Factored(R),Factored(R)) -> Factored(R)
--R refine : (Factored(R),(R -> Factored(R))) -> Factored(R)
--R
--E 1

)spool
)lisp (bye)

--- FactoredFunctionUtilities.help ---

====================================================================
FactoredFunctionUtilities examples
====================================================================

FactoredFunctionUtilities implements some utility functions for
manipulating factored objects.

See Also:
- \)show FactoredFunctionUtilities

---

**FactoredFunctionUtilities (FRUTIL)**

Exports:
- mergeFactors
- refine

---

```lisp
)abbrev package FRUTIL FactoredFunctionUtilities
++ Description:
++ \spadtype{FactoredFunctionUtilities} implements some utility
++ functions for manipulating factored objects.

FactoredFunctionUtilities(R): Exports == Implementation where
R: IntegralDomain
FR ==> Factored R

Exports == with
  refine: (FR, R-> FR) -> FR
  ++ refine(u,fn) is used to apply the function \userfun{fn} to
  ++ each factor of \spadvar{u} and then build a new factored
  ++ object from the results. For example, if \spadvar{u} were
  ++ created by calling \spad{nilFactor(10,2)} then
  ++ \spad{refine(u,factor)} would create a factored object equal
  ++ to that created by \spad{factor(100)} or
  ++ \spad{primeFactor(2,2) * primeFactor(5,2)}.
```
mergeFactors: (FR,FR) -> FR
  ++ mergeFactors(u,v) is used when the factorizations of \spadvar{u}
  ++ and \spadvar{v} are known to be disjoint, e.g. resulting from a
  ++ content/primitive part split. Essentially, it creates a new
  ++ factored object by multiplying the units together and appending
  ++ the lists of factors.

Implementation ==> add
fg: FR
func: R -> FR
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FF ==> Record(flg: fUnion, fctr: R, xpnt: Integer)

mergeFactors(f,g) ==
  makeFR(unit(f)*unit(g),append(factorList f,factorList g))

refine(f, func) ==
  u := unit(f)
  l: List FF := empty()
  for item in factorList f repeat
    fitem := func item.fctr
    u := u*unit(fitem) ** (item.xpnt :: NonNegativeInteger)
    if item.xpnt = 1 then
      l := concat(factorList fitem,l)
    else l := concat([ [v.flg,v.fctr,v.xpnt*item.xpnt] for v in factorList fitem ],l)
  makeFR(u,l)

— FRUTIL.dotabb —

"FRUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRUTIL"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FRUTIL" -> "ALGEBRA"

package FACUTIL FactoringUtilities

— FactoringUtilities.input —

)set break resume
)sys rm -f FactoringUtilities.output
This package provides utilities used by the factorizers which operate on polynomials represented as univariate polynomials with multivariate coefficients.

See Also:
  o )show FactoringUtilities

---

FactoringUtilities.help

This package provides utilities used by the factorizers which operate on polynomials represented as univariate polynomials with multivariate coefficients.

See Also:
  o )show FactoringUtilities

---
FactoringUtilities (FACUTIL)

Exports:
  completeEval  degree  lowerPolynomial  normalDeriv  raisePolynomial
  ran  variables

— package FACUTIL FactoringUtilities —

)abbrev package FACUTIL FactoringUtilities
++ Author: Barry Trager
++ Date Created: March 12, 1992
++ Description:
++ This package provides utilities used by the factorizers
++ which operate on polynomials represented as univariate polynomials
++ with multivariate coefficients.

FactoringUtilities(E,OV,R,P) : C == T where
  E : OrderedAbelianMonoidSup
  OV : OrderedSet
  R : Ring
  P : PolynomialCategory(R,E,OV)

SUP ==> SparseUnivariatePolynomial
NNI ==> NonNegativeInteger
Z  ==> Integer

C == with
  completeEval : (SUP P,List OV,List R) -> SUP R
  ++ completeEval(upoly, lvar, lval) evaluates the polynomial upoly
  ++ with each variable in lvar replaced by the corresponding value
  ++ in lval. Substitutions are done for all variables in upoly
  ++ producing a univariate polynomial over R.
  degree : (SUP P,List OV) -> List NNI
  ++ degree(upoly, lvar) returns a list containing the maximum
  ++ degree for each variable in lvar.
  variables : SUP P -> List OV
  ++ variables(upoly) returns the list of variables for the coefficients
++ of upoly.
lowerPolynomial: SUP P -> SUP R
++ lowerPolynomial(upoly) converts upoly to be a univariate polynomial
++ over R. An error if the coefficients contain variables.
raisePolynomial: SUP R -> SUP P
++ raisePolynomial(rpoly) converts rpoly from a univariate polynomial
++ over r to be a univariate polynomial with polynomial coefficients.
normalDeriv : (SUP P,Z) -> SUP P
++ normalDeriv(poly,i) computes the ith derivative of poly divided
++ by i!.
ran : Z -> R
++ ran(k) computes a random integer between -k and k as a member of R.

T == add

lowerPolynomial(f:SUP P) : SUP R ==
  zero? f => 0$SUP(R)
  monomial(ground leadingCoefficient f, degree f)$SUP(R) +
  lowerPolynomial(reductum f)

raisePolynomial(u:SUP R) : SUP P ==
  zero? u => 0$SUP(P)
  monomial(leadingCoefficient(u)::P, degree u)$SUP(P) +
  raisePolynomial(reductum u)

completeEval(f:SUP P,lvar:List OV,lval:List R) : SUP R ==
  zero? f => 0$SUP(R)
  monomial(ground eval(leadingCoefficient f,lvar,lval),degree f)$SUP(R) +
  completeEval(reductum f,lvar,lval)

degree(f:SUP P,lvar:List OV) : List NNI ==
  coefs := coefficients f
  ldeg:= ["max"/[degree(fc,xx) for fc in coefs] for xx in lvar]

variables(f:SUP P) : List OV ==
  "setUnion"/[variables cf for cf in coefficients f]

if R has FiniteFieldCategory then
  ran(k:Z):R == random()$R
else
  ran(k:Z):R == (random(2*k+1)$Z -k)::R

-- Compute the normalized m derivative
normalDeriv(f:SUP P,m:Z) : SUP P==
  (n1:Z:=degree f) < m => 0$SUP(P)
  n1=m => (leadingCoefficient f)::SUP(P)
  k:=binomial(n1,m)
  ris:=SUP:=0$SUP(P)
  n:Z:=n1
  while n>= m repeat
while n1>n repeat
  k:=(k*(n1-m)) quo n1
  n1:=n1-1
  ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
  f:=reductum f
  n:=degree f
  ris

— FACUTIL.dotabb —

"FACUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACUTIL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FACUTIL" -> "PFECAT"

package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber

— FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.input —

)set break resume
)sys rm -f FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)spool FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
--R
--R FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber(K::PseudoAlgebraicClosureOfAlgExtOfRationalNumber)
--R Abbreviation for FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber is FACTEXT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FACTEXT
--R
--R------------------------------------------ Operations ------------------------------------------
--R factor : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R factorSqFree : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R
FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber (FACTEXT)

Exports:
factor  factorSqFree

— package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber —

)abbrev package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: May, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF

\texttt{FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber(K):Exports == Implementation where}
\texttt{K: PseudoAlgebraicClosureOfAlgExtOfRationalNumberCategory}
\texttt{SUP ==> SparseUnivariatePolynomial}
\texttt{Q ==> PseudoAlgebraicClosureOfRationalNumber}
\texttt{NNI ==> NonNegativeInteger}
\texttt{UP ==> SUP(K)}
\texttt{UPUP ==SUP(SUP(K))}
\texttt{FACTRNQ == FactorisationOverPseudoAlgebraicClosureOfRationalNumber(Q)}

\texttt{Exports ==> with}
\texttt{factor: (UP,K) -> Factored UP}
\texttt{factorSqFree: (UP,K) -> Factored UP}

\texttt{Implementation ==> add}
\texttt{up2Rat: UP -> SUP(Q)}
\texttt{rat2up: SUP(Q) -> UP}

\texttt{factRat: UP -> Factored UP}
\texttt{liftPoly: (UP, K) -> UPUP}

\texttt{liftDefPoly: UP -> UPUP}
\texttt{norm: (UP, K) -> UP}
\texttt{factParPert: ( UP,K,K) -> Factored UP}
\texttt{trans: (UP, K) -> UP}
\texttt{swapCoefWithVar: ( UP , NNI) -> UPUP}
\texttt{frRat2frUP: Factored SUP(Q) -> Factored UP}

\texttt{factor(pol,a)==}
\texttt{polSF:= squareFree pol}
\texttt{reduce("**", [ factorSqFree(fr.fctr,a)**(fr.xpnt pretend NNI) _
    for fr in factorList polSF ] , 1)}

\texttt{factorSqFree(pol,a)==}
\texttt{ratPol: SUP(Q)}
\texttt{aa:Q}
\texttt{ground? a =>}
\texttt{aa:= retract(a)@Q}
\texttt{ratPol:= up2Rat pol}
\texttt{frRat2frUP factor(ratPol,aa)$FACTRNQ::Factored UP}
\texttt{nPol:= norm(pol,a)}
\texttt{ta:=previousTower a}
PACKAGE FACTORISATIONOVERPSEUDOALGEBRAICCLOSEDOFALGEXTOFRATIONALNUMBERS

```plaintext
factN := factor(nPol, ta)
lfactnPol := factorList factN
G := UP := 1
L := Factored UP := 1
for fr in lfactnPol repeat
    G := gcd([pol, fr.fctr])
    pol := pol quo UP G
    if one? fr.xpnt then
        L := L * flagFactor(G, 1, "prime")$Factored(UP)
    else
        L := L * factParPert(G, a, a)
    L

factParPert(pol, a, b) ==
    pol := trans(pol, b)
    frpol := factorList factor(polt, a)
    sl := [fr.fctr for fr in frpol]
    slt := [trans(p, -b) for p in sl]
    nfrpol := [flagFactor(p, fr.xpnt, fr.flg)$Factored(UP) for p in slt for fr in frpol]
    reduce("*", nfrpol)

frRat2frUP(fr) ==
    frpol := factorList fr
    sl := [fr.fctr for fr in frpol]
    slt := [rat2up p for p in sl]
    nfrpol := [flagFactor(p, fr.xpnt, fr.flg)$Factored(UP) for p in slt for fr in frpol]
    reduce("*", nfrpol)

up2Rat(pol) ==
    zero?(pol) => 0
    d := degree pol
    a:Q := retract(leadingCoefficient pol)$Q
    monomial(a, d)$SUP(Q) + up2Rat(reductum pol)

rat2up(pol) ==
    zero?(pol) => 0
    d := degree pol
    a:K := (leadingCoefficient pol) :: K
    monomial(a, d)$UP + rat2up(reductum pol)

trans(pol, a) ==
    zero? pol => 0
    lc := leadingCoefficient pol
    d := degree pol
    lc*(monomial(1,1)$UP + monomial(-a,0)$UP)**d + trans(reductum pol, a)

liftDefPoly(pol) ==
    zero?(pol) => 0
```

lc:= leadingCoefficient pol
d:= degree pol
monomial( monomial(lc,0)$UP , d )$UPUP + liftDefPoly reductum pol

norm(pol,a)==
lpol:=liftPoly(pol,a)
defPol:=definingPolynomial a
ldefPol:=liftDefPoly defPol
resultant(ldefPol,lpol)

swapCoefWithVar(coef,n)==
ground? coef =>
  monomial( monomial( retract coef , n)$SUP(K) , 0)$UPUP
lcoef:=leadingCoefficient(coef)
d:=degree(coef)
monomial(monomial(lcoef,n)$SUP(K),d)$UPUP+_swapCoefWithVar(reductum coef,n )

liftPoly(pol,a)==
zero? pol => 0
lcoef:=leadingCoefficient pol
n:=degree pol
liftCoef:= lift(lcoef,a)$K
swapCoefWithVar(liftCoef , n) + liftPoly( reductum pol , a )

— FACTEXT.dotabb —
"FACTEXT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FACTEXT"]
"PACEXTC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PACEXTC"]
"FACTEXT" -> "PACEXTC"

package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber

— FactorisationOverPseudoAlgebraicClosureOfRationalNumber.input

)set break resume
)sys rm -f FactorisationOverPseudoAlgebraicClosureOfRationalNumber.output
---S 1 of 1
)show FactorisationOverPseudoAlgebraicClosureOfRationalNumber

--R FactorisationOverPseudoAlgebraicClosureOfRationalNumber(K: PseudoAlgebraicClosureOfRationalNumberCategory)
--R Abbreviation for FactorisationOverPseudoAlgebraicClosureOfRationalNumber is FACTRN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FACTRN
--R
--R------------------------------- Operations --------------------------------
--R factor : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R factorSqFree : (SparseUnivariatePolynomial(K),K) -> Factored(SparseUnivariatePolynomial(K))
--R
--E 1

)spool
)lisp (bye)

====================================================================
FactorisationOverPseudoAlgebraicClosureOfRationalNumber examples
====================================================================

Part of the Package for Algebraic Function Fields in one variable PAFF

See Also:
o )show FactorisationOverPseudoAlgebraicClosureOfRationalNumber
FactorisationOverPseudoAlgebraicClosureOfRationalNumber (FACTRN)

Exports:
factor  factorSqFree

— package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber —

)abbrev package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: May, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
FactorisationOverPseudoAlgebraicClosureOfRationalNumber(K):Exports ==
Implementation where
K:PseudoAlgebraicClosureOfRationalNumberCategory
SUP ==> SparseUnivariatePolynomial
Q ==> Fraction Integer
NNI ==> NonNegativeInteger
UP ==> SUP(K)
UPUP ==> SUP(SUP(K))

Exports ==> with
factor: (UP,K) -> Factored UP
factorSqFree: (UP,K) -> Factored UP

Implementation ==> add
up2Rat: UP -> SUP(Q)
rat2up: SUP(Q) -> UP

factRat: UP -> Factored UP
liftPoly: (UP, K) -> UPUP

liftDefPoly: UP -> UPUP
norm: (UP, K) -> UP

factParPert: ( UP,K,K) -> Factored UP

trans: (UP, K) -> UP

swapCoefWithVar: ( UP , NNI) -> UPUP

frRat2frUP: Factored SUP(Q) -> Factored UP

factor(pol,a)==
polSF:= squareFree pol
reduce("*" , _, [ factorSqFree(fr.fctr,a)**(fr.xpnt pretend NNI)_,
   for fr in factorList polSF] , 1)

factorSqFree(pol,a)==
ratPol:UP(Q)
ground? a =>
ratPol:= up2Rat pol
frRat2frUP factor( ratPol )\$RationalFactorize( SUP(Q) ) :: Factored UP
nPol:= norm(pol,a)
ta:=previousTower a
factN := factor( nPol , ta )
lfactnPolar:= factorList factN
G:UP:=1
L: Factored UP:= 1
for fr in lfactnPolar repeat
  G:= gcd( [ pol , fr.fctr ] )
pol:= pol quo$UP G
  if one? fr.xpnt then
    L := L * flagFactor( G, 1 ,"prime")$Factored(UP)
  else
    L := L * factParPert( G, a, a )
L

factParPert(pol, a, b)==
polt:=trans(pol,b)
frpol:= factorList factor(polt,a)
sl:= [ fr.fctr for fr in frpol ]
slt:= [ trans(p , -b) for p in sl ]
nfrpol:= [ flagFactor( p, fr.xpnt , fr.flg )$Factored(UP) _
   for p in slt for fr in frpol ]
reduce("*" , nfrpol)

frRat2frUP(fr)==
frpol:= factorList fr
sl:= [ fr.fctr for fr in frpol ]
slt:= [ rat2up p for p in sl ]
nfrpol := [ flagFactor(p, fr.xpnt, fr.flg) \Factored(UP) _
    for p in slt for fr in frpol ]
reduce("*", nfrpol)

up2Rat(pol) ==
    zero?(pol) => 0
    d := degree pol
    a:Q := retract(leadingCoefficient pol)Q
    monomial(a,d)\$SUP(Q) + up2Rat(reductum pol)

rat2up(pol) ==
    zero?(pol) => 0
    d := degree pol
    a:K := (leadingCoefficient pol) :: K
    monomial(a,d)\$UP + rat2up(reductum pol)

trans(pol,a) ==
    zero? pol => 0
    lc := leadingCoefficient pol
    d := degree pol
    lc*( monomial(1,1)\$UP + monomial(-a,0)\$UP )**d + trans(reductum pol, a)

liftDefPoly(pol) ==
    zero?(pol) => 0
    lc := leadingCoefficient pol
    d := degree pol
    monomial( monomial(lc,0)\$UP , d )\$UPUP + liftDefPoly reductum pol

norm(pol,a) ==
    lpol := liftPoly(pol,a)
    defPol := definingPolynomial a
    ldefPol := liftDefPoly defPol
    resultant(ldefPol,lpol)

swapCoefWithVar(coef,n) ==
    ground? coef =>
        monomial( monomial( retract coef, n)\$SUP(K) , o)\$UPUP
    lcoef := leadingCoefficient(coef)
    d := degree(coef)
    monomial( monomial(lcoef, n)\$SUP(K) , d)\$UPUP + _
        swapCoefWithVar( reductum coef, n )

liftPoly(pol,a) ==
    zero? pol => 0
    lcoef := leadingCoefficient pol
    n := degree pol
    liftCoef := lift(lcoef,a)\$K
    swapCoefWithVar(liftCoef, n) + liftPoly( reductum pol , a )
package FGLMICPK FGLMIIfCanPackage

-- FGLMIIfCanPackage.input --

)set break resume
)sys rm -f FGLMIIfCanPackage.output
)spool FGLMIIfCanPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show FGLMIIfCanPackage

--R FGLMIIfCanPackage(R: GcdDomain,ls: List(Symbol)) is a package constructor
--R Abbreviation for FGLMIIfCanPackage is FGLMICPK
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FGLMICPK

--R--------------------------------- Operations ---------------------------------
--R fglmIfCan : List(Polynomial(R)) -> Union(List(Polynomial(R)),"failed")
--R groebner : List(Polynomial(R)) -> List(Polynomial(R))
--R zeroDimensional? : List(Polynomial(R)) -> Boolean

--R
--E 1

)spool
)lisp (bye)

-- FGLMIIfCanPackage.help --
FGLMIfCanPackage examples
====================================================================
This is just an interface between several packages and domains. The goal is to compute lexicographical Groebner bases of sets of polynomial with type Polynomial R by the FGLM algorithm if this is possible (i.e. if the input system generates a zero-dimensional ideal).

See Also:
o )show FGLMIfCanPackage

FGLMIfCanPackage (FGLMICPK)

Exports:
fglmIfCan groebner zeroDimensional?

— package FGLMICPK FGLMIfCanPackage —

)abbrev package FGLMICPK FGLMIfCanPackage
++ Author: Marc Moreno Maza
++ Date Created: 08/02/1999
++ Date Last Updated: 08/02/1999
++ Description:
++ This is just an interface between several packages and domains.
++ The goal is to compute lexicographical Groebner bases
++ of sets of polynomial with type \spadtype{Polynomial R}
++ by the FGLM algorithm if this is possible (i.e.
++ if the input system generates a zero-dimensional ideal).

FGLMIfCanPackage(R,ls): Exports == Implementation where
R: GcdDomain
ls: List Symbol
V ==> OrderedVariableList ls
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
Q1 ==> Polynomial R
Q2 ==> HomogeneousDistributedMultivariatePolynomial(ls,R)
Q3 ==> DistributedMultivariatePolynomial(ls,R)
E2 ==> HomogeneousDirectProduct(#ls,NonNegativeInteger)
E3 ==> DirectProduct(#ls,NonNegativeInteger)
poltopol ==> PolToPol(ls, R)
lingrobpack ==> LinGroebnerPackage(ls,R)
groebnerpack2 ==> GroebnerPackage(R,E2,V,Q2)
groebnerpack3 ==> GroebnerPackage(R,E3,V,Q3)
Exports == with

zeroDimensional?: List(Q1) -> B
++ \texttt{zeroDimensional?(lq1)} returns true iff
++ \texttt{lq1} generates a zero-dimensional ideal
++ w.r.t. the variables of \texttt{ls}.
fglmIfCan: List(Q1) -> Union(List(Q1), "failed")
++ \texttt{fglmIfCan(lq1)} returns the lexicographical Groebner
++ basis of \texttt{lq1} by using the FGLM strategy,
++ if \texttt{zeroDimensional?(lq1)} holds.
groebner: List(Q1) -> List(Q1)
++ \texttt{groebner(lq1)} returns the lexicographical Groebner
++ basis of \texttt{lq1}. If \texttt{lq1} generates a zero-dimensional
++ ideal then the FGLM strategy is used, otherwise
++ the Sugar strategy is used.

Implementation == add

zeroDim?(lq2: List Q2): Boolean ==
lq2 := groebner(lq2)$groebnerpack2
empty? lq2 => false
#lq2 < #ls => false
lv: List(V) := [(variable(s)$V)::V for s in ls]
for q2 in lq2 while not empty?(lv) repeat
  m := leadingMonomial(q2)
x := mainVariable(m)::V
  if ground?(leadingCoefficient(univariate(m,x))) then
    lv := remove(x, lv)
empty? lv

zeroDimensional?(lq1: List(Q1)): Boolean ==
lq2: List(Q2) := [pToHdmp(q1)$poltopol for q1 in lq1]
zeroDim?(lq2)

fglmIfCan(lq1:List(Q1)): Union(List(Q1),"failed") ==
package FORDER FindOrderFinite

— FindOrderFinite.input —

)set break resume
)sys rm -f FindOrderFinite.output
)spool FindOrderFinite.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FindOrderFinite
--R
--R FindOrderFinite(F: Join(Finite,Field),UP: UnivariatePolynomialCategory(F),UPUP: UnivariatePolynomialCategory(F)) is a package constructor.
FindOrderFinite (FORDER)

Exports:
order

---

FindOrderFinite examples

Finds the order of a divisor over a finite field

See Also:
o )show FindOrderFinite

---
)abbrev package FORDER FindOrderFinite
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Finds the order of a divisor over a finite field

FindOrderFinite(F, UP, UPUP, R): Exports == Implementation where
  F : Join(Finite, Field)
  UP : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R : FunctionFieldCategory(F, UP, UPUP)

Exports ==> with
  order: FiniteDivisor(F, UP, UPUP, R) -> NonNegativeInteger
  ++ order(x) undocumented

Implementation ==> add
  order d ==
    dd := d := reduce d
    for i in 1.. repeat
      principal? dd => return(i::NonNegativeInteger)
      dd := reduce(d + dd)

— FORDER.dotabb —

"FORDER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FORDER"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"FORDER" -> "FFCAT"

— package FAMR2 FiniteAbelianMonoidRingFunctions2 —

)set break resume
)sys rm -f FiniteAbelianMonoidRingFunctions2.output
)spool FiniteAbelianMonoidRingFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FiniteAbelianMonoidRingFunctions2
--R
--R FiniteAbelianMonoidRingFunctions2(E: OrderedAbelianMonoid,R1: Ring,A1: FiniteAbelianMonoidRing(R1,E),R2: Ring,A2: FiniteAbelianMonoidRing(R2,E)) is a package constructor
--R Abbreviation for FiniteAbelianMonoidRingFunctions2 is FAMR2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FAMR2
--R
--R------------------------------- Operations --------------------------------
--R map : ((R1 -> R2),A1) -> A2
--R
--E 1

)spool
)lisp (bye)

———
        — FiniteAbelianMonoidRingFunctions2.help —

====================================================================
FiniteAbelianMonoidRingFunctions2 examples
====================================================================

FiniteAbelianMonoidRing
The packages defined in this file provide fast fraction free rational
interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

See Also:
o )show FiniteAbelianMonoidRingFunctions2

———

FiniteAbelianMonoidRingFunctions2 (FAMR2)
Exports:
map

— package FAMR2 FiniteAbelianMonoidRingFunctions2 —

)abbrev package FAMR2 FiniteAbelianMonoidRingFunctions2
++ Author: Martin Rubey
++ Description:
++ This package provides a mapping function for
++ \spadtype{FiniteAbelianMonoidRing}
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FiniteAbelianMonoidRingFunctions2(E: OrderedAbelianMonoid,
R1: Ring,
A1: FiniteAbelianMonoidRing(R1, E),
R2: Ring,
A2: FiniteAbelianMonoidRing(R2, E)) _
: Exports == Implementation where

Exports == with

map: (R1 -> R2, A1) -> A2
++ \spad{map}(f, a) applies the map f to each coefficient in a. It is
++ assumed that f maps 0 to 0

Implementation == add

map(f: R1 -> R2, a: A1): A2 ==
if zero? a then 0$A2
else
  monomial(f leadingCoefficient a, degree a)$A2 + map(f, reductum a)

— FAMR2.dotabb —

"FAMR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FAMR2"]
"FAMR" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAMR"]
"FAMR2" -> "FAMR"
package FDIV2 FiniteDivisorFunctions2

— FiniteDivisorFunctions2.input —

)set break resume
)sys rm -f FiniteDivisorFunctions2.output
)spool FiniteDivisorFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteDivisorFunctions2
--R
--R FiniteDivisorFunctions2(R1: Field,UP1: UnivariatePolynomialCategory(R1),UPUP1: UnivariatePolynomialCategory(Fraction(UP2)),F2: FunctionFieldCategory(R2,UP2,UPUP2)) is a package constructor
--R Abbreviation for FiniteDivisorFunctions2 is FDIV2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FDIV2
--R
--R--------------------------------- Operations ---------------------------------
--R map : ((R1 -> R2),FiniteDivisor(R1,UP1,UPUP1,F1)) -> FiniteDivisor(R2,UP2,UPUP2,F2)
--R
--E 1

)spool
)lisp (bye)

— FiniteDivisorFunctions2.help —

====================================================================
FiniteDivisorFunctions2 examples
====================================================================

Lift a map to finite divisors.

See Also:
o )show FiniteDivisorFunctions2

———
FiniteDivisorFunctions2 (FDIV2)

Exports:
map

— package FDIV2 FiniteDivisorFunctions2 —

)abbrev package FDIV2 FiniteDivisorFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 19 May 1993
++ Description:
++ Lift a map to finite divisors.

FiniteDivisorFunctions2(R1, UP1, UPUP1, F1, R2, UP2, UPUP2, F2):
Exports == Implementation where
  R1 : Field
  UP1 : UnivariatePolynomialCategory R1
  UPUP1: UnivariatePolynomialCategory Fraction UP1
  F1 : FunctionFieldCategory(R1, UP1, UPUP1)
  R2 : Field
  UP2 : UnivariatePolynomialCategory R2
  UPUP2: UnivariatePolynomialCategory Fraction UP2
  F2 : FunctionFieldCategory(R2, UP2, UPUP2)

Exports ==> with
  map: (R1 -> R2, FiniteDivisor(R1, UP1, UPUP1, F1)) ->
    FiniteDivisor(R2, UP2, UPUP2, F2)
  ++ map(f,d) \undocumented{}  

Implementation ==> add
  import UnivariatePolynomialCategoryFunctions2(R1,UP1,R2,UP2)
  import FunctionFieldCategoryFunctions2(R1,UP1,UPUP1,F1,R2,UP2,UPUP2,F2)
  import FractionalIdealFunctions2(UP1, Fraction UP1, UPUP1, F1, UP2, Fraction UP2, UPUP2, F2)

  map(f, d) ==
rec := decompose d
divisor map(f, rec.principalPart) +
divisor map((s:UP1):UP2 ++> map(f,s), rec.id)

---

FDIV2.dotabb ---

"FDIV2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FDIV2"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"FDIV2" -> "FFCAT"

package FFFACTOR FiniteFieldFactorization

--- FiniteFieldFactorization.input ---

)set break resume
)sys rm -f FiniteFieldFactorization.output
)spool FiniteFieldFactorization.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldFactorization

--R
--R FiniteFieldFactorization(K: FiniteFieldCategory,PolK: UnivariatePolynomialCategory(K)) is a package constructor
--R Abbreviation for FiniteFieldFactorization is FFFACTUR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFACTUR
--R
--R----------------------------------- Operations -----------------------------------
--R factorCantorZassenhaus : (PolK,NonNegativeInteger) -> List(PolK)
--R factorSquareFree : PolK -> List(PolK)
--R factorUsingMusser : PolK -> Factored(PolK)
--R factorUsingYun : PolK -> Factored(PolK)
--R
--E 1

)spool
)lisp (bye)
FiniteFieldFactorization (FFFACTOR)

Exports:
  factor  factorCantorZassenhaus  factorSquareFree
  factorUsingMusser  factorUsingYun  irreducible?

-- package FFFACTOR FiniteFieldFactorization --

)abbreviation package FFFACTOR FiniteFieldFactorization
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: March 2013 by Tim Daly
++ Description:
++ Part of the PAFF package
FiniteFieldFactorization(K : FiniteFieldCategory,
  PolK : UnivariatePolynomialCategory(K)) : with

  factorSquareFree : PolK -> List(PolK)
factorCantorZassenhaus : (PolK, NonNegativeInteger) -> List(PolK)

factor : PolK -> Factored(PolK)

factorUsingYun : PolK -> Factored(PolK)

factorUsingMusser : PolK -> Factored(PolK)

irreducible? : PolK -> Boolean

== add

import FiniteFieldSquareFreeDecomposition(K, PolK)

p : NonNegativeInteger := characteristic()$K

p' : NonNegativeInteger := p quo 2   -- used for odd p : (p-1)/2

q : NonNegativeInteger := size()$K

q' : NonNegativeInteger := q quo 2   -- used for odd q : (q-1)/2

X : PolK := monomial(1, 1)

primeKdim : NonNegativeInteger :=
  q_quo_p : NonNegativeInteger := q quo p ; e : NonNegativeInteger := 1
  while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)

exp(P : PolK, n : NonNegativeInteger, R : PolK) : PolK ==
  PP : PolK := P rem R ; Q : PolK := 1
  repeat
    if odd?(n) then Q := Q * PP rem R
    (n := n quo 2) = 0 => leave
    PP := PP * PP rem R
  return Q

pPowers(P : PolK) : PrimitiveArray(PolK) == -- P is monic
  n := degree(P)
  result : PrimitiveArray(PolK) := new(n, 1)
  result(1) := Qi := Q := exp(X, p, P)
  for i in 2 .. n-1 repeat (Qi := Qi*Q rem P ; result(i) := Qi)
  return result

pExp(Q : PolK, Xpowers : PrimitiveArray(PolK)) : PolK ==
  Q' : PolK := 0
  while Q' ^= 0 repeat
    Q' := Q' +primeFrobenius(leadingCoefficient(Q))*Xpowers(degree(Q))
    Q := reductum(Q)
return Q'

pTrace(Q : PolK, d : NonNegativeInteger, P : PolK, Xpowers : PrimitiveArray(PolK)) : PolK ==
  Q : PolK := Q rem P
  result : PolK := Q
  for i in 1 .. d-1 repeat result := Q + pExp(result, Xpowers)
  return result rem P

random(n : NonNegativeInteger) : PolK ==
  repeat
    if (deg := (random(n)$Integer)::NonNegativeInteger) > 0 then leave
  repeat
    if (x : K := random()$K) ^= 0 then leave
    result : PolK := monomial(x, deg) + +/
    [monomial(random()$K, i) for i in 0 .. deg-1]
  return result

internalFactorCZ(P : PolK, -- P monic-squarefree
d : NonNegativeInteger, Xpowers : PrimitiveArray(PolK)) : List(PolK) ==
  listOfFactors : List(PolK) := [P]
  degree(P) = d => return listOfFactors
  result : List(PolK) := []
  pDim : NonNegativeInteger := d * primeKdim
  Q : PolK := P
  repeat
    G := pTrace(random(degree(Q)), pDim, Q, Xpowers)
    if p > 2 then G := exp(G, p', Q) - 1
    Q1 := gcd(G, Q) ; d1 := degree(Q1)
    if d1 > 0 and d1 < degree(Q) then
      listOfFactors := rest(listOfFactors)
      if d1 = d then result := cons(Q1, result)
        else listOFFactors := cons(Q1, listOFFactors)
    Q1 := Q quo Q1 ; d1 := degree(Q1)
    if d1 = d then result := cons(Q1, result)
      else listOFFactors := cons(Q1, listOFFactors)
    if empty?(listOFFactors) then leave
  Q := first(listOFFactors)
  return result

internalFactorSquareFree(P : PolK) : List(PolK) == -- P is monic-squarefree
  degree(P) = 1 => [P]
  result : List(PolK) := []
  Xpowers : PrimitiveArray(PolK) := pPowers(P)
  S : PolK := Xpowers(1)
  for j in 1 .. primeKdim-1 repeat S := pExp(S, Xpowers)
  for i in 1 .. repeat -- S = X**(q**i) mod P
    if degree(R := gcd(S - X, P)) > 0 then
result := concat(internalFactorCZ(R, i, Xpowers), result)
if degree (P) = degree (R) then return result
P := P quo R
if i >= degree(P) quo 2 then return cons(P, result)
for j in 0 .. degree(P)-1 repeat Xpowers(j) := Xpowers(j) rem P
S := S rem P
else if i >= degree(P) quo 2 then return cons(P, result)
for j in 1 .. primeKdim repeat S := pExp(S, Xpowers)

internalFactor(P:PolK, sqrfree:PolK -> Factored(PolK)) : Factored(PolK) ==
result : Factored(PolK)
if (d := minimumDegree(P)) > 0 then
P := P quo monomial(1, d)
result := primeFactor(X, d)
else
result := 1
degree(P) = 0 => P * result
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
degree(P) = 1 => lcP::PolK * primeFactor(P, 1) * result
sqfP : Factored(PolK) := sqrfree(P)
for x in factors(sqfP) repeat
  xFactors : List(PolK) := internalFactorSquareFree(x.factor)
  result := result * [primeFactor(Q, x.exponent) for Q in xFactors]
return lcP::PolK * result

factorUsingYun(P : PolK) : Factored(PolK) == internalFactor(P, Yun)
factorUsingMusser(P : PolK) : Factored(PolK) == internalFactor(P, Musser)
factor(P : PolK) : Factored(PolK) == factorUsingYun(P)
factorSquareFree(P : PolK) : List(PolK) ==
degree(P) = 0 => []
discriminant(P) = 0 => error("factorSquareFree : non quadratfrei")
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
return internalFactorSquareFree(P)
factorCantorZassenhaus(P : PolK, d : NonNegativeInteger) : List(PolK) ==
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
degree(P) = 1 => [P]
return internalFactorCZ(P, d, pPowers(P))
qExp(Q : PolK, XqPowers : PrimitiveArray(PolK)) : PolK ==
Q' : PolK := 0
while Q ^= 0 repeat
  Q' := Q' + leadingCoefficient(Q) * XqPowers(degree(Q))
  Q := reductum(Q)
return Q'
qPowers (Xq : PolK, P : PolK) : PrimitiveArray(PolK) == -- Xq = X**q mod P
n := degree(P)
result := PrimitiveArray(PolK) := new(n, 1)
result(1) := Q := Xq
for i in 2 .. n-1 repeat (Q := Q*Xq rem P ; result(i) := Q)
return result

discriminantTest?(P : PolK) : Boolean ==
(discriminant(P)) = 0 => true
StickelbergerTest : Boolean := (delta ** q' = 1) = even?(degree(P))
return StickelbergerTest

evenCharacteristicIrreducible?(P : PolK) : Boolean ==
(n := degree(P)) = 0 => false
n = 1 => true
degree(gcd(P, D(P))) > 0 => false
if (1cP := leadingCoefficient(P)) = 1 then P := inv(1cP) * P
S := exp(X, q, P)
if degree(gcd(S - X, P)) > 0 then
    return false
if n < 4 then return true
maxDegreeToTest : NonNegativeInteger := n quo 2
for i in 2 .. maxDegreeToTest repeat
    S := qExp(S, XqPowers)
    if degree(gcd(S - X, P)) > 0 then
        return false
return true

oddCharacteristicIrreducible?(P : PolK) : Boolean ==
(n := degree(P)) = 0 => false
n = 1 => true
discriminantTest?(P) => false
if (1cP := leadingCoefficient(P)) = 1 then P := inv(1cP) * P
S := exp(X, q, P)
if degree(gcd(S - X, P)) > 0 then
    return false
if n < 6 then return true
maxDegreeToTest : NonNegativeInteger := n quo 3
for i in 2 .. maxDegreeToTest repeat
    S := qExp(S, XqPowers)
    if degree(gcd(S - X, P)) > 0 then
        return false
return true

if p = 2 then
    irreducible?(P : PolK) : Boolean == evenCharacteristicIrreducible?(P)
ext
irreducible?(P : PolK) : Boolean == oddCharacteristicIrreducible?(P)

--

-- FFFACTOR.dotabb --

"FFFACTOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFACTOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFACTOR" -> "PFECAT"

--

package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect

-- FiniteFieldFactorizationWithSizeParseBySideEffect.input --

)set break resume
)sys rm -f FiniteFieldFactorizationWithSizeParseBySideEffect.output
)spool FiniteFieldFactorizationWithSizeParseBySideEffect.output
)set message test on
)clear all

-- 1 of 1
)show FiniteFieldFactorizationWithSizeParseBySideEffect
--R FiniteFieldFactorizationWithSizeParseBySideEffect(K: FiniteFieldCategory,PolK: UnivariatePolynomialCategory) is a package constructor
--R Abbreviation for FiniteFieldFactorizationWithSizeParseBySideEffect is FFFACTSE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFACTSE
--R
--R----------------------------------- Operations --------------------------------
--R factorCantorZassenhaus : (PolK,NonNegativeInteger) -> List(PolK)
--R factorSquareFree : PolK -> List(PolK)
--R factorUsingMusser : PolK -> Factored(PolK)
--R factorUsingYun : PolK -> Factored(PolK)
--R
--E 1

)spool
)lisp (bye)
FiniteFieldFactorizationWithSizeParseBySideEffect (FFFACTSE)

Exports:
factor        factorCantorZassenhaus        factorSquareFree
factorUsingMusser  factorUsingYun         irreducible?

--- package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect ---

)abbrev package FFFACTSE FiniteFieldFactorizationWithSizeParseBySideEffect
++ Author: Patrice Naudin, Claude Quitté, Kaj Laursen
++ Date Created: September 1996
++ Date Last Updated: April 2010 by Tim Daly
++ Description:
++ Part of the package for Algebraic Function Fields in one variable (PAFF)
++ It has been modified (very slightly) so that each time the "factor"
++ function is used, the variable related to the size of the field
++ over which the polynomial is factorized is reset. This is done in
++ order to be used with a "dynamic extension field" which size is not
++ fixed but set before calling the "factor" function and which is
++ parse by side effect to this package via the function "size". See
++ the local function "initialize" of this package.

FiniteFieldFactorizationWithSizeParseBySideEffect(K : FiniteFieldCategory,
    PolK : UnivariatePolynomialCategory(K)) : with

    factorSquareFree : PolK -> List(PolK)
    factorCantorZassenhaus : (PolK, NonNegativeInteger) -> List(PolK)
    factor : PolK -> Factored(PolK)
    factorUsingYun : PolK -> Factored(PolK)
    factorUsingMusser : PolK -> Factored(PolK)
    irreducible? : PolK -> Boolean

    == add

    import FiniteFieldSquareFreeDecomposition(K, PolK)
    p : NonNegativeInteger := characteristic()$K
    p' : NonNegativeInteger := p quo 2  -- used for odd p : (p-1)/2
    q : NonNegativeInteger := size()$K
    q' : NonNegativeInteger := q quo 2  -- used for odd q : (q-1)/2
    X : PolK := monomial(1, 1)
    primeKdim : NonNegativeInteger :=
        q_quo_p : NonNegativeInteger := q quo p ; e : NonNegativeInteger := 1
        while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)  
        e

    initialize(): Void() ==
        q : NonNegativeInteger := size()$K
        q' : NonNegativeInteger := q quo 2  -- used for odd q : (q-1)/2
    primeKdim : NonNegativeInteger :=
        q_quo_p : NonNegativeInteger := q quo p ; e :NonNegativeInteger := 1
        while q_quo_p > 1 repeat (e := e + 1 ; q_quo_p := q_quo_p quo p)  
        e

    exp(P : PolK, n : NonNegativeInteger, R : PolK) : PolK ==
        PP : PolK := P rem R ; Q : PolK := 1
        repeat
            if odd?(n) then Q := Q * PP rem R
            (n := n quo 2) = 0 => leave
            PP := PP * PP rem R
        return Q
CHAPTER 7. CHAPTER F

\textbf{pPowers}(P : \text{PolK}) : \text{PrimitiveArray}(\text{PolK}) == -- P is monic
\begin{verbatim}
n := degree(P)
result : \text{PrimitiveArray}(\text{PolK}) := \text{new}(n, 1)
result(1) := \text{Qi := Q := exp(X, p, P)}
for i in 2 .. n-1 repeat (Qi := Qi*Q rem P ; result(i) := Qi)
return result
\end{verbatim}

\textbf{pExp}(Q : \text{PolK}, \text{Xpowers : PrimitiveArray(\text{PolK})}) : \text{PolK} ==
\begin{verbatim}
Q' : \text{PolK} := 0
while Q \neq 0 repeat
\begin{verbatim}
Q' := Q' + \text{primeFrobenius(leadingCoefficient(Q))} * \text{Xpowers(degree(Q))}
Q := \text{reductum(Q)}
\end{verbatim}
return Q'
\end{verbatim}

\textbf{pTrace}(Q : \text{PolK}, d : \text{NonNegativeInteger}, P : \text{PolK}, \text{Xpowers : PrimitiveArray(\text{PolK})}) : \text{PolK} ==
\begin{verbatim}
Q : \text{PolK} := Q \text{ rem P}
result : \text{PolK} := Q
for i in 1 .. d-1 repeat result := Q + \text{pExp(result, Xpowers)}
return result rem P
\end{verbatim}

\textbf{random}(n : \text{NonNegativeInteger}) : \text{PolK} ==
\begin{verbatim}
repeat
\begin{verbatim}
if (deg := (random(n)$\text{Integer})$\text{NonNegativeInteger}) > 0 then leave
\end{verbatim}
repeat
\begin{verbatim}
if (x : K := random()$\text{K}) \neq 0 then leave
result : \text{PolK} :=
\begin{verbatim}
\text{monomial}(x, deg) + +[:\text{monomial}(random()$\text{K}, i) \text{ for i in 0 .. deg-1}]
\end{verbatim}
return result
\end{verbatim}
\end{verbatim}
\end{verbatim}

\textbf{internalFactorCZ}(P : \text{PolK}, -- P monic-squarefree
\text{d:NonNegativeInteger, Xpowers:PrimitiveArray(\text{PolK})}) : \text{List(\text{PolK})} ==
\begin{verbatim}
listOfFactors : \text{List(\text{PolK})} := [P]
degree(P) = d \Rightarrow \text{return listOfFactors}
result : \text{List(\text{PolK})} := []
pDim : \text{NonNegativeInteger} := d * \text{primeKdim}
Q : \text{PolK} := P
repeat
\begin{verbatim}
G := \text{pTrace(random(degree(Q)), pDim, Q, Xpowers)}
if p > 2 then G := \text{exp}(G, p', Q) - 1
Q1 := \text{gcd}(G, Q) ; d1 := degree(Q1)
if d1 > 0 and d1 < degree(Q) then
\begin{verbatim}
listOfFactors := \text{rest(listOfFactors)}
if d1 = d then result := \text{cons(Q1, result)}
else listOfFactors := \text{cons(Q1, listOfFactors)}
Q1 := Q \text{ quo Q1} ; d1 := degree(Q1)
if d1 = d then result := \text{cons(Q1, result)}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
else listOfFactors := cons(Q1, listOfFactors)
if empty?(listOfFactors) then leave
Q := first(listOfFactors)
return result

internalFactorSquareFree(P:PolK):List(PolK) == -- P is monic-squareFree
degree(P) = 1 => [P]
result : List(PolK) := []
Xpowers : PrimitiveArray(PolK) := pPowers(P)
S : PolK := Xpowers(1)
for j in 1..primeKdim-1 repeat S := pExp(S, Xpowers)
for i in 1 .. repeat -- S = X**(q**i) mod P
  if degree(R := gcd(S - X, P)) > 0 then
    result := concat(internalFactorCZ(R, i, Xpowers), result)
  if degree (P) = degree (R) then return result
  P := P quo R
  if i >= degree(P) quo 2 then return cons(P, result)
  for j in 0 .. degree(P)-1 repeat Xpowers(j):=Xpowers(j) rem P
  S := S rem P
else if i >= degree(P) quo 2 then return cons(P, result)
  for j in 1 .. primeKdim repeat S := pExp(S, Xpowers)

internalFactor(P:PolK, sqrfree:PolK -> Factored(PolK)) : Factored(PolK) ==
result : Factored(PolK)
if (d := minimumDegree(P)) > 0 then
  P := P quo monomial(1, d)
  result := primeFactor(X, d)
else
  result := 1
degree(P) = 0 => P * result
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
degree(P) = 1 => lcP::PolK * primeFactor(P, 1) * result
sqfP : Factored(PolK) := sqrfree(P)
for x in factors(sqfP) repeat
  xFactors : List(PolK) := internalFactorSquareFree(x.factor)
  result:=result * */[primeFactor(Q, x.exponent) for Q in xFactors]
return lcP::PolK * result

factorUsingYun(P : PolK) : Factored(PolK) == internalFactor(P, Yun)

factorUsingMusser(P : PolK) : Factored(PolK) == internalFactor(P, Musser)

factor(P : PolK) : Factored(PolK) ==
initialize()
factorUsingYun(P)

factorSquareFree(P : PolK) : List(PolK) ==
degree(P) = 0 => []
discriminant(P) = 0 => error("factorSquareFree : non quadratfrei")
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
return internalFactorSquareFree(P)

factorCantorZassenhaus(P : PolK, d : NonNegativeInteger) : List(PolK) ==
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
degree(P) = 1 => [P]
return internalFactorCZ(P, d, pPowers(P))

qExp(Q : PolK, XqPowers : PrimitiveArray(PolK)) : PolK ==
Q' : PolK := 0
while Q ^= 0 repeat
  Q' := Q' + leadingCoefficient(Q) * XqPowers(degree(Q))
  Q := reductum(Q)
return Q'

qPowers (Xq:PolK, P:PolK) : PrimitiveArray(PolK) == -- Xq = X**q mod P
n := degree(P)
result : PrimitiveArray(PolK) := new(n, 1)
result(1) := Q := Xq
for i in 2 .. n-1 repeat (Q := Q*Xq rem P ; result(i) := Q)
return result

discriminantTest?(P : PolK) : Boolean ==
(delta : K := discriminant(P)) = 0 => true
StickelbergerTest : Boolean := (delta ** q' = 1) = even?(degree(P))
return StickelbergerTest

evenCharacteristicIrreducible?(P : PolK) : Boolean ==
(n := degree(P)) = 0 => false
n = 1 => true
degree(gcd(P, D(P))) > 0 => false
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
S : PolK := exp(X, q, P)
if degree(gcd(S - X, P)) > 0 then
  return false
if n < 4 then return true
maxDegreeToTest : NonNegativeInteger := n quo 2
XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
for i in 2 .. maxDegreeToTest repeat
  S := qExp(S, XqPowers)
  if degree(gcd(S - X, P)) > 0 then
    return false
return true

oddCharacteristicIrreducible?(P : PolK) : Boolean ==
(n := degree(P)) = 0 => false
n = 1 => true
discriminantTest?(P) => false
if (lcP := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
S : PolK := exp(X, q, P)
if degree(gcd(S - X, P)) > 0 then

return false
if n < 6 then return true
maxDegreeToTest : NonNegativeInteger := n quo 3
XqPowers : PrimitiveArray(PolK) := qPowers(S, P)
for i in 2 .. maxDegreeToTest repeat
  S := qExp(S, XqPowers)
  if degree(gcd(S - X, P)) > 0 then
    return false
  return true
if p = 2 then
  irreducible?(P : PolK) : Boolean == evenCharacteristicIrreducible?(P)
else
  irreducible?(P : PolK) : Boolean == oddCharacteristicIrreducible?(P)

package FFF FiniteFieldFunctions

-- FiniteFieldFunctions.input --

)set break resume
)sys rm -f FiniteFieldFunctions.output
)spool FiniteFieldFunctions.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show FiniteFieldFunctions
--R
--R FiniteFieldFunctions(GF: FiniteFieldCategory) is a package constructor
--R Abbreviation for FiniteFieldFunctions is FFF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFF
FiniteFieldFunctions help

FiniteFieldFunctions examples

FiniteFieldFunctions(GF) is a package with functions concerning finite extension fields of the finite ground field GF, e.g. Zech logarithms.

See Also:
  o )show FiniteFieldFunctions

Exports:
createLowComplexityNormalBasis  createLowComplexityTable  createMultiplicationMatrix
createMultiplicationTable  createZechTable  sizeMultiplication

— package FFF FiniteFieldFunctions —

)abbrev package FFF FiniteFieldFunctions ++ Author: J. Grabmeier, A. Scheerhorn ++ Date Created: 21 March 1991 ++ Date Last Updated: 31 March 1991 ++ References:
++ Lidl, R. & Niederreiter, H., "Finite Fields",
++ Encycl. of Math. 20, Addison-Wesley, 1983
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ FiniteFieldFunctions(GF) is a package with functions
++ concerning finite extension fields of the finite ground field GF,
++ e.g. Zech logarithms.

FiniteFieldFunctions(GF): Exports == Implementation where
GF : FiniteFieldCategory   -- the ground field

PI   ==> PositiveInteger
NNI  ==> NonNegativeInteger
I    ==> Integer
SI   ==> SingleInteger
SUP  ==> SparseUnivariatePolynomial GF
V    ==> Vector
M    ==> Matrix
L    ==> List
OUT  ==> OutputForm
SAE  ==> SimpleAlgebraicExtension
ARR  ==> PrimitiveArray(SI)
TERM == Record(value:GF,index:SI)
MM   ==> ModMonic(GF,SUP)
PF   ==> PrimeField

Exports == with

createZechTable: SUP -> ARR
++ createZechTable(f) generates a Zech logarithm table for the cyclic
++ group representation of a extension of the ground field by the
++ primitive polynomial f(x), i.e. \spad{Z(i)},
++ defined by x**Z(i) = 1+x**i is stored at index i.
++ This is needed in particular
++ to perform addition of field elements in finite fields represented
++ in this way. See \spadtype{FFCGP}, \spadtype{FFCGX}.
createMultiplicationTable: SUP -> V L TERM
++ createMultiplicationTable(f) generates a multiplication
++ table for the normal basis of the field extension determined
++ by f. This is needed to perform multiplications
++ between elements represented as coordinate vectors to this basis.
++ See \spadtype{FFNBP}, \spadtype{FFNBX}.
createMultiplicationMatrix: V L TERM -> M GF
++ createMultiplicationMatrix(m) forms the multiplication table
++ m into a matrix over the ground field.
-- only useful for the user to visualise the multiplication table
-- in a nice form
sizeMultiplication: V L TERM -> NNI
++ sizeMultiplication(m) returns the number of entries
++ of the multiplication table m.
-- the time of the multiplication of field elements depends
-- on this size
createLowComplexityTable: PI -> Union(Vector List TERM,"failed")
++ createLowComplexityTable(n) tries to find
++ a low complexity normal basis of degree n over GF
++ and returns its multiplication matrix
++ Fails, if it does not find a low complexity basis
createLowComplexityNormalBasis: PI -> Union(SUP, V L TERM)
++ createLowComplexityNormalBasis(n) tries to find a
++ a low complexity normal basis of degree n over GF
++ and returns its multiplication matrix
++ If no low complexity basis is found it calls
++ \axiomFunFrom{createNormalPoly}{FiniteFieldPolynomialPackage}(n)
++ to produce a normal polynomial of degree n over GF

Implementation ==> add

createLowComplexityNormalBasis(n) ==
  (u:=createLowComplexityTable(n)) case "failed" =>
  createNormalPoly(n)$FiniteFieldPolynomialPackage(GF)
  u::(V L TERM)
  -- try to find a low complexity normal basis multiplication table
  -- of the field of extension degree n
  -- the algorithm is from:
  -- Wassermann A., Konstruktion von Normalbasen,
createLowComplexityTable(n) ==
  q:=size()$GF
  -- this algorithm works only for prime fields
  p:=characteristic()$GF
  -- search of a suitable parameter k
  k:NNI:=0
  for i in 1..n-1 while (k=0) repeat
    if prime?(i*n+1) and not(p = (i*n+1)) then
      primitive?(q::PF(i*n+1))$PF(i*n+1) =>$PF(i*n+1) =>
      a:NNI:=1
k := i

\[ t1 := PF(k \cdot n + 1) \cdot (q \cdot PF(k \cdot n + 1))^n \]

\[ \text{gcd}(n, a = \text{discreteLog}(q \cdot PF(n \cdot i + 1)) \cdot PF(n \cdot i + 1)) \]

k := i

\[ t1 := \text{primitiveElement()} \cdot PF(k \cdot n + 1)^n \]

k = 0 => "failed"

-- initialize some start values

multmat : M PF(p) := zero(n, n)

p1 := (k \cdot n + 1)
pkn := q \cdot PF(p1)
t := t1 pretend PF(p1)

if odd?(k) then

\[ j_t := \text{positiveRemainder}((k - a) \text{ quo } 2, k) + 1 \]

else

\[ j_t := 1 \]

\[ v_t := (k \text{ quo } 2) + 1 \]

-- compute matrix

vec : Vector I := zero(p1 pretend NNI)

for x in 1..k repeat

for l in 1..n repeat

\[ \text{vec.((t}^{x-1} \cdot pkn^{l-1}) \text{ pretend Integer} + 1) := \text{positiveRemainder}(l, p1) \]

for j in 1..n repeat

\[ \text{vec.(help pretend I + 1)} \]

for v in 1..k repeat

if (j^=j_t) or (v^=v_t) then

\[ \text{help := PF(p1)} := t^{x-1} \cdot pkn^{j-1} \cdot p1 \]

\[ \text{setelt(lvj, v, j, vec.(help pretend I + 1))} \]

for j in 1..n repeat

\[ \text{setelt(lvj, v, j, vec.(help pretend I + 1))} \]

for j in 1..n repeat

\[ \text{setelt(lvj, v, j, vec.(help pretend I + 1))} \]

for v in 1..k repeat

if v^=v_t then

\[ \text{setelt(multmat, j, lvj, elt(multmat, j, lvj) + 1)} \]

\[ \text{setelt(multmat, j, lvj, elt(multmat, j, lvj) + 1)} \]

\[ \text{setelt(multmat, j, lvj, elt(multmat, j, lvj) + 1)} \]

-- multmat

m := nrows(multmat) \cdot (M PF(p))
multable : V L TERM := new(m, nil() \cdot (L TERM)) \cdot (V L TERM)

for i in 1..m repeat

for j in 1..k repeat

if v^=v_t then

\[ \text{setelt(multmat, j, lvj, elt(multmat, j, lvj) + 1)} \]

\[ \text{setelt(multmat, j, lvj, elt(multmat, j, lvj) + 1)} \]

\[ \text{setelt(multmat, j, lvj, elt(multmat, j, lvj) + 1)} \]

-- multmat

m := nrows(multmat) \cdot (M PF(p))
multable : V L TERM := new(m, nil() \cdot (L TERM)) \cdot (V L TERM)

for i in 1..m repeat

for j in (1..I) \cdot (m::I) repeat

if (v,j = 0) then

\[ \text{-- take -v.j to get trace 1 instead of -1} \]

\[ \text{term := (((convert(-v.j) \cdot GF, (j-2) pretend SI) \cdot TERM)} \]
\begin{verbatim}
1 := cons(term,1)$(L TERM)
qsetelt_!(multtable,i,copy 1)$(V L TERM)
multable

sizeMultiplication(m) ==
s:NNI:=0
for i in 1..#m repeat
   s := s + #(m.i)
s
createMultiplicationTable(f:SUP) ==
sizeGF:NNI:=size()$GF -- the size of the ground field
m:PI:=degree(f)$SUP pretend PI
m=1 =>
   [[[-coefficient(f,0)$SUP,(-1)::SI]$TERM]$(L TERM)]::(V L TERM)
m1:=m-1
-- initialize basis change matrices
setPoly(f)$MM
e:=reduce(monomial(1,1)$SUP)$MM ** sizeGF
w:=1$MM
qpow:PrimitiveArray(MM):=new(m,0)
qpow.0:=1$MM
for i in 1..m1 repeat
   qpow.i:=(w:=w*e)
   -- qpow.i = x**(i*q)
qexp:PrimitiveArray(MM):=new(m,0)
qexp.0:=reduce(monomial(1,1)$SUP)$MM
mat:M GF:=zero(m,m)$(M GF)
qsetelt_!(mat,2,1,1$GF)$(M GF)
h:=qpow.1
qexp.1:=h
setColumn_!(mat,2,Vectorise(h)$MM)$M GF)
for i in 2..m1 repeat
   g:=0$MM
   while h ^= 0 repeat
      g:=g + leadingCoefficient(h) * qpow.degree(h)$MM
      h:=reductum(h)$MM
      qexp.1:=g
      setColumn_!(mat,i+1,Vectorise(h:=g)$MM)$M GF)
   -- loop invariant: qexp.i = x**(q**i)
   mat1:=inverse(mat)$M GF)
   mat1 = "failed" =>
      error "createMultiplicationTable: polynomial must be normal"
   mat:=mat1 :: (M GF)
-- initialize multiplication table
multable:V L TERM:=new(m,nil()$(L TERM))$V L TERM
for i in 1..m repeat
   1:L TERM:=nil()$(L TERM)
v:V GF:=mat *$(M GF) Vectorise(qexp.(i-1) *$MM qexp.0)$MM
for j in (1::SI)..(m::SI) repeat
\end{verbatim}
if (v.j ^= 0$GF) then
  term:TERM:=[(v.j),j-(2::SI)]$TERM
 l:=cons(term,l)$(L TERM)
 qsetelt_!(multtable,i,copy l)$(V L TERM)
 multtable

createZechTable(f:SUP) ==
  sizeGF:NNI:=size()$GF -- the size of the ground field
  m:=degree(f)$SUP::PI
  qm1:SI:=(sizeGF ** m -1) pretend SI
  zechlog:ARR:=new(((sizeGF ** m + 1) quo 2)::NNI,-1::SI)$ARR
  helparr:ARR:=new(sizeGF ** m::NNI,0$SI)$ARR
  primElement:=reduce(monomial(1,1)$SUP)$SAE(GF,SUP,f)
  a:=primElement
  for i in 1..qm1-1 repeat
    helparr.(lookup(a -$SAE(GF,SUP,f) 1$SAE(GF,SUP,f)_
      )$SAE(GF,SUP,f)):=i::SI
    a:=a * primElement
  characteristic() = 2 =>
    a:=primElement
    for i in 1..(qm1 quo 2) repeat
      zechlog.i:=helparr.lookup(a)$SAE(GF,SUP,f)
      a:=a * primElement
      zechlog
      a:=1$SAE(GF,SUP,f)
    for i in 0..((qm1-2) quo 2) repeat
      zechlog.i:=helparr.lookup(a)$SAE(GF,SUP,f)
      a:=a * primElement
      zechlog

createMultiplicationMatrix(m) ==
  n:NNI:=#m
  mat:M GF:=zero(n,n)$(M GF)
  for i in 1..n repeat
    for t in m.i repeat
      qsetelt_!(mat,i,t.index+2,t.value)
  mat

--- FFF.dotabb ---

"FFF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFF" -> "PFECAT"
package FFHOM FiniteFieldHomomorphisms

— FiniteFieldHomomorphisms.input —

)set break resume
)sys rm -f FiniteFieldHomomorphisms.output
)spool FiniteFieldHomomorphisms.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show FiniteFieldHomomorphisms
--R
--R FiniteFieldHomomorphisms(F1: FiniteAlgebraicExtensionField(GF),GF: FiniteFieldCategory,F2: FiniteAlgebraicExtensionField(GF)) is a package constructor
--R Abbreviation for FiniteFieldHomomorphisms is FFHOM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFHOM
--R
--R-------------------------------- Operation --------------------------------
--R coerc : F1 -> F2 coerc : F2 -> F1
--R
--E 1

)spool
)lisp (bye)

— FiniteFieldHomomorphisms.help —

====================================================================
FiniteFieldHomomorphisms examples
====================================================================

FiniteFieldHomomorphisms(F1,GF,F2) exports coercion functions of elements between the fields F1 and F2, which both must be finite simple algebraic extensions of the finite ground field GF.

See Also:
o )show FiniteFieldHomomorphisms

———
FiniteFieldHomomorphisms (FFHOM)

Exports:

coerce

-- package FFHOM FiniteFieldHomomorphisms --

)abbrev package FFHOM FiniteFieldHomomorphisms
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encyclopaedia of Mathematics and
++ J. Grabmeier, A. Scheerhorn: Finite Fields in AXIOM.
++ Description:
++ FiniteFieldHomomorphisms(F1,GF,F2) exports coercion functions of
++ elements between the fields F1 and F2, which both must be
++ finite simple algebraic extensions of the finite ground field GF.

FiniteFieldHomomorphisms(F1,GF,F2): Exports == Implementation where
   F1: FiniteAlgebraicExtensionField(GF)
   GF: FiniteFieldCategory
   F2: FiniteAlgebraicExtensionField(GF)
   -- the homorphism can only convert elements w.r.t. the last extension .
   -- Adding a function 'groundField()' which returns the groundfield of GF
   -- as a variable of type FiniteFieldCategory in the new compiler, one
   -- could build up 'convert' recursively to get an homomorphism w.r.t
   -- the whole extension.

I    ==>  Integer
NNI  ==>  NonNegativeInteger
SI   ==>  SingleInteger
PI   ==>  PositiveInteger
SUP  ==>  SparseUnivariatePolynomial
M    ==>  Matrix GF
FFP  ==>  FiniteFieldExtensionByPolynomial
FFPOL2 ==> FiniteFieldPolynomialPackage2
FFPOLY ==> FiniteFieldPolynomialPackage
OUT ==> OutputForm

Exports ==> with

coerce: F1 -> F2
  ++ coerce(x) is the homomorphic image of x from
  ++ F1 in F2. Thus coerce is a
  ++ field homomorphism between the fields extensions
  ++ F1 and F2 both over ground field GF
  ++ (the second argument to the package).
  ++ Error: if the extension degree of F1 doesn't divide
  ++ the extension degree of F2.
  ++ Note that the other coercion function in the
  ++ \spadtype{FiniteFieldHomomorphisms} is a left inverse.

coerce: F2 -> F1
  ++ coerce(x) is the homomorphic image of x from
  ++ F2 in F1, where coerce is a
  ++ field homomorphism between the fields extensions
  ++ F2 and F1 both over ground field GF
  ++ (the second argument to the package).
  ++ Error: if the extension degree of F2 doesn't divide
  ++ the extension degree of F1.
  ++ Note that the other coercion function in the
  ++ \spadtype{FiniteFieldHomomorphisms} is a left inverse.
-- coerce(coerce(x:F1)@F2)@F1 = x and coerce(coerce(y:F2)@F1)@F2 = y

Implementation ==> add

-- global variables ===================================================

degree1:NNI:= extensionDegree()$F1
degree2:NNI:= extensionDegree()$F2
-- the degrees of the last extension

-- a necessary condition for the one field being an subfield of
-- the other one is, that the respective extension degrees are
-- multiples
if max(degree1,degree2) rem min(degree1,degree2) ^= 0 then
  error "FFHOM: one extension degree must divide the other one"

conMat1to2:M:= zero(degree2,degree1)$M
-- conversion Matix for the conversion direction F1 -> F2
conMat2to1:M:= zero(degree1,degree2)$M
-- conversion Matix for the conversion direction F2 -> F1

repType1:=representationType()$F1
repType2:=representationType()$F2
-- the representation types of the fields

init?:Boolean:=true
-- gets false after initialization

defPol1:=definingPolynomial()$F1
defPol2:=definingPolynomial()$F2
-- the defining polynomials of the fields

-- functions ===========================================================

compare: (SUP GF,SUP GF) -> Boolean
-- compares two polynomials

convertWRTsameDefPol12: F1 -> F2
convertWRTsameDefPol21: F2 -> F1
-- homomorphism if the last extension of F1 and F2 was build up
-- using the same defining polynomials

convertWRTdifferentDefPol12: F1 -> F2
convertWRTdifferentDefPol21: F2 -> F1
-- homomorphism if the last extension of F1 and F2 was build up
-- with different defining polynomials

initialize: () -> Void
-- computes the conversion matrices

compare(g:(SUP GF),f:(SUP GF)) ==
der degree(f)$(SUP GF) >$NNI degree(g)$(SUP GF) => true
der degree(f)$(SUP GF) <$NNI degree(g)$(SUP GF) => false
equal:Integer:=0
for i in degree(f)$(SUP GF)..0 by -1 while equal=0 repeat
  not zero?(coefficient(f,i)$(SUP GF))$GF and _
  zero?(coefficient(g,i)$(SUP GF))$GF => equal:=1
  not zero?(coefficient(g,i)$(SUP GF))$GF and _
  zero?(coefficient(f,i)$(SUP GF))$GF => equal:=(-1)
(f1:=lookup(coefficient(f,i)$(SUP GF))$GF) >$PositiveInteger _
(g1:=lookup(coefficient(g,i)$(SUP GF))$GF) =>$PositiveInteger g1 => equal:=1
f1 <$PositiveInteger g1 => equal:=(-1)
equal=1 => true
false

initialize() ==
-- 1) in the case of equal def. polynomials initialize is called only
-- if one of the rep. types is "normal" and the other one is "polynomial"
-- we have to compute the basis change matrix 'mat', which i-th
-- column are the coordinates of a**((q**i), the i-th component of
-- the normal basis ('a' the root of the def. polynomial and q the
defPol1 = $(SUP GF) defPol2 =>
-- new code using reducedQPowers
mat:=zero(degree1,degree1)$M
arr:=reducedQPowers(defPol1)$FFPOLY(GF)
for i in 1..degree1 repeat
  setColumn_!(mat,i,vectorise(arr.(i-1),degree1)$SUP(GF))$M
-- old code
-- here one of the representation types must be "normal"
-- a:=basis()$FFP(GF,defPol1).2 -- the root of the def. polynomial
-- setColumn_!(mat,1,coordinates(a)$FFP(GF,defPol1))$M
-- for i in 2..degree1 repeat
--  a:= a **$FFP(GF,defPol1) size()$GF
--  setColumn_!(mat,i,coordinates(a)$FFP(GF,defPol1))$M
-- for the direction "normal" -> "polynomial" we have to multiply the
-- coordinate vector of an element of the normal basis field with
-- the matrix 'mat'. In this case 'mat' is the correct conversion
-- matrix for the conversion of F1 to F2, its inverse the correct
-- inversion matrix for the conversion of F2 to F1
repType1 = "normal" => -- repType2 = "polynomial"
conMat1to2:=copy(mat)
conMat2to1:=copy(inverse(mat)$M :: M)
-- we finish the function for one case, hence reset initialization flag
init? := false
void()$Void
-- print("'normal' <=> 'polynomial' matrices initialized":OUT)
-- in the other case we have to change the matrices
-- repType2 = "normal" and repType1 = "polynomial"
conMat2to1:=copy(mat)
conMat1to2:=copy(inverse(mat)$M :: M)
-- print("'normal' <=> 'polynomial' matrices initialized":OUT)
-- we finish the function for one case, hence reset initialization flag
init? := false
void()$Void
-- 2) in the case of different def. polynomials we have to order the
-- fields to get the same isomorphism, if the package is called with
-- the fields F1 and F2 swapped.
dPbig:= defPol2
rTbig:= repType2
dPsmall:= defPol1
rTsmall:= repType1
degbig:=degree2
degsmall:=degree1
if compare(defPol2,defPol1) then
  degsmall:=degree2
degbig:=degree1
dPbig:= defPol1
rTbig:= repType1
dPsmall:= defPol2
rTsmall:= repType2
-- 3) in every case we need a conversion between the polynomial
-- represented fields. Therefore we compute 'root' as a root of the
-- 'smaller' def. polynomial in the 'bigger' field.
-- We compute the matrix 'matsb', which i-th column are the coordinates
-- of the (i-1)-th power of root, i=1..degsmall. Multiplying a
-- coordinate vector of an element of the 'smaller' field by this
-- matrix, we got the coordinates of the corresponding element in the
-- 'bigger' field.
-- compute the root of dPsmall in the 'big' field
root:=rootOfIrreduciblePoly(dPsmall)$FFPOL2(FFP(GF,dPbig),GF)
-- set up matrix for polynomial conversion
matsb:=zero(degbig,degsmall)$M
qsetelt_!(matsb,1,1,1$GF)$M
a:=root
for i in 2..degsmall repeat
  setColumn_!(matsb,i,coordinates(a)$FFP(GF,dPbig))$M
  a := a *$FFP(GF,dPbig) root
-- the conversion from 'big' to 'small': we can't invert matsb
-- directly, because it has degbig rows and degsmall columns and
-- may be no square matrix. Therefore we construct a square matrix
-- mat from degsmall linear independent rows of matsb and invert it.
-- Now we get the conversion matrix 'matbs' for the conversion from
-- 'big' to 'small' by putting the columns of mat at the indices
-- of the linear independent rows of matsb to columns of matbs.
ra:I:=1 -- the rank
mat:M:=transpose(row(matsb,1))$M -- has already rank 1
rowind:I:=2
iVec:Vector I:=new(degsmall,1$I)$(Vector I)
while ra < degsmall repeat
  if rank(vertConcat(mat,transpose(row(matsb,rowind))$M)$M)$M > ra then
    mat:=vertConcat(mat,transpose(row(matsb,rowind))$M)$M
    ra:=ra+1
    iVec.ra := rowind
  end if
  rowind:=rowind + 1
end while
mat:=inverse(mat)$M :: M
matbs:=zero(degsmall,degbig)$M
for i in 1..degsmall repeat
  setColumn_!(matbs,i,column(mat,i)$M)$M
end for
-- 4) if the 'bigger' field is "normal" we have to compose the
-- polynomial conversion with a conversion from polynomial to normal
-- between the FFP(GF,dPbig) and FFNPB(GF,dPbig) the 'bigger'
-- field. Therefore we compute a conversion matrix 'mat' as in 1)
-- Multiplying with the inverse of 'mat' yields the desired
-- conversion from polynomial to normal. Multiplying this matrix by
-- the above computed 'matsb' we got the matrix for converting form
-- 'small polynomial' to 'big normal'.
-- set up matrix 'mat' for polynomial to normal
if rTbig = "normal" then


arr:=reducedQPowers(dPbig)$FFPOLY(GF)
mat:=zero(degbig,degbig)$M
for i in 1..degbig repeat
    setColumn_!(mat,i,vectorise(arr.(i-1),degbig)$SUP(GF))$M
-- old code
--a:=basis()$FFP(GF,dPbig).2 -- the root of the def.Polynomial
--setColumn_!(mat,1,coordinates(a)$FFP(GF,dPbig))$M
--for i in 2..degbig repeat
-- a:= a **$FFP(GF,dPbig) size()$GF
-- setColumn_!(mat,i,coordinates(a)$FFP(GF,dPbig))$M
-- print(inverse(mat)$M::OUT)
matbs:=(inverse(mat)$M :: M) * matsb
-- print("inv *..":OUT)
matbs:=matbs * mat
-- 5) if the 'smaller' field is "normal" we have first to convert
-- from 'small normal' to 'small polynomial', that is from
-- FFPNBP(GF,dPsmall) to FFP(GF,dPsmall). Therefore we compute a
-- conversion matrix 'mat' as in 1). Multiplying with 'mat'
-- yields the desired conversion from normal to polynomial.
-- Multiplying the above computed 'matsb' with 'mat' we got the
-- matrix for converting form 'small normal' to 'big normal'.
-- set up matrix 'mat' for normal to polynomial
if rTsmall = "normal" then
    arr:=reducedQPowers(dPsmall)$FFPOLY(GF)
    mat:=zero(degsmall,degsmall)$M
    for i in 1..degsmall repeat
        setColumn_!(mat,i,vectorise(arr.(i-1),degsmall)$SUP(GF))$M
    -- old code
    --b:FFP(GF,dPsmall):=basis()$FFP(GF,dPsmall).2
    --setColumn_!(mat,1,coordinates(b)$FFP(GF,dPsmall))$M
    --for i in 2..degsmall repeat
    -- b:= b **$FFP(GF,dPsmall) size()$GF
    -- setColumn_!(mat,i,coordinates(b)$FFP(GF,dPsmall))$M
    -- print(mat::OUT)
    matsb:= matsb * mat
    matbs:=(inverse(mat) :: M) * matbs
    -- now 'matsb' is the corret conversion matrix for 'small' to 'big'
    -- and 'matbs' the corret one for 'big' to 'small'.
    -- depending on the above ordering the conversion matrices are
    -- initialized
    dPbig =$(SUP GF) defPol2 =>
        conMat1to2 :=matsb
        conMat2to1 :=matbs
        -- print(conMat1to2::OUT)
        -- print(conMat2to1::OUT)
        -- print("conversion matrices initialized":OUT)
        --we finish the function for one case, hence reset initialization flag
        init? := false
        void()$Void
        conMat1to2 :=matbs
conMat2to1 := matsb
-- print(conMat1to2::OUT)
-- print(conMat2to1::OUT)
-- print("conversion matrices initialized":::OUT)
--we finish the function for one case, hence reset initialization flag
init? := false
void()$Void

coerce(x:F1) ==
inGroundField?(x)$F1 => retract(x)$F1 :: F2
-- if x is already in GF then we can use a simple coercion
defPol1 =$$(SUP GF) defPol2 => convertWRTsameDefPol12(x)
convertWRTdifferentDefPol12(x)

convertWRTsameDefPol12(x:F1) ==
repType1 = repType2 => x pretend F2
-- same groundfields, same defining polynomials, same
-- representation types --> F1 = F2, x is already in F2
repType1 = "cyclic" =>
x = 0$F1 => 0$F2
-- the SI corresponding to the cyclic representation is the exponent of
-- the primitiveElement, therefore we exponentiate the primitiveElement
-- of F2 by it.
primitiveElement()$F2 **$F2 (x pretend SI)
repType2 = "cyclic" =>
x = 0$F1 => 0$F2
-- to get the exponent, we have to take the discrete logarithm of the
-- element in the given field.
(discreteLog(x)$F1 pretend SI) pretend F2
-- here one of the representation types is "normal"
if init? then initialize()
-- here a conversion matrix is necessary, (see initialize())
represents(conMat1to2 *$(Matrix GF) coordinates(x)$F1)$F2

convertWRTdifferentDefPol12(x:F1) ==
if init? then initialize()
-- if we want to convert into a 'smaller' field, we have to test,
-- whether the element is in the subfield of the 'bigger' field, which
-- corresponds to the 'smaller' field
if degree1 > degree2 then
  if positiveRemainder(degree2,degree(x)$F1)$$= 0 then
    error "coerce: element doesn't belong to smaller field"
    represents(conMat1to2 *$(Matrix GF) coordinates(x)$F1)$F2

-- the three functions below equal the three functions above up to
-- '1' exchanged by '2' in all domain and variable names

coerce(x:F2) ==
inGroundField?(x)$F2 => retract(x)$F2 :: F1
-- if x is already in GF then we can use a simple coercion
defPol1 =$\{\text{SUP GF}\} \text{defPol1} \Rightarrow \text{convertWRTsameDefPol21}(x)

convertWRTsameDefPol21(x:F2) ==
repType1 = repType2 => x pretend F1
-- same groundfields, same defining polynomials,
-- same representation types --> F1 = F2, that is:
-- x is already in F1
repType2 = "cyclic" =>
x = 0$F2 => 0$F1
primitiveElement()$F1 **$F1 (x pretend SI)
repType1 = "cyclic" =>
x = 0$F2 => 0$F1
(discreteLog(x)$F2 pretend SI) pretend F1
-- here one of the representation types is "normal"
if init? then initialize()
represents(conMat2to1 *$(Matrix GF) coordinates(x)$F2)$F1

convertWRTdifferentDefPol21(x:F2) ==
if init? then initialize()
if degree2 > degree1 then
  if positiveRemainder(degree1,degree(x)$F2) ^= 0 then
    error "coerce: element doesn't belong to smaller field"
  represents(conMat2to1 *$(Matrix GF) coordinates(x)$F2)$F1

— FFHOM.dotabb —

"FFHOM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFHOM"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"FFHOM" \rightarrow "FAXF"

package FFPOLY FiniteFieldPolynomialPackage

— FiniteFieldPolynomialPackage.input —

)set break resume
)sys rm -f FiniteFieldPolynomialPackage.output
)spool FiniteFieldPolynomialPackage.output
This package provides a number of functions for generating, counting and testing irreducible, normal, primitive, random polynomials over finite fields.
See Also:
- )show FiniteFieldPolynomialPackage

FiniteFieldPolynomialPackage (FFPOLY)

Exports:
- createIrreduciblePoly
- createPrimitiveNormalPoly
- nextIrreduciblePoly
- nextPrimitiveNormalPoly
- numberOfIrreduciblePoly
- primitive?
- createNormalPoly
- createPrimitivePoly
- nextNormalPoly
- nextPrimitivePoly
- normal?
- createNormalPrimitivePoly
- leastAffineMultiple
- nextNormalPrimitivePoly
- normal?
- createPrimitiveNormalPoly
- random
- numberOfIrreduciblePoly
- reducedQPowers
- numberOfNormalPoly
- numberOfPrimitivePoly
- nextPrimitiveNormalPoly

— package FFPOLY FiniteFieldPolynomialPackage —

)abbrev package FFPOLY FiniteFieldPolynomialPackage
++ Author: A. Bouyer, J. Grabmeier, A. Scheerhorn, R. Sutor, B. Trager
++ Date Created: January 1991
++ Date Last Updated: 1 June 1994
++ References:
++ [LS] Lenstra, H. W. & Schoof, R. J., "Primitive Normal Bases
++ Encycl. of Math. 20, Addison-Wesley, 1983
++ J. Grabmeier, A. Scheerhorn: Finite Fields in Axiom.
++ Description:
++ This package provides a number of functions for generating, counting
++ and testing irreducible, normal, primitive, random polynomials
++ over finite fields.
FiniteFieldPolynomialPackage GF : Exports == Implementation where

GF : FiniteFieldCategory

I ==> Integer
L ==> List
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
Rec ==> Record(expnt:NNI, coeff:GF)
Repr ==> L Rec
SUP ==> SparseUnivariatePolynomial GF

Exports ==> with
  -- qEulerPhiCyclotomic : PI -> PI
  -- ++ qEulerPhiCyclotomic(n)$FFPOLY(GF) yields the q-Euler’s function
  -- ++ of the n-th cyclotomic polynomial over the field GF of
  -- ++ order q (cf. [LN] p.122);
  -- ++ error if n is a multiple of the field characteristic.
  primitive? : SUP -> Boolean
  ++ primitive?(f) tests whether the polynomial f over a finite
  ++ field is primitive, i.e. all its roots are primitive.
  normal? : SUP -> Boolean
  ++ normal?(f) tests whether the polynomial f over a finite field is
  ++ normal, i.e. its roots are linearly independent over the field.
  numberOfIrreduciblePoly : PI -> PI
  ++ numberOfIrreduciblePoly(n)$FFPOLY(GF) yields the number of
  ++ monic irreducible univariate polynomials of degree n
  ++ over the finite field GF.
  numberOfPrimitivePoly : PI -> PI
  ++ numberOfPrimitivePoly(n)$FFPOLY(GF) yields the number of
  ++ primitive polynomials of degree n over the finite field GF.
  numberOfNormalPoly : PI -> PI
  ++ numberOfNormalPoly(n)$FFPOLY(GF) yields the number of
  ++ normal polynomials of degree n over the finite field GF.
  createIrreduciblePoly : PI -> SUP
  ++ createIrreduciblePoly(n)$FFPOLY(GF) generates a monic irreducible
  ++ univariate polynomial of degree n over the finite field GF.
  createPrimitivePoly : PI -> SUP
  ++ createPrimitivePoly(n)$FFPOLY(GF) generates a primitive polynomial
  ++ of degree n over the finite field GF.
  createNormalPoly : PI -> SUP
  ++ createNormalPoly(n)$FFPOLY(GF) generates a normal polynomial
  ++ of degree n over the finite field GF.
  createNormalPrimitivePoly : PI -> SUP
  ++ createNormalPrimitivePoly(n)$FFPOLY(GF) generates a normal and
  ++ primitive polynomial of degree n over the field GF.
  ++ Note that this function is equivalent to createPrimitiveNormalPoly(n)
  createPrimitiveNormalPoly : PI -> SUP
  ++ createPrimitiveNormalPoly(n)$FFPOLY(GF) generates a normal and
CHAPTER 7. CHAPTER F

++ primitive polynomial of degree n over the field GF.
++ polynomial of degree n over the field GF.
nextIrreduciblePoly : SUP -> Union(SUP, "failed")
++ nextIrreduciblePoly(f) yields the next monic irreducible polynomial
++ over a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \texttt{f < g} if
++ the number of monomials of f is less
++ than this number for g.
++ If f and g have the same number of monomials,
++ the lists of exponents are compared lexicographically.
++ If these lists are also equal, the lists of coefficients
++ are compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
nextPrimitivePoly : SUP -> Union(SUP, "failed")
++ nextPrimitivePoly(f) yields the next primitive polynomial over
++ a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \texttt{f < g} if
++ the number of monomials of f is less than this number for g.
++ If these values are equal, then \texttt{f < g} if
++ the number of monomials of f is less than that for g or if
++ the lists of exponents of f are lexicographically less than the
++ corresponding list for g.
++ If these lists are also equal, the lists of coefficients are
++ compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
nextNormalPoly : SUP -> Union(SUP, "failed")
++ nextNormalPoly(f) yields the next normal polynomial over
++ a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \texttt{f < g} if
++ the number of monomials of f is less than that for g.
++ In case these numbers are equal, \texttt{f < g} if
++ the number of monomials of f is less than that for g or if
++ the list of exponents of f are lexicographically less than the
++ corresponding list for g.
++ If these lists are also equal, the lists of coefficients are
++ compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
nextNormalPrimitivePoly : SUP -> Union(SUP, "failed")
++ nextNormalPrimitivePoly(f) yields the next normal primitive polynomial
++ over a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if the lookup of the constant
++ term of f is less than
++ this number for g or if
++ lookup of the coefficient of the term of degree n-1
++ of f is less than this number for g.
++ Otherwise, \spad{f < g}
++ if the number of monomials of f is less than
++ that for g or if the lists of exponents for f are
++ lexicographically less than those for g.
++ If these lists are also equal, the lists of coefficients are
++ compared according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
++ This operation is equivalent to nextPrimitiveNormalPoly(f).

nextPrimitiveNormalPoly : SUP -> Union(SUP, "failed")
++ nextPrimitiveNormalPoly(f) yields the next primitive normal polynomial
++ over a finite field GF of the same degree as f in the following
++ order, or "failed" if there are no greater ones.
++ Error: if f has degree 0.
++ Note that the input polynomial f is made monic.
++ Also, \spad{f < g} if the lookup of the
++ constant term of f is less than
++ lookup of the coefficient of the term of degree n-1
++ of f is less than this number for g.
++ If these numbers are equals, \spad{f < g}
++ if the number of monomials of f is less than
++ that for g, or if the lists of exponents for f are lexicographically
++ less than those for g.
++ If these lists are also equal, the lists of coefficients are
++ coefficients according to the lexicographic ordering induced by
++ the ordering of the elements of GF given by lookup.
++ This operation is equivalent to nextNormalPrimitivePoly(f).

-- random : () -> SUP
-- ++ random()$FFPOLY(GF) generates a random monic polynomial
-- ++ of random degree over the field GF

random : PI -> SUP
++ random(n)$FFPOLY(GF) generates a random monic polynomial
++ of degree n over the finite field GF.

random : (PI, PI) -> SUP
++ random(m,n)$FFPOLY(GF) generates a random monic polynomial
++ of degree d over the finite field GF, d between m and n.

leastAffineMultiple : SUP -> SUP
++ leastAffineMultiple(f) computes the least affine polynomial which
++ is divisible by the polynomial f over the finite field GF,
++ i.e. a polynomial whose exponents are 0 or a power of q, the
++ size of GF.
reducedQPowers: SUP -> PrimitiveArray SUP
++ reducedQPowers(f)
++ generates \spad{[x,x**q,x**(q**2),...,x**(q**(n-1))]} 
++ reduced modulo f where \spad{q = size()$GF} and \spad{n = degree f}.
--
-- we intend to implement also the functions
-- cyclotomicPoly: PI -> SUP, order: SUP -> PI,
-- and maybe a new version of irreducible?

Implementation ==> add

import IntegerNumberTheoryFunctions
import DistinctDegreeFactorize(GF, SUP)

MM := ModMonic(GF, SUP)

sizeGF : PI := size()$GF :: PI

revListToSUP(l:Repr):SUP ==
  newl:Repr := empty()
  -- cannot use map since copy for Record is an XLAM
  for t in l repeat newl := cons(copy t, newl)
  newl pretend SUP

listToSUP(l:Repr):SUP ==
  newl:Repr := [copy t for t in l]
  newl pretend SUP

nextSubset : (L NNI, NNI) -> Union(L NNI, "failed")
-- for a list s of length m with 1 <= s.1 < ... < s.m <= bound,
-- nextSubset(s, bound) yields the immediate successor of s
-- (resp. "failed" if s = [1,...,bound])
-- where s < t if and only if:
-- (i) #s < #t; or
-- (ii) #s = #t and s < t in the lexicographical order;
-- (we have chosen to fix the signature with NNI instead of PI
-- to avoid coercions in the main functions)

reducedQPowers(f) ==
  m:PI:=degree(f)$SUP pretend PI
  m1:I:=m-1
  setPoly(f)$MM
  e:=reduce(monomial(1,1)$SUP)$MM ** sizeGF
  w:=1$MM
  qpow:PrimitiveArray SUP:=new(m,0)
  qpow.0:=1$SUP
  for i in 1..m1 repeat qpow.i:=lift(w:=w*e)$MM
  qexp:PrimitiveArray SUP:=new(m,0)
m = 1 =>
qexp.(0$I):= (-coefficient(f,0$NNI)$SUP)::SUP
qexp
qexp.0$I:=monomial(1,1)$SUP
h:=qpow.1
qexp.1:=h
for i in 2..m1 repeat
g:=0$SUP
while h ^= 0 repeat
g:=g + leadingCoefficient(h) * qpow.degree(h)
h:=reductum(h)
qexp.i:=(h:=g)
qexp

leastAffineMultiple(f) ==
-- [LS] p.112
qexp:=reducedQPowers(f)
n:=degree(f)$SUP
b:Matrix GF:= transpose matrix [entries vectorise
(qexp.i,n) for i in 0..n-1]
coll:Matrix GF:= new(n,1,0)
coll(1,1) := 1
ns : List Vector GF := nullSpace (horizConcat(col1,b) )
----------------------------------------------------------------
-- perhaps one should use that the first vector in ns is already
-- the right one
----------------------------------------------------------------
dim:=n+2
coeffVector : Vector GF
until empty? ns repeat
newCoeffVector := ns.1
i : PI :=(n+1) pretend PI
while newCoeffVector(i) = 0 repeat
i := (i - 1) pretend PI
if i < dim then
dim := i
coeffVector := newCoeffVector
ns := rest ns
(coeffVector(1)::SUP) +(+/[monomial(coeffVector.k, _
sizeGF**((k-2)::NNI))$SUP for k in 2..dim]

-- qEulerPhiCyclotomic n ==
-- n = 1 => (sizeGF - 1) pretend PI
-- p : PI := characteristic()$GF :: PI
-- (n rem p) = 0 => error
-- "cyclotomic polynomial not defined for this argument value"
-- q : PI := sizeGF
-- -- determine the multiplicative order of q modulo n
-- e : PI := 1
-- qe : PI := q
-- while (qe rem n) ^= 1 repeat
-- e := e + 1
-- qe := qe * q
-- ((qe - 1) ** ((eulerPhi(n) quo e) pretend PI) ) pretend PI

numberOfIrreduciblePoly n ==
-- we compute the number Nq(n) of monic irreducible polynomials
-- of degree n over the field GF of order q by the formula
-- Nq(n) = (1/n)* sum(moebiusMu(n/d)*q**d) where the sum extends
-- over all divisors d of n (cf. [LN] p.93, Th. 3.25)
n = 1 => sizeGF
-- the contribution of d = 1 :
lastd : PI := 1
qd : PI := sizeGF
sum : I := moebiusMu(n) * qd
-- the divisors d > 1 of n :
divisorsOfn : L PI := rest(divisors n) pretend L PI
for d in divisorsOfn repeat
qd := qd * (sizeGF) ** ((d - lastd) pretend PI)
sum := sum + moebiusMu(n quo d) * qd
lastd := d
(sum quo n) :: PI

numberOfPrimitivePoly n == (eulerPhi((sizeGF ** n) - 1) quo n) :: PI
-- [each root of a primitive polynomial of degree n over a field
-- with q elements is a generator of the multiplicative group
-- of a field of order q**n (definition), and the number of such
-- generators is precisely eulerPhi(q**n - 1)]

numberOfNormalPoly n ==
-- we compute the number Nq(n) of normal polynomials of degree n
-- in GF[X], with GF of order q, by the formula
-- Nq(n) = (1/n) * qPhi(X**n - 1) (cf. [LN] p.124) where,
-- for any polynomial f in GF[X] of positive degree n,
-- qPhi(f) = q**n * (1 - q**(n1)) *...* (1 - q**(nr)) =
-- q**n * ((q**(n1)-1) / q**(n1)) *...* ((q**(nr)-1) / q**(n_r)),
-- the ni being the degrees of the distinct irreducible factors
-- of f in its canonical factorization over GF
-- hence, if n = m * p**r where p is the characteristic of GF
-- and gcd(m,p) = 1, we get
-- Nq(n) = (1/n)* q**(n-m) * qPhi(X**m - 1)
-- now X**m - 1 is the product of the (pairwise relatively prime)
-- cyclotomic polynomials Qd(X) for which d divides m
-- ([LN] p.64, Th. 2.45), and each Qd(X) factors into
-- eulerPhi(d)/e (distinct) monic irreducible polynomials in GF[X]
-- of the same degree e, where e is the least positive integer k
-- such that d divides q**k - 1 ([LN] p.65, Th. 2.47)
n = 1 => (sizeGF - 1) :: NN : PI
m : PI := n
p : PI := characteristic($GF :: PI
q : PI := sizeGF
while (m rem p) = 0 repeat -- find m such that
  m := (m quo p) :: PI -- n = m * p**r and gcd(m,p) = 1
m = 1 =>
  -- know that n is a power of p
  ((q ** ((n-1)::NNI) ) * (q - 1) ) quo n) :: PI
prod : I := q - 1
divisorsOfm : L PI := rest(divisors m) pretend L PI
for d in divisorsOfm repeat
  -- determine the multiplicative order of q modulo d
e : PI := 1
qe : PI := q
while (qe rem d) ^= 1 repeat
  e := e + 1
  qe := qe * q
  prod := prod * _
  ((qe - 1) ** ((eulerPhi(d) quo e) pretend PI) ) pretend PI
(q**((n-m) pretend PI) * prod quo n) pretend PI

primitive? f ==
  -- let GF be a field of order q; a monic polynomial f in GF[X]
  -- of degree n is primitive over GF if and only if its constant
  -- term is non-zero, f divides X**(q**n - 1) - 1 and,
  -- for each prime divisor d of q**n - 1,
  -- f does not divide X**((q**n - 1) / d) - 1
  -- (cf. [LM] p.89, Th. 3.16, and p.87, following Th. 3.11)
n : NNI := degree f
n = 0 => false
leadingCoefficient f ^= 1 => false
coefficient(f, 0) = 0 => false
q : PI := sizeGF
qn1: PI := (q**n - 1) :: NNI :: PI
setPoly f
x := reduce(monomial(1,1)$SUP)$MM -- X rem f represented in MM
-- may be improved by tabulating the residues x**(i*q)
-- for i = 0,...,n-1 :
--
lift(x ** qn1)$MM ^= 1 => false -- X**(q**n - 1) rem f in GF[X]
lrec : L Record(factor:I, exponent:I) := factors(factor qn1)
lfact : L PI := [] -- collect the prime factors
for rec in lrec repeat -- of q**n - 1
  lfact := cons((rec.factor) :: PI, lfact)
for d in lfact repeat
  if (expt := (qn1 quo d)) >= n then
    lift(x ** expt)$MM = 1 => return false
true

normal? f ==
-- let GF be a field with q elements; a monic irreducible
-- polynomial f in GF[X] of degree n is normal if its roots
-- x, x**q, ..., x**(q**(n-1)) are linearly independent over GF
n := degree f
n = 0 => false
leadingCoefficient f ^= 1 => false
coefficient(f, 0) = 0 => false
n = 1 => true
not irreducible? f => false
g := reducedQPowers(f)
l := [entries vectorise(g.i, n) for i in 0..(n-1)::NNI]
rank(matrix(l)$Matrix(GF)) = n => true
false

nextSubset(s, bound) ==
m := #(s)
m = 0 => [1]
-- find the first element s(i) of s such that s(i) + 1 < s(i+1):
noGap := true
i := 0
rest0Fs := L NNI
while noGap and not empty?(rest0Fs := rest s) repeat
  -- after i steps (0 <= i <= m-1) we have s = [s(i), ..., s(m)]
  -- and rest0Fs = [s(i+1), ..., s(m)]
  second0Fs := first rest0Fs  -- s(i+1)
  first0FsPlus1 := first s + 1  -- s(i) + 1
  second0Fs = first0FsPlus1 =>
    s := rest0Fs
    i := i + 1
    setfirst_!(s, first0FsPlus1)  -- s := [s(i)+1, s(i+1),..., s(m)]
  noGap := false
if noGap then  -- here s = [s(m)]
  first0Fs := first s
  first0Fs < bound => setfirst_!(s, first0Fs + 1)  -- s := [s(m)+1]
  m < bound =>
    setfirst_!(s, m + 1)  -- s := [m+1]
    i := m
  return "failed"  -- (here m = s(m) = bound)
for j in 1..m by -1 repeat  -- reconstruct the destroyed
  s := cons(j, s)  -- initial part of s
s

nextIrreduciblePoly f ==
n := degree f
n = 0 => error "polynomial must have positive degree"
-- make f monic
if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
-- if f = fn*X**n + ... + f(i0)*X**i0 with the fi non-zero
-- then fRepr := [[n,fn], ..., [i0,f{i0}]]
fRepr := Repr := f pretend Repr
fcopy : Repr := []
-- we can not simply write fcopy := copy fRepr because
-- the input(!) f would be modified by assigning
-- a new value to one of its records
for term in fRepr repeat
  fcopy := cons(copy term, fcopy)
if term.expnt ^= 0 then
  fcopy := cons([0,0]$Rec, fcopy)
tailpol : Repr := []
headpol : Repr := fcopy -- [[0,f0], ..., [n,fn]] where
  -- fi is non-zero for i > 0
fcopy := reverse fcopy
weight : NNI := (#(fcopy) - 1) :: NNI -- #s(f) as explained above
taillookuplist : L NNI := []
-- the zeroes in the headlookuplist stand for the fi
-- whose lookup's were not yet computed :
headlookuplist : L NNI := new(weight, 0)
s : L NNI := [] -- we will compute s(f) only if necessary
n1 : NNI := (n - 1) :: NNI
repeat
  -- (run through the possible weights)
  while not empty? headlookuplist repeat
    -- find next polynomial in the above order with fixed weight;
    -- assume at this point we have
    -- headpol = [[i1,f{i1}], [i2,f{i2}], ..., [n,1]]
    -- and tailpol = [[k,fk], ..., [0,f0]] (with k < i1)
    term := first headpol
    j := first headlookuplist
    if j = 0 then j := lookup(term.coeff)$GF
    j := j + 1 -- lookup(f{i1})$GF + 1
    j rem sizeGF = 0 =>
      -- in this case one has to increase f{i2}
      tailpol := cons(term, tailpol) -- [[i1,f{i1}], ..., [0,f0]]
      headpol := rest headpol -- [[i2,f{i2}], ..., [n,1]]
    taillookuplist := cons(j, taillookuplist)
    headlookuplist := rest headlookuplist
    -- otherwise set f{i1} := index(j)$GF
    setelt(first headpol, coeff, index(j :: PI)$GF)
    setfirst_!(headlookuplist, j)
  if empty? taillookuplist then
    pol := revListToSUP(headpol)
    --
    -- may be improved by excluding reciprocal polynomials
    --
    irreducible? pol => return pol
  else
    -- go back to fk
    headpol := cons(first tailpol, headpol) -- [[k,fk], ..., [n,1]]
    tailpol := rest tailpol
    headlookuplist := cons(first taillookuplist, headlookuplist)
nextPrimitivePoly f ==
  n : NNI := degree f
  n = 0 => error "polynomial must have positive degree"
  -- make f monic
  if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
  -- if f = fn*X**n + ... + fi*X**i0 with the fi non-zero
  -- then fRepr := [[n,fn], ... , [i0,fi0]]
  fRepr : Repr := f pretend Repr
  fcopy : Repr := []
  -- we can not simply write fcopy := copy fRepr because
  -- the input(!) f would be modified by assigning
  -- a new value to one of its records
  for term in fRepr repeat
    fcopy := cons(copy term, fcopy)
    if term.expnt ^= 0 then
      term := [0,0]$Rec
      fcopy := cons(term, fcopy)
  fcopy := reverse fcopy
  xn : Rec := first fcopy
  c0 : GF := term.coeff
  l : NNI := lookup(c0)$GF rem sizeGF
  n = 1 =>
    -- the polynomial X + c is primitive if and only if -c
    -- is a primitive element of GF
    q1 : NNI := (sizeGF - 1) :: NNI
    while l < q1 repeat -- find next c such that -c is primitive
      l := l + 1
      c := index(l :: PI)$GF
      primitive?(-c)$GF =>
        return [xn, [0,c]$Rec] pretend SUP
      "failed"
weight : NNI := (#(fcopy) - 1) :: NNI -- #s(f)+1 as explained above
s : L NNI := [] -- we will compute s(f) only if necessary
n1 : NNI := (n - 1) :: NNI
-- a necessary condition for a monic polynomial f of degree n
-- over GF to be primitive is that (-1)^n * f(0) be a
-- primitive element of GF (cf. [LN] p.90, Th. 3.18)
c : GF := c0
while l < sizeGF repeat
  -- (run through the possible values of the constant term)
  noGenerator : Boolean := true
  while noGenerator and l < sizeGF repeat
    -- find least c >= c0 such that (-1)^n c0 is primitive
    primitive?((-1)**n * c)$GF => noGenerator := false
    l := l + 1
    c := index(l :: PI)$GF
  noGenerator => return "failed"
  constterm : Rec := [0, c]$Rec
  if c = c0 and weight > 1 then
    headpol : Repr := rest reverse fcopy -- [[i0,f{i0}],...,[n,1]]
    -- fi is non-zero for i>0
    -- the zeroes in the headlookuplist stand for the fi
    -- whose lookup's were not yet computed:
    headlookuplist : L NNI := new(weight, 0)
  else
    -- X**n + c can not be primitive for n > 1 (cf. [LN] p.90,
    -- Th. 3.18); next possible polynomial is X**n + X + c
    headpol : Repr := [[1,0]$Rec, xn] -- 0*X + X**n
    headlookuplist : L NNI := [sizeGF]
    s := [0,1]
    weight := 2
    tailpol : Repr := []
    taillookuplist : L NNI := []
  notReady : Boolean := true
  while notReady repeat
    -- (run through the possible weights)
    headlookuplist repeat
      -- find next polynomial in the above order with fixed
      -- constant term and weight; assume at this point we have
      -- headpol = [[i1,f{i1}], [i2,f{i2}], ... , [n,1]] and
      -- tailpol = [[k,fk],...,[k0,fk0]] (k0<...<k<i1<i2<...<n)
      term := first headpol
      j := first headlookuplist
      if j = 0 then j := lookup(term.coeff)$GF
      j := j + 1 -- lookup(f{i1})$GF + 1
      j rem sizeGF = 0 =>
        -- in this case one has to increase f{i2}
      tailpol := cons(term, tailpol) -- [[i1,f{i1}],...,[k0,f{k0}]]
      headpol := rest headpol
        -- [[i2,f{i2}],...,[n,1]]
      taillookuplist := cons(j, taillookuplist)
  headlookuplist := rest headlookuplist
CHAPTER 7. CHAPTER F

-- otherwise set f{i1} := index(j)$GF
setelt(first headpol, coeff, index(j :: PI)$GF)
setfirst!(headlookuplist, j)
if empty? taillookuplist then
  pol := revListToSUP cons(constterm, headpol)
  --
  -- may be improved by excluding reciprocal polynomials
  --
  primitive? pol => return pol
else
  -- go back to fk
  headpol := cons(first tailpol, headpol) -- [[k,fk],...,[n,1]]
tailpol := rest tailpol
  headlookuplist := cons(first taillookuplist,
                        headlookuplist)
taillookuplist := rest taillookuplist
if weight = n then notReady := false
else
  -- must search for polynomial with greater weight
  if empty? s then -- compute s(f)
    restfcopy := rest fcopy
    for entry in restfcopy repeat s := cons(entry.expnt, s)
s1 := nextSubset(rest s, n1) :: L NNI
  s := cons(0, s1)
  weight := #s
  taillookuplist := []
  headlookuplist := cons(sizeGF, new((weight-2) :: NNI, 1))
tailpol := []
  -- headpol = [[s.2,0], [s.3,1], ..., [s.weight,1], [n,1]] :
  headpol := [[first s1, 0]$Rec]
  while not empty? (s1 := rest s1) repeat
    headpol := cons([first s1, 1]$Rec, headpol)
  headpol := reverse cons([n, 1]$Rec, headpol)
  -- next polynomial must have greater constant term
  l := l + 1
  c := index(l :: PI)$GF
  "failed"

nextNormalPoly f ==
n : NNI := degree f
n = 0 => error "polynomial must have positive degree"
-- make f monic
if (lcf := leadingCoefficient f) ^= 1 then f := (inv lcf) * f
-- if f = fn*X**n + ... + f{i0}*X**{i0} with the fi non-zero
-- then fRepr := [[n,fn], ..., [i0,f{i0}]]
fRepr : Repr := f pretend Repr
fcopy : Repr := []
-- we can not simply write fcopy := copy fRepr because
-- the input(!) f would be modified by assigning
-- a new value to one of its records
for term in fRepr repeat
  fcopy := cons(copy term, fcopy)
if term.expnt ^= 0 then
  term := [0,0]$Rec
  fcopy := cons(term, fcopy)
fcopy := reverse fcopy
-- [n,1], [r,fr], ... , [0,f0]
xn : Rec := first fcopy
middlepol : Repr := rest fcopy
-- [r,fr], ... , [0,f0]
a0 : GF := (first middlepol).coeff -- fr
l : NNI := lookup(a0)$GF rem sizeGF
n = 1 =>
  -- the polynomial X + a is normal if and only if a is not zero
  l = sizeGF - 1 => "failed"
  [xn, [0, index((1+1) :: PI)$GF]$Rec] pretend SUP
n1 : NNI := (n - 1) :: NNI
n2 : NNI := (n1 - 1) :: NNI
  -- if the polynomial X**n + a * X**(n-1) + ... is normal then
  -- a = -(x + x**q +...+ x**(q**n)) can not be zero (where q = #GF)
a : GF := a0
  -- if a = 0 then set a := 1
if l = 0 then
  l := 1
  a := 1$GF
while l < sizeGF repeat
  -- (run through the possible values of a)
if a = a0 then
  -- middlepol = [[0,f0], ... , [m,fm]] with m < n-1
  middlepol := reverse rest middlepol
  weight : NNI := #middlepol -- #s(f) as explained above
  -- the zeroes in the middlelookuplist stand for the fi
  -- whose lookup's were not yet computed :
  middlelookuplist : L NNI := new(weight, 0)
  s : L NNI := [] -- we will compute s(f) only if necessary
else
  middlepol := [[0,0]$Rec]
  middlelookuplist : L NNI := [sizeGF]
  s : L NNI := [0]
  weight : NNI := 1
headpol : Repr := [xn, [n1, a]$Rec] -- X**n + a * X**(n-1)
tailpol : Repr := []
taillookuplist : L NNI := []
notReady : Boolean := true
while notReady repeat
  -- (run through the possible weights)
while not empty? middlelookuplist repeat
  -- find next polynomial in the above order with fixed
  -- a and weight; assume at this point we have
  -- middlepol = [[i1,f{i1}], [i2,f{i2}], ... , [m,fm]] and
  -- tailpol = [[k,fk],...,[0,f0]] ( with k<i1<i2<...<m)
term := first middlepol
j := first middlelookuplist
if j = 0 then j := lookup(term.coeff)$GF
j := j + 1 -- lookup(f{i1})$GF + 1
j rem sizeGF = 0 =>
  -- in this case one has to increase f{i2}
  -- tailpol = [[i1,f(i1)],...,[0,f0]]
tailpol := cons(term, tailpol)
middlepol := rest middlepol -- [[i2,f(i2)],...,[m, fm]]
taillookuplist := cons(j, taillookuplist)
middlelookuplist := rest middlelookuplist
-- otherwise set f{i1} := index(j)$GF
setelt(first middlepol, coeff, index(j :: PI)$GF)
setfirst_!(middlelookuplist, j)
if empty? taillookuplist then
  pol := listToSUP append(headpol, reverse middlepol)
  --
  -- may be improved by excluding reciprocal polynomials
  --
  normal? pol => return pol
else
  -- go back to fk
  -- middlepol = [[k,fk],...,[m,fm]]
middlepol := cons(first tailpol, middlepol)
tailpol := rest tailpol
middlelookuplist := cons(first taillookuplist, middlelookuplist)
taillookuplist := rest taillookuplist
if weight = n1 then notReady := false
else
  -- must search for polynomial with greater weight
  if empty? s then -- compute s(f)
    restfcopy := rest rest fcopy
    for entry in restfcopy repeat s := cons(entry.expnt, s)
  s1 := nextSubset(rest s, n2) :: L NNI
  s := cons(0, s1)
  weight := #s
taillookuplist := []
middlelookuplist := cons(sizeGF, new((weight-1) :: NNI, 1))
tailpol := []
  -- middlepol = [[0,0], [s.2,1], ... , [s.weight,1]] :
middlepol := []
s1 := reverse s1
while not empty? s1 repeat
  middlepol := cons([first s1, 1]$Rec, middlepol)
s1 := rest s1
middlepol := cons([0,0]$Rec, middlepol)
  -- next polynomial must have greater a
  l := l + 1
  a := index(l :: PI)$GF
"failed"
nextNormalPrimitivePoly f ==
  n := degree f
  n = 0 => error "polynomial must have positive degree"
  -- make f monic
  if lcf := leadingCoefficient f ^= 1 then f := (inv lcf) * f
  -- if f = f_n*X**n + ... + f{i0}*X**{i0} with the fi non-zero
  -- then fRepr := [[n,fn], ..., [i0,f{i0}]]
  fRepr : Repr := f pretend Repr
  fcopy : Repr := []
  -- we can not simply write fcopy := copy fRepr because
  -- the input(!) f would be modified by assigning
  -- a new value to one of its records
  for term in fRepr repeat
    fcopy := cons(copy term, fcopy)
    if term.expnt ^= 0 then
      term := [0,0]$
      fcopy := cons(term, fcopy)
  fcopy := reverse fcopy -- [[n,1], [r,fr], ... , [0,f0]]
  xn : Rec := first fcopy
  c0 : GF := term.coeff
  lc : NNI := lookup(c0)$GF rem sizeGF
  n = 1 =>
    -- the polynomial X + c is primitive if and only if -c
    -- is a primitive element of GF
    q1 : NNI := (sizeGF - 1) :: NNI
    while lc < q1 repeat -- find next c such that -c is primitive
      lc := lc + 1
      c := index(lc :: PI)$GF
      primitive?(-c)$GF =>
        return [xn, [0,c]$Rec] pretend SUP
    "failed"
  n1 : NNI := (n - 1) :: NNI
  n2 : NNI := (n1 - 1) :: NNI
  middlepol : Repr := rest fcopy -- [[r,fr],...,[i0,f{i0}],[0,f0]]
  a0 : GF := (first middlepol).coeff
  la : NNI := lookup(a0)$GF rem sizeGF
  -- if the polynomial X**n + a * X**(n-1) +...+ c is primitive and
  -- normal over GF then (-1)**n * c is a primitive element of GF
  -- (cf. [LN] p.90, Th. 3.18), and a = -(x + x**q +...+ x**(q**n))
  -- is not zero (where q = #GF)
  c : GF := c0
  a : GF := a0
  -- if a = 0 then set a := 1
  if la = 0 then
    la := 1
    a := 1$GF
  while lc < sizeGF repeat
    -- (run through the possible values of the constant term)
    noGenerator : Boolean := true
while noGenerator and lc < sizeGF repeat
  -- find least \( c \geq c0 \) such that \((-1)^n \cdot c \) is primitive
  primitive?((-1)**n * c)$GF => noGenerator := false
  lc := lc + 1
  c := index(lc :: PI)$GF
noGenerator => return "failed"
constterm : Rec := [0, c]$Rec
while la < sizeGF repeat
  -- (run through the possible values of \( a \))
  headpol : Repr := [xn, [n1, a]$Rec] -- \( X^n + aX^{n-1} \)
  if c = c0 and a = a0 then
    -- middlepol = \([i0,f{i0}], \ldots, [m,fm]\) with \( m < n-1 \)
    middlepol := rest reverse rest middlepol
    weight : NNI := #middlepol + 1 -- \#s(f)+1 as explained above
  -- the zeroes in the middlelookuplist stand for the fi
  -- whose lookup's were not yet computed :
    middlelookuplist : L NNI := new((weight-1) :: NNI, 0)
  s : L NNI := [] -- we will compute s(f) only if necessary
  else
    pol := listToSUP append(headpol, [constterm])
    normal? pol and primitive? pol => return pol
    middlepol := [[1,0]$Rec]
    middlelookuplist : L NNI := [sizeGF]
    s : L NNI := [0,1]
    weight : NNI := 2
    tailpol : Repr := []
    taillookuplist : L NNI := []
    notReady : Boolean := true
  while notReady repeat
    -- (run through the possible weights)
    while not empty? middlelookuplist repeat
      -- find next polynomial in the above order with fixed
      -- \( c, a \) and weight; assume at this point we have
      -- middlepol = \([i0,f{i0}], [i2,f{i2}], \ldots, [m,fm]\)
      -- tailpol = \([k,fk], [k0,fk0]\) (k0<...<k<i1<...<m)
      term := first middlepol
      j := first middlelookuplist
      if j = 0 then j := lookup(term.coeff)$GF
      j := j + 1 -- lookup(f{i1})$GF + 1
      j rem sizeGF = 0 =>
        -- in this case one has to increase \( f{i2} \)
        -- tailpol = [[i1,f{i1}],[k0,f{k0}]]
        tailpol := cons(term, tailpol)
      middlepol := rest middlepol -- \([i2,f{i2}], \ldots, [m,fm]\)
      taillookuplist := cons(j, taillookuplist)
    middlelookuplist := rest middlelookuplist
    -- otherwise set \( f{i1} := index(j :: PI)$GF \)
    setelt(first middlepol, coeff, index(j :: PI)$GF)
    setfirst_!(middlelookuplist, j)
  if empty? taillookuplist then
pol := listToSUP append(headpol, reverse
    cons(constterm, middlepol))
--
-- may be improved by excluding reciprocal polynomials
--
normal? pol and primitive? pol => return pol
else
-- go back to fk
-- middlepol = [[k,fk],...,[m,fm]]
middlepol := cons(first tailpol, middlepol)
tailpol := rest tailpol
middlelookuplist := cons(first taillookuplist, middlelookuplist)
taillookuplist := rest taillookuplist
if weight = n1 then notReady := false
else
-- must search for polynomial with greater weight
if empty? s then -- compute s(f)
    restfcopy := rest rest fcopy
    for entry in restfcopy repeat s := cons(entry.expnt, s)
    s1 := nextSubset(rest s, n2) :: L NNI
    s := cons(0, s1)
    weight := #s
    taillookuplist := []
middlelookuplist := cons(sizeGF, new((weight-2)::NNI, 1))
tailpol := []
-- middlepol = [[s.2,0], [s.3,1], ... , [s.weight,1] :
middlepol := [[first s1, 0]$Rec]
while not empty? (s1 := rest s1) repeat
    middlepol := cons([first s1, 1]$Rec, middlepol)
-- next polynomial must have greater a
la := la + 1
a := index(la :: PI)$GF
-- next polynomial must have greater constant term
lc := lc + 1
c := index(lc :: PI)$GF
la := 1
a := 1$GF
"failed"

nextPrimitiveNormalPoly f == nextNormalPrimitivePoly f

createIrreduciblePoly n ==
    x := monomial(1,1)$SUP
    n = 1 => x
    xn := monomial(1,n)$SUP
    n >= sizeGF => nextIrreduciblePoly(xn + x) :: SUP
    -- (since in this case there is most no irreducible binomial X+a)
    odd? n => nextIrreduciblePoly(xn + 1) :: SUP
nextIrreduciblePoly(xn) :: SUP

createPrimitivePoly n ==
  -- (see also the comments in the code of nextPrimitivePoly)
  xn := monomial(1,n)$SUP
  n = 1 => xn + monomial(-primitiveElement()$GF, 0)$SUP
  c0 : GF := (-1)**n * primitiveElement()$GF
  constterm : Rec := [0, c0]$Rec
  -- try first (probably faster) the polynomials
  -- f = X**n + f{n-1}*X**(n-1) +...+ f1*X + c0 for which
  -- fi is 0 or 1 for i=1,...,n-1,
  -- and this in the order used to define nextPrimitivePoly
  s : L NNI := [0,1]
  weight : NNI := 2
  s1 : L NNI := [1]
  n1 : NNI := (n - 1) :: NNI
  notReady : Boolean := true
  while notReady repeat
    polRepr : Repr := [constterm]
    while not empty? s1 repeat
      polRepr := cons([first s1, 1]$Rec, polRepr)
      s1 := rest s1
    polRepr := cons([n, 1]$Rec, polRepr)
    --
    -- may be improved by excluding reciprocal polynomials
    --
    primitive? (pol := listToSUP polRepr) => return pol
    if weight = n then notReady := false
    else
      s1 := nextSubset(rest s, n1) :: L NNI
      s := cons(0, s1)
      weight := #s
    -- if there is no primitive f of the above form
    -- search now from the beginning, allowing arbitrary
    -- coefficients f_i, i = 1,...,n-1
  nextPrimitivePoly(xn + monomial(c0, 0)$SUP) :: SUP

createNormalPoly n ==
  n = 1 => monomial(1,1)$SUP + monomial(-1,0)$SUP
  -- get a normal polynomial f = X**n + a * X**(n-1) + ... 
  -- with a = -1
  -- [recall that if f is normal over the field GF of order q
  -- then a = -(x + x**q +...+ x**(q**n)) can not be zero;
  -- hence the existence of such an f follows from the
  -- normal basis theorem ([LN] p.60, Th. 2.35) and the
  -- surjectivity of the trace ([LN] p.55, Th. 2.23 (iii))]
  nextNormalPoly(monomial(1,n)$SUP
  + monomial(-1, (n-1) :: NNI)$SUP) :: SUP

createNormalPrimitivePoly n ==
xn := monomial(1, n)$SUP
n1 := NNI := (n - 1) :: NNI

c0 := GF := (-1)**n * primitiveElement()$GF
constterm := monomial(c0, 0)$SUP

-- try first the polynomials f = X**n + a * X**(n-1) + ...
-- with a = -1
pol := xn + monomial(-1, n1)$SUP + constterm

normal? pol and primitive? pol => pol

res := nextNormalPrimitivePoly(pol)
res case SUP => res

-- if there is no normal primitive f with a = -1
-- get now one with arbitrary (non-zero) a
-- (the existence is proved in [LS])

pol := xn + monomial(1, n1)$SUP + constterm

normal? pol and primitive? pol => pol

nextNormalPrimitivePoly(pol) :: SUP

createPrimitiveNormalPoly n == createNormalPrimitivePoly n

-- qAdicExpansion m ==
-- ragits : List I := wholeRagits(m :: (RadixExpansion sizeGF))
-- pol := 0
-- expt := #ragits
-- for i in ragits repeat
-- expt := (expt - 1) :: NNI
-- if i ^= 0 then pol := pol + monomial(index(i::PI)$GF, expt)
-- pol

-- random == qAdicExpansion(random()$I)

-- random n ==
-- pol := monomial(1, n)$SUP
-- n1 := NNI := (n - 1) :: NNI
-- for i in 0..n1 repeat
-- if (c := random()$GF) ^= 0 then
-- pol := pol + monomial(c, i)$SUP
-- pol

random n ==

n1 := NNI := (n - 1) :: NNI
for i in 0..n1 repeat
if (c := random()$GF) ^= 0 then
polRepr := cons([i, c]$Rec, polRepr)
cons([n, 1$GF]$Rec, polRepr) pretend SUP

random(m, n) ==
if m > n then (m, n) := (n, m)

d := NNI := (n - m) :: NNI
if d > 1 then n := ((random()$I rem (d::PI)) + m) :: PI
random(n)

---

-- FFPOLY.dotabb --

"FFPOLY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFPOLY"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFPOLY" -> "PFECAT"

---

package FFPOLY2 FiniteFieldPolynomialPackage2

--- FiniteFieldPolynomialPackage2.input ---

)set break resume
)sys rm -f FiniteFieldPolynomialPackage2.output
)spool FiniteFieldPolynomialPackage2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldPolynomialPackage2
  --R
  --R FiniteFieldPolynomialPackage2(F,GF: FiniteFieldCategory)where
  --R  F: FieldOfPrimeCharacteristicwith
  --R    coerce : GF -> F
  --R    lookup : F -> PositiveInteger
  --R    basis : PositiveInteger -> Vector(F)
  --R    Frobenius : F -> F is a package constructor
  --R Abbreviation for FiniteFieldPolynomialPackage2 is FFPOLY2
  --R This constructor is exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFPOLY2
  --R
  --R----------------------------------- Operations -----------------------------------
  --R rootOfIrreduciblePoly : SparseUnivariatePolynomial(GF) -> F
  --R
  --E 1

)spool
)lisp (bye)
FiniteFieldPolynomialPackage2 (FFPOLY2)

Exports:
rootOfIrreduciblePoly

--- package FFPOLY2 FiniteFieldPolynomialPackage2 ---

)abbrev package FFPOLY2 FiniteFieldPolynomialPackage2
++ Authors: J.Grabmeier, A.Scheerhorn
++ Date Created: 26.03.1991
++ References:
++ R.Lidl, H.Niederreiter: Finite Field, Encycloedia of Mathematics and
++ Description:
++ FiniteFieldPolynomialPackage2(F,GF) exports some functions concerning
++ finite fields, which depend on a finite field GF and an
++ algebraic extension F of GF, e.g. a zero of a polynomial
++ over GF in F.

FiniteFieldPolynomialPackage2(F,GF):Exports == Implementation where
  F:FieldOfPrimeCharacteristic with
  coerc: GF -> F
  ++ coerc(x) undocumented{}
  lookup: F -> PositiveInteger
  ++ lookup(x) undocumented{}
  basis: PositiveInteger -> Vector F
  ++ basis(n) undocumented{}
  Frobenius: F -> F
  ++ Frobenius(x) undocumented{}
  -- F should be a algebraic extension of the finite field GF, either an
  -- algebraic closure of GF or a simple algebraic extension field of GF
  GF:FiniteFieldCategory

I    ==> Integer
NNI  ==> NonNegativeInteger
PI   ==> PositiveInteger
SUP  ==> SparseUnivariatePolynomial
MM   ==> ModMonic(GF,SUP GF)
OUT  ==> OutputForm
M    ==> Matrix
V    ==> Vector
L    ==> List
FFPOLY ==> FiniteFieldPolynomialPackage(GF)
SUPF2 ==> SparseUnivariatePolynomialFunctions2(GF,F)

Exports == with

  rootOfIrreduciblePoly:SUP GF -> F
  ++ rootOfIrreduciblePoly(f) computes one root of the monic,
  ++ irreducible polynomial f,
  ++ which degree must divide the extension degree
  ++ of F over GF,
  ++ i.e. f splits into linear factors over F.

Implementation ==> add

-- we use berlekamps trace algorithm
-- it is not checked whether the polynomial is irreducible over GF]
rootOfIrreduciblePoly(pf) ==
  -- not irreducible(pf)$FFPOLY =>
  -- error("polynomial has to be irreducible")
  sizeGF:=size()$GF
  -- if the polynomial is of degree one, we're ready
deg:=degree(pf)$(SUP GF)::PI
deg = 0 => error("no roots")
deg = 1 => -coefficient(pf,0)$(SUP GF)::F
p : SUP F := map(coerce,pf)$SUPF2
-- compute qexp, qexp(i) = x **(size()GF) ** i mod p
-- with this list it's easier to compute the gcd(p(x),trace(x))
qexp:=reducedQPowers(pf)$FFPOLY
stillToFactor:=p
-- take linear independent elements, the basis of F over GF
basis:Vector F:=basis(deg)$F
basispointer:1:=1
-- as p is irreducible over GF, 0 can't be a root of p
-- therefore we can use the predicate zero?(root) for indicating
-- whether a root is found
root:=0$F
while zero?(root)$F repeat
  beta:F:=basis.basispointer
  -- gcd(trace(x)+gf,p(x)) has degree 0, that's why we skip beta=1
  if beta = 1$F then
    basispointer:=basispointer + 1
    beta:= basis.basispointer
    basispointer:=basispointer+1
  -- compute the polynomial trace(beta * x) mod p(x) using explist
  trModp:SUP F:= map(coerce,qexp.0)$SUPF2 * beta
  for i in 1..deg-1 repeat
    beta:=Frobenius(beta)
    trModp:=trModp +$(SUP F) beta *$(SUP F) map(coerce,qexp.i)$SUPF2
  -- if it is of degree 0, it doesn't help us finding a root
  if degree(trModp)$(SUP F) > 0 then
    -- for all elements gf of GF do
    for j in 1..sizeGF repeat
      -- compute gcd(trace(beta * x) + gf,stillToFactor)
      h:=gcd(stillToFactor,trModp +$(SUP F) _
               (index(j pretend PI)$GF::F::(SUP F))$(SUP F))$(SUP F)
      -- make the gcd polynomial monic
      if leadingCoefficient(h)$(SUP F) ^= 1$F then
        h := (inv leadingCoefficient(h)) * h
      degh:=degree(h)$(SUP F)
degSTF:=degree(stillToFactor)$(SUP F)
      -- if the gcd has degree one we are ready
      degh = 1 => root:=-coefficient(h,0)$(SUP F)
      -- if the quotient of stillToFactor and the gcd has
      -- degree one, we're also ready
      degSTF - degh = 1 =>
        root:=-coefficient(stillToFactor quo h,0)$(SUP F)
      -- otherwise the gcd helps us finding a root, only if its
      -- degree is between 2 and degree(stillToFactor)-2
      if degh > 1 and degh < degSTF then
        2*degh > degSTF => stillToFactor := stillToFactor quo h
      stillToFactor := h
    root
package FFSLPE FiniteFieldSolveLinearPolynomialEquation

 FiniteFieldSolveLinearPolynomialEquation.input —

)set break resume
)sys rm -f FiniteFieldSolveLinearPolynomialEquation.output
)spool FiniteFieldSolveLinearPolynomialEquation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteFieldSolveLinearPolynomialEquation
--R
--R FiniteFieldSolveLinearPolynomialEquation(F: FiniteFieldCategory,FP: UnivariatePolynomialCategory(FPP)) is a package constructor
--R Abbreviation for FiniteFieldSolveLinearPolynomialEquation is FFSLPE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFSLPE
--R
--R---------------------------------- Operations ----------------------------------
--R solveLinearPolynomialEquation : (List(FPP),FPP) -> Union(List(FPP),"failed")
--R
--E 1

)spool
)lisp (bye)

 FiniteFieldSolveLinearPolynomialEquation.help —
This package solves linear diophantine equations for Bivariate polynomials over finite fields

See Also:
• )show FiniteFieldSolveLinearPolynomialEquation

FiniteFieldSolveLinearPolynomialEquation (FFSLPE)

Exports:
solveLinearPolynomialEquation

— package FFSLPE FiniteFieldSolveLinearPolynomialEquation —

)abbrev package FFSLPE FiniteFieldSolveLinearPolynomialEquation
++ Author: Davenport
++ Date Created: 1991
++ Description:
++ This package solves linear diophantine equations for Bivariate polynomials
++ over finite fields

FiniteFieldSolveLinearPolynomialEquation(F:FiniteFieldCategory,
    FP:UnivariatePolynomialCategory F,
    FPP:UnivariatePolynomialCategory FP): with
    solveLinearPolynomialEquation: (List FPP, FPP) -> Union(List FPP,"failed")
    ++ solveLinearPolynomialEquation([f1, ..., fn], g)
    ++ (where the fi are relatively prime to each other)
    ++ returns a list of ai such that
++ \spad{g/prod fi = sum ai/fi}
++ or returns "failed" if no such list of ai's exists.

==
oldlp:List FPP := []
slpePrime: FP := monomial(1,1)
oldtable:Vector List FPP := []
lp: List FPP
p: FPP
import DistinctDegreeFactorize(F,FP)
solveLinearPolynomialEquation(lp,p) ==
  if (oldlp ^= lp) then
    -- we have to generate a new table
    deg:= +/[degree u for u in lp]
    ans:Union(Vector List FPP,"failed"):"failed"
    slpePrime:=monomial(1,1)+monomial(1,0) -- x+1: our starting guess
    while (ans case "failed") repeat
      ans:=tablePow(deg,slpePrime,lp)$GenExEuclid(FP,FPP)
      if (ans case "failed") then
        slpePrime:= nextItem(slpePrime)::FP
      while (degree slpePrime > 1) and
        not irreducible? slpePrime repeat
        slpePrime := nextItem(slpePrime)::FP
      oldtable:=(ans:: Vector List FPP)
      answer:=solveid(p,slpePrime,oldtable)
      answer

-----

— FFSLPE.dotabb —

"FFSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSLPE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFSLPE" -> "PFECAT"

-----

package FFSQFR FiniteFieldSquareFreeDecomposition

— FiniteFieldSquareFreeDecomposition.input —

)set break resume
)sys rm -f FiniteFieldSquareFreeDecomposition.output
)spool FiniteFieldSquareFreeDecomposition.output
)set message test on


---

FiniteFieldSquareFreeDecomposition.help

====================================================================
FiniteFieldSquareFreeDecomposition examples
====================================================================

Part of the package for Algebraic Function Fields in one variable (PAFF)

See also:

  o )show FiniteFieldSquareFreeDecomposition
FiniteFieldSquareFreeDecomposition (FFSQFR)

Exports:
PolK Yun

--- package FFSQFR FiniteFieldSquareFreeDecomposition ---

)abbrev package FFSQFR FiniteFieldSquareFreeDecomposition
++ Author: Patrice Naudin and Claude Quitte
++ Date Created: September 1996
++ Date Last Updated: April 2010 by Tim Daly
++ Description:
++ Part of the package for Algebraic Function Fields in one variable (PAFF)
FiniteFieldSquareFreeDecomposition (K : FiniteFieldCategory,
   PolK : UnivariatePolynomialCategory (K)) : with
       Musser : PolK -> Factored (PolK)
       Yun : PolK -> Factored (PolK)
== add

p : NonNegativeInteger := characteristic()$K
tableOfSquareFreePolynomials := Table (Integer, PolK)
oneYunStep2uple := Record (
    simpleDecomposition : tableOfSquareFreePolynomials,
    gcdOfArgumentAndDerivative : PolK
  )

rawMusser (P : PolK) : Factored(PolK) ==
Q := gcd(P, D(P))
A := P quo Q
decomposition : Factored(PolK) := 1
B := PolK

for i in 1 .. repeat
  if i rem p ^= 0 then
    B := gcd(A, Q)
    decomposition := sqfrFactor(A quo B, i) * decomposition
  if B = 1 then leave
A := B
Q := Q quo A
if Q ^= 1 then
    decomposition := decomposition * rawMusser (charthRoot(Q)::PolK) ** p
return decomposition

Musser (P : PolK) : Factored(PolK) ==
degree (P) = 0 => return P::Factored(PolK)
if (lcP : K := leadingCoefficient(P)) ^= 1 then P := inv(lcP) * P
return lcP::PolK * rawMusser (P)

oneYunStep (P : PolK) : oneYunStep2uple ==
C : PolK := D (P) ; A : PolK := gcd(P, C)
gcd_P_P' : PolK := A ; B : PolK := P quo A
result : tableOfSquareFreePolynomials := empty ()
i : Integer := 1
repeat
    C := (C quo A) - D(B)
    if C = 0 then leave
    A := gcd(B, C)
    if A ^= 1 then
        result (i) := A
        B := B quo A
        i := i + 1
    result (i) := B
return [result, gcd_P_P']

rawYun (P : PolK) : tableOfSquareFreePolynomials ==
u : oneYunStep2uple := oneYunStep (P)
gcd_P_P' : PolK := u.gcdOfArgumentAndDerivative
U : tableOfSquareFreePolynomials := u.simpleDecomposition
R : PolK := gcd_P_P'
for j in indices (U) repeat
    for k in 1 .. j-1 repeat
        R := R quo U(j)
    if R = 1 then return U
V : tableOfSquareFreePolynomials := rawYun (charthRoot (R)::PolK)
result : tableOfSquareFreePolynomials := empty ()
gcd_Uj_Vk : PolK ;
for k in indices (V) repeat -- boucle 1
    for j in indices (U) | not (U(j) = 1) repeat -- boucle 2
        gcd_Uj_Vk := gcd (U(j), V(k))
        if not (gcd_Uj_Vk = 1) then
            result (j+p*k) := gcd_Uj_Vk
            V (k) := V(k) quo gcd_Uj_Vk
            U (j) := U(j) quo gcd_Uj_Vk
if \( V(k) = 1 \) then leave

if not \( (V(k) = 1) \) then
  result \((p\cdot k)\) := \( V(k) \)

for \( j \) in indices \( (U) \mid \text{not } (U(j) = 1) \) repeat -- boucle 3
  result \((j)\) := \( U(j) \)

return result

\[
\text{Yun}(P : \text{PolK}) : \text{Factored}(\text{PolK}) \Rightarrow
\begin{align*}
\text{degree}(P) = 0 & \Rightarrow P : \text{Factored}(\text{PolK}) \\
\text{if} \ (1cP := \text{leadingCoefficient}(P)) \neq 1 \ &\Rightarrow P := \text{inv}(1cP) \cdot P \\
U : \text{tableOfSquareFreePolynomials} := \text{rawYun}(P) \\
PFactored : \text{Factored}(\text{PolK}) := 1 \\
\text{for } i \ \text{in } \text{indices } (U) \ \text{repeat} \\
PFactored := PFactored \cdot \text{sqfrFactor}(U(i), i) \\
\text{return } (1cP : \text{PolK}) \cdot PFactored
\end{align*}
\]

——

— FFSQFR.dotabb —

"FFSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFSQFR" -> "PFECAT"

——

package FLAGG2 FiniteLinearAggregateFunctions2

— FiniteLinearAggregateFunctions2.input —

)set break resume
)sys rm -f FiniteLinearAggregateFunctions2.output
)spool FiniteLinearAggregateFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteLinearAggregateFunctions2
--R
--R FiniteLinearAggregateFunctions2(S: Type,A: FiniteLinearAggregate(S),R: Type,B: FiniteLinearAggregate(B)) is a package constructor
---R Abbreviation for FiniteLinearAggregateFunctions2 is FLAGG2
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for FLAGG2
---R
---R---------------------------------- Operations -----------------------------
---R map : ((S -> R),A) -> B  reduce : (((S,R) -> R),A,R) -> R
---R scan : (((S,R) -> R),A,R) -> B
---R
---E 1

)spool
)lisp (bye)

---

— FiniteLinearAggregateFunctions2.help —

====================================================================
FiniteLinearAggregateFunctions2 examples
====================================================================

FiniteLinearAggregateFunctions2 provides functions involving two
FiniteLinearAggregates where the underlying domains might be
different. An example of this might be creating a list of rational
numbers by mapping a function across a list of integers where the
function divides each integer by 1000.

See Also:
o )show FiniteLinearAggregateFunctions2

---

FiniteLinearAggregateFunctions2 (FLAGG2)
Exports:
map reduce scan

— package FLAGG2 FiniteLinearAggregateFunctions2 —

)abbrev package FLAGG2 FiniteLinearAggregateFunctions2
++ Description:
++ FiniteLinearAggregateFunctions2 provides functions involving two
++ FiniteLinearAggregates where the underlying domains might be
++ different. An example of this might be creating a list of rational
++ numbers by mapping a function across a list of integers where the
++ function divides each integer by 1000.

FiniteLinearAggregateFunctions2(S, A, R, B):
Exports == Implementation where
 S, R: Type
A : FiniteLinearAggregate S
B : FiniteLinearAggregate R

Exports ==> with
map : (S -> R, A) -> B
++ map(f,a) applies function f to each member of aggregate
++ \spad{a} resulting in a new aggregate over a
++ possibly different underlying domain.
reduce : ((S, R) -> R, A, R) -> R
++ reduce(f,a,r) applies function f to each
++ successive element of the
++ aggregate \spad{a} and an accumulant initialized to r.
++ For example,
++ \spad{reduce(_+$Integer,[1,2,3],0)}
++ does \spad{3+(2+(1+0))}. Note that third argument r
++ may be regarded as the
++ identity element for the function f.
scan : ((S, R) -> R, A, R) -> B
++ scan(f,a,r) successively applies
++ \spad{reduce(f,x,r)} to more and more leading sub-aggregates
++ x of aggregate \spad{a}.
++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
++ \spad{scan(f,a,r)} returns
++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
Implementation ==> add
if A has ListAggregate(S) then -- A is a list-oid
  reduce(fn, l, ident) ==
    empty? l => ident
    reduce(fn, rest l, fn(first l, ident))
if B has ListAggregate(R) or not(B has shallowlyMutable) then
  -- A is a list-oid, and B is either list-oids or not mutable
  map(f, l) == construct [f s for s in entries l]
scan(fn, l, ident) ==
  empty? l => empty()
  val := fn(first l, ident)
  concat(val, scan(fn, rest l, val))
else -- A is a list-oid, B a mutable array-oid
  map(f, l) ==
    i := minIndex(w := new(#l,NIL$Lisp)$B)
    for a in entries l repeat (qsetelt_(w, i, f a); i := inc i)
    w

scan(fn, l, ident) ==
  i := minIndex(w := new(#l,NIL$Lisp)$B)
  vl := ident
  for a in entries l repeat
    vl := qsetelt_(w, i, fn(a, vl))
    i := inc i
  w

else -- A is an array-oid
  reduce(fn, v, ident) ==
    val := ident
    for i in minIndex v .. maxIndex v repeat
      val := fn(qelt(v, i), val)
    val

if B has ListAggregate(R) then -- A is an array-oid, B a list-oid
  map(f, v) ==
    construct [f qelt(v, i) for i in minIndex v .. maxIndex v]

scan(fn, v, ident) ==
  w := empty()$B
  for i in minIndex v .. maxIndex v repeat
    ident := fn(qelt(v, i), ident)
  w := concat(ident, w)
  reverse_! w
else -- A and B are array-oid's
  if B has shallowlyMutable then -- B is also mutable
    map(f, v) ==
      w := new(#v,NIL$Lisp)$B
      for i in minIndex w .. maxIndex w repeat
        qsetelt_(w, i, f qelt(v, i))
      w

scan(fn, v, ident) ==
  w := new(#v,NIL$Lisp)$B
  vl := ident
  for i in minIndex v .. maxIndex v repeat
vl := qsetelt_!(w, i, fn(qelt(v, i), vl))
else
    map(f, v) ==
        construct [f qelt(v, i) for i in minIndex v .. maxIndex v]

scan(fn, v, ident) ==
    w := empty()$B
    for i in minIndex v .. maxIndex v repeat
        ident := fn(qelt(v, i), ident)
        w := concat(w, ident)
    w

— FLAGG2.dotabb —
"FLAGG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLAGG2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG2" -> "FLAGG"

package FLASORT FiniteLinearAggregateSort

— FiniteLinearAggregateSort.input —

)set break resume
/sys rm -f FiniteLinearAggregateSort.output
/spool FiniteLinearAggregateSort.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteLinearAggregateSort
--R
--R FiniteLinearAggregateSort(S: Type,V)where
--R V: FiniteLinearAggregate(S)with
--R shallowlyMutable is a package constructor
--R Abbreviation for FiniteLinearAggregateSort is FLASORT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FLASORT
This package exports 3 sorting algorithms which work over FiniteLinearAggregates. Sort package (in-place) for shallowlyMutable Finite Linear Aggregates

See Also:
- )show FiniteLinearAggregateSort

Exports:
heapSort quickSort shellSort
package FLASORT FiniteLinearAggregateSort

)abbrev package FLASORT FiniteLinearAggregateSort
++ Author: Michael Monagan Sep/88
++ Description:
++ This package exports 3 sorting algorithms which work over
++ FiniteLinearAggregates.
++ Sort package (in-place) for shallowlyMutable Finite Linear Aggregates
-- the following package is only instantiated over %
-- thus shouldn’t be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('FiniteLinearAggregateSort, $mutableDomains)

FiniteLinearAggregateSort(S, V): Exports == Implementation where
S: Type
V: FiniteLinearAggregate(S) with shallowlyMutable
B ==> Boolean
I ==> Integer

Exports ==> with
quickSort: ((S, S) -> B, V) -> V
++ quickSort(f, agg) sorts the aggregate agg with the ordering function
++ f using the quicksort algorithm.
heapSort : ((S, S) -> B, V) -> V
++ heapSort(f, agg) sorts the aggregate agg with the ordering function
++ f using the heapsort algorithm.
shellSort: ((S, S) -> B, V) -> V
++ shellSort(f, agg) sorts the aggregate agg with the ordering function
++ f using the shellSort algorithm.

Implementation ==> add
siftUp : ((S, S) -> B, V, I, I) -> Void
partition: ((S, S) -> B, V, I, I, I) -> I
QuickSort: ((S, S) -> B, V, I, I) -> V

quickSort(l, r) == QuickSort(l, r, minIndex r, maxIndex r)

siftUp(l, r, i, n) ==
t := qelt(r, i)
while (j := 2*i+1) < n repeat
  if (k := j+1) < n and l(qelt(r, j), qelt(r, k)) then j := k
  if l(t,qelt(r,j)) then
    qsetelt_!(r, i, qelt(r, j))
    qsetelt_!(r, j, t)
    i := j
  else leave
heapSort(l, r) ==
  not zero? minIndex r => error "not implemented"
  n := (#r)::I
  for k in shift(n,-1) - 1 .. 0 by -1 repeat siftUp(l, r, k, n)
  for k in n-1 .. 1 by -1 repeat
    swap!(r, 0, k)
    siftUp(l, r, 0, k)
  r

partition(l, r, i, j, k) ==
  -- partition r[i..j] such that r.s <= r.k <= r.t
  x := qelt(r, k)
  t := qelt(r, i)
  qsetelt_!(r, k, qelt(r, j))
  while i < j repeat
    if l(x,t) then
      qsetelt_!(r, j, t)
      j := j-1
      t := qsetelt_!(r, i, qelt(r, j))
    else (i := i+1; t := qelt(r, i))
    qsetelt_!(r, j, x)
  j

QuickSort(l, r, i, j) ==
  n := j - i
  -- if one? n and l(qelt(r, j), qelt(r, i)) then swap!(r, i, j)
  if (n = 1) and l(qelt(r, j), qelt(r, i)) then swap!(r, i, j)
  n < 2 => return r
  -- for the moment split at the middle item
  k := partition(l, r, i, j, i + shift(n,-1))
  QuickSort(l, r, i, k - 1)
  QuickSort(l, r, k + 1, j)

shellSort(l, r) ==
  m := minIndex r
  n := maxIndex r
  -- use Knuths gap sequence: 1,4,13,40,121,...
  g := 1
  while g <= (n-m) repeat g := 3*g+1
  g := g quo 3
  while g > 0 repeat
    for i in m+g..n repeat
      j := i-g
      while j >= m and l(qelt(r, j+g), qelt(r, j)) repeat
        swap!(r,j,j+g)
        j := j-g
      g := g quo 3
    r
package FSAGG2 FiniteSetAggregateFunctions2

-- FiniteSetAggregateFunctions2.input --

)set break resume
)sys rm -f FiniteSetAggregateFunctions2.output
)spool FiniteSetAggregateFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FiniteSetAggregateFunctions2
--R
--R FiniteSetAggregateFunctions2(S: SetCategory,A: FiniteSetAggregate(S),R: SetCategory,B: FiniteSetAggregateFunctions2)
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FSAGG2
--R
--R-------------------------------------------- Operations --------------------------------------------
--R map : ((S -> R),A) -> B              reduce : (((S,R) -> R),A,R) -> R
--R scan : (((S,R) -> R),A,R) -> B

--R

)spool
)lisp (bye)

-- FiniteSetAggregateFunctions2.help --

====================================================================
FiniteSetAggregateFunctions2 examples
FiniteSetAggregateFunctions2 provides functions involving two finite set aggregates where the underlying domains might be different. An example of this is to create a set of rational numbers by mapping a function across a set of integers, where the function divides each integer by 1000.

See Also:
- )show FiniteSetAggregateFunctions2

---

FiniteSetAggregateFunctions2 (FSAGG2)

Exports:
- map
- reduce
- scan

--- package FSAGG2 FiniteSetAggregateFunctions2 ---

)abbrev package FSAGG2 FiniteSetAggregateFunctions2
++ Author: Robert S. Sutor
++ Date Created: 15 May 1990
++ Date Last Updated: 14 Oct 1993
++ Description:
++ FiniteSetAggregateFunctions2 provides functions involving two
++ finite set aggregates where the underlying domains might be
++ different. An example of this is to create a set of rational
++ numbers by mapping a function across a set of integers, where the
++ function divides each integer by 1000.

FiniteSetAggregateFunctions2(S, A, R, B): Exports == Implementation where
    S, R: SetCategory
A : FiniteSetAggregate S
B : FiniteSetAggregate R

Exports ==> with

map : (S -> R, A) -> B
++ map(f,a) applies function f to each member of
++ aggregate \spad{a}, creating a new aggregate with
++ a possibly different underlying domain.
reduce : ((S, R) -> R, A, R) -> R
++ reduce(f,a,r) applies function f to each
++ successive element of the aggregate \spad{a} and an
++ accumulant initialised to r.
++ For example,
++ \spad{reduce(_+$Integer,[1,2,3],0)}
++ does a \spad{3+(2+(1+0))}.
++ Note that third argument r may be regarded
++ as an identity element for the function.
scan : ((S, R) -> R, A, R) -> B
++ scan(f,a,r) successively applies \spad{reduce(f,x,r)}
++ to more and more leading sub-aggregates x of
++ aggregate \spad{a}.
++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
++ \spad{scan(f,a,r)} returns
++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.

Implementation ==> add
map(fn, a) ==
  set(map(fn, parts a)$ListFunctions2(S, R))$B
reduce(fn, a, ident) ==
  reduce(fn, parts a, ident)$ListFunctions2(S, R)
scan(fn, a, ident) ==
  set(scan(fn, parts a, ident)$ListFunctions2(S, R))$B

— FSAGG2.dotabb —

"FSAGG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSAGG2"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"FSAGG2" -> "FSAGG"

package FLOATCP FloatingComplexPackage

— FloatingComplexPackage.input —
---S 1 of 1
)show FloatingComplexPackage

---R
---R FloatingComplexPackage(Par: Join(Field,OrderedRing)) is a package constructor
---R Abbreviation for FloatingComplexPackage is FLOATCP
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for FLOATCP

---R--------------------------------- Operations ---------------------------------
---R
---R complexRoots : (Fraction(Polynomial(Complex(Integer))),Par) -> List(Complex(Par))
---R complexRoots : (List(Fraction(Polynomial(Complex(Integer)))),List(Symbol),Par) -> List(List(Complex(Par)))
---R complexSolve : (List(Fraction(Polynomial(Complex(Integer)))),Par) -> List(List(Equation(Polynomial(Complex(Par)))))
---R complexSolve : (List(Equation(Fraction(Polynomial(Complex(Integer))))),Par) -> List(List(Equation(Polynomial(Complex(Par)))))
---R complexSolve : (Fraction(Polynomial(Complex(Integer))),Par) -> List(Equation(Polynomial(Complex(Par)))))
---R complexSolve : (Equation(Fraction(Polynomial(Complex(Integer)))),Par) -> List(Equation(Polynomial(Complex(Par)))))

---E 1

)spool
)lisp (bye)

—— FloatingComplexPackage.help ——

====================================================================
FloatingComplexPackage examples
====================================================================

This is a package for the approximation of complex solutions for
systems of equations of rational functions with complex rational
coefficients. The results are expressed as either complex rational
numbers or complex floats depending on the type of the precision
parameter which can be either a rational number or a floating point number.

See Also:
o )show FloatingComplexPackage

——
FloatingComplexPackage (FLOATCP)

Exports:
  complexRoots  complexSolve

— package FLOATCP FloatingComplexPackage —

)abbrev package FLOATCP FloatingComplexPackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Description:
++ This is a package for the approximation of complex solutions for
++ systems of equations of rational functions with complex rational
++ coefficients. The results are expressed as either complex rational
++ numbers or complex floats depending on the type of the precision
++ parameter which can be either a rational number or a floating point number.

FloatingComplexPackage(Par): Cat == Cap where
  Par : Join(Field, OrderedRing)
  K ==> GI
  FPK ==> Fraction P K
  C ==> Complex
  I ==> Integer
  NNI ==> NonNegativeInteger
  P ==> Polynomial
  EQ ==> Equation
  L ==> List
  SUP ==> SparseUnivariatePolynomial
  RN ==> Fraction Integer
  NF ==> Float
  CF ==> Complex Float
  GI ==> Complex Integer
  GRN ==> Complex RN
  SE ==> Symbol
  RFI ==> Fraction P I
  INFSP ==> InnerNumericFloatSolvePackage
Cat == with


complexSolve: (L FPK,Par) -> L L EQ P C Par
  ++ complexSolve(lp,eps) finds all the complex solutions to
  ++ precision eps of the system lp of rational functions
  ++ over the complex rationals with respect to all the
  ++ variables appearing in lp.

complexSolve: (L EQ FPK,Par) -> L L EQ P C Par
  ++ complexSolve(leq,eps) finds all the complex solutions
  ++ to precision eps of the system leq of equations
  ++ of rational functions over complex rationals
  ++ with respect to all the variables appearing in lp.

complexSolve: (FPK,Par) -> L EQ P C Par
  ++ complexSolve(p,eps) find all the complex solutions of the
  ++ rational function p with complex rational coefficients
  ++ with respect to all the variables appearing in p,
  ++ with precision eps.

complexSolve: (EQ FPK,Par) -> L EQ P C Par
  ++ complexSolve(eq,eps) finds all the complex solutions of the
  ++ equation eq of rational functions with rational rational coefficients
  ++ with respect to all the variables appearing in eq,
  ++ with precision eps.

complexRoots: (FPK,Par) -> L C Par
  ++ complexRoots(rf, eps) finds all the complex solutions of a
  ++ univariate rational function with rational number coefficients.
  ++ The solutions are computed to precision eps.

complexRoots: (L FPK,L SE,Par) -> L L C Par
  ++ complexRoots(lrf, lv, eps) finds all the complex solutions of a
  ++ list of rational functions with rational number coefficients
  ++ with respect the the variables appearing in lv.
  ++ Each solution is computed to precision eps and returned as
  ++ list corresponding to the order of variables in lv.

Cap == add

-- find the complex zeros of an univariate polynomial --
complexRoots(q:FPK,eps:Par) : L C Par ==
  p:=numer q
  complexZeros(univariate p,eps)$ComplexRootPackage(SUP GI, Par)

-- find the complex zeros of an univariate polynomial --
complexRoots(lp:L FPK,lv:L SE,eps:Par) : L L C Par ==
  lnum:=[numer p for p in lp]
  lden:=[dp for p in lp |(dp:=denom p)^=1]
innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)

complexSolve(lp:L FPK,eps : Par) : L L EQ P C Par ==
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |((dp:=denom p)^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden=[] then
lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
[[equation(x::(P C Par),r::(P C Par)) for x in lv for r in nres]
 for nres in innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)]

complexSolve(le:L EQ FPK,eps : Par) : L L EQ P C Par ==
lp:=[lhs ep - rhs ep for ep in le]
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |((dp:=denom p)^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden=[] then
lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
[[equation(x::(P C Par),r::(P C Par)) for x in lv for r in nres]
 for nres in innerSolve(lnum,lden,lv,eps)$INFSP(K,C Par,Par)]

complexSolve(p : FPK,eps : Par) : L EQ P C Par ==
(mvar := mainVariable numer p ) case "failed" =>
   error "no variable found"
x:P C Par:=mvar::SE::(P C Par)
[equation(x,val::(P C Par)) for val in complexRoots(p,eps)]

complexSolve(eq : EQ FPK,eps : Par) : L EQ P C Par ==
complexSolve(lhs eq - rhs eq,eps)

package FLOATRP FloatingRealPackage

--- FLOATTCP.dotabb ---
"FLOATTCP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLOATTCP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"FLOATTCP" -> "COMPCAT"

--- FloatingRealPackage.input ---
)set break resume
This is a package for the approximation of real solutions for systems of polynomial equations over the rational numbers. The results are expressed as either rational numbers or floats depending on the type of the precision parameter which can be either a rational number or a floating point number.

See Also:
- `)show FloatingRealPackage`
FloatingRealPackage (FLOATRP)

Exports:
realRoots solve

--- package FLOATRP FloatingRealPackage ---

)abbrev package FLOATRP FloatingRealPackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Description:
++ This is a package for the approximation of real solutions for
++ systems of polynomial equations over the rational numbers.
++ The results are expressed as either rational numbers or floats
++ depending on the type of the precision parameter which can be
++ either a rational number or a floating point number.

FloatingRealPackage(Par): Cat == Cap where
  I      => Integer
  NNI    => NonNegativeInteger
  P      => Polynomial
  EQ     => Equation
  L      => List
  SUP    => SparseUnivariatePolynomial
  RN     => Fraction Integer
  NF     => Float
  CF     => Complex Float
  GI     => Complex Integer
  GRN    => Complex RN
  SE     => Symbol
  RFI    => Fraction P I
  INFSP  => InnerNumericFloatSolvePackage

  Par : Join(OrderedRing, Field) -- RN or NewFloat

  Cat == with
solve: (L RFI,Par) -> L L EQ P Par
++ solve(lp,eps) finds all of the real solutions of the
++ system lp of rational functions over the rational numbers
++ with respect to all the variables appearing in lp,
++ with precision eps.

solve: (L EQ RFI,Par) -> L L EQ P Par
++ solve(leq,eps) finds all of the real solutions of the
++ system leq of equations of rational functions
++ with respect to all the variables appearing in lp,
++ with precision eps.

solve: (RFI,Par) -> L EQ P Par
++ solve(p,eps) finds all of the real solutions of the
++ univariate rational function p with rational coefficients
++ with respect to the unique variable appearing in p,
++ with precision eps.

solve: (EQ RFI,Par) -> L EQ P Par
++ solve(eq,eps) finds all of the real solutions of the
++ univariate equation eq of rational functions
++ with respect to the unique variables appearing in eq,
++ with precision eps.

realRoots: (L RFI,L SE,Par) -> L L Par
++ realRoots(lp,lv,eps) computes the list of the real
++ solutions of the list lp of rational functions with rational
++ coefficients with respect to the variables in lv,
++ with precision eps. Each solution is expressed as a list
++ of numbers in order corresponding to the variables in lv.

realRoots : (RFI,Par) -> L Par
++ realRoots(rf, eps) finds the real zeros of a univariate
++ rational function with precision given by eps.

Cap == add

makeEq(nres:L Par,lv:L SE) : L EQ P Par ==
  [equation(x::(P Par),r::(P Par)) for x in lv for r in nres]

-- find the real zeros of an univariate rational polynomial --
realRoots(p:RFI,eps:Par) : L Par ==
  innerSolve(numer p,eps)$INFSP(I,Par,Par)

-- real zeros of the system of polynomial lp --
realRoots(lp:L RFI,lv:L SE,eps: Par) : L L Par ==
  lnum:=[numer p for p in lp]
  lden:=[dp for p in lp |((dp:=denom p)^=1]
  innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)
solve(lp:L RFI,eps : Par) : L L EQ P Par ==
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p)^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden=[] then
  lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
[makeEq(numres,lv) for numres
  in innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)]

solve(le:L EQ RFI,eps : Par) : L L EQ P Par ==
lp:=[lhs ep - rhs ep for ep in le]
lnum:=[numer p for p in lp]
lden:=[dp for p in lp |(dp:=denom p)^=1]
lv:="setUnion"/[variables np for np in lnum]
if lden=[] then
  lv:=setUnion(lv,"setUnion"/[variables dp for dp in lden])
[makeEq(numres,lv) for numres
  in innerSolve(lnum,lden,lv,eps)$INFSP(I,Par,Par)]

solve(p : RFI,eps : Par) : L EQ P Par ==
(mvar := mainVariable numer p ) case "failed" =>
  error "no variable found"
x:P Par:=mvar::SE::(P Par)
[equation(x,val::(P Par)) for val in realRoots(p,eps)]

solve(eq : EQ RFI,eps : Par) : L EQ P Par ==
solve(lhs eq - rhs eq,eps)

---

-- FLOATRP.dotabb --

"FLOATRP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FLOATRP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FLOATRP" -> "PFECAT"

---

package FCPAK1 FortranCodePackage1

-- FortranCodePackage1.input --

)set break resume
)sys rm -f FortranCodePackage1.output
FortranCodePackage1 provides some utilities for producing useful objects in FortranCode domain.

The Package may be used with the FortranCode domain and its printCode or possibly via an outputAsFortran.

The package provides items of use in connection with ASPs in the AXIOM-NAG link and, where appropriate, naming accords with that in IRENA.

The easy-to-use functions use Fortran loop variables I1, I2, and it is users' responsibility to check that this is sensible. The advanced functions use SegmentBinding to allow users control over Fortran loop variable names.
FortranCodePackage1 (FCPAK1)

Exports:
identitySquareMatrix zeroMatrix zeroSquareMatrix zeroVector

--- package FCPAK1 FortranCodePackage1 ---

)abbrev package FCPAK1 FortranCodePackage1
++ Author: Grant Keady and Godfrey Nolan
++ Date Created: April 1993
++ Description:
++ \spadtype{FortranCodePackage1} provides some utilities for
++ producing useful objects in FortranCode domain.
++ The Package may be used with the FortranCode domain and its
++ \spad{printCode} or possibly via an outputAsFortran.
++ (The package provides items of use in connection with ASPs
++ in the AXIOM-NAG link and, where appropriate, naming accords
++ with that in IRENA.)
++ The easy-to-use functions use Fortran loop variables I1, I2,
++ and it is users' responsibility to check that this is sensible.
++ The advanced functions use SegmentBinding to allow users control
++ over Fortran loop variable names.
-- Later might add functions to build
-- diagonalMatrix from List, i.e. the FC version of the corresponding
-- AXIOM function from MatrixCategory;
-- bandedMatrix, i.e. the full-matrix-FC version of the corresponding
-- AXIOM function in BandedMatrix Domain
-- bandedSymmetricMatrix, i.e. the full-matrix-FC version of the corresponding
-- AXIOM function in BandedSymmetricMatrix Domain
FortranCodePackage1: Exports == Implementation where

NNI ==> NonNegativeInteger
PI  ==> PositiveInteger
PIN ==> Polynomial(Integer)
SBINT ==> SegmentBinding(Integer)
SEGINT ==> Segment(Integer)
LSBINT ==> List(SegmentBinding(Integer))
SBPIN ==> SegmentBinding(Polynomial(Integer))
SEGPIN ==> Segment(Polynomial(Integer))
LSBPIN ==> List(SegmentBinding(Polynomial(Integer)))
FC  ==> FortranCode

EXPRESSION ==> Union(Expression Integer,Expression Float,Expression Complex Integer,Expression Complex Float)

Exports == with

zeroVector: (Symbol,PIN) -> FC
++ zeroVector(s,p) undocumented{}

zeroMatrix: (Symbol,PIN,PIN) -> FC
++ zeroMatrix(s,p,q) uses loop variables in the Fortran, I1 and I2
++ zeroMatrix(s,b,d) in this version gives the user control
++ over names of Fortran variables used in loops.

zeroSquareMatrix: (Symbol,PIN) -> FC
++ zeroSquareMatrix(s,p) undocumented{}

identitySquareMatrix: (Symbol,PIN) -> FC
++ identitySquareMatrix(s,p) undocumented{}

Implementation ==> add

import FC

zeroVector(fname:Symbol,n:PIN):FC ==
ue:Expression(Integer) := 0
i1:Symbol := "I1"::Symbol
lp1:PIN := 1::PIN
hp1:PIN := n
segp1:SEGPIN:= segment(lp1,hp1)$SEGPIN
segbp1:SBPIN := equation(i1,segp1)$SBPIN
ip1:PIN := i1::PIN
indices:List(PIN) := [ip1]
fa:FC := forLoop(segbp1,assign(fname,indices,ue)$FC)$FC
fa

zeroMatrix(fname:Symbol,m:PIN,n:PIN):FC ==
ue:Expression(Integer) := 0
i1:Symbol := "I1"::Symbol
lp1:PIN := 1::PIN
hp1:PIN := m
segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
segbp1:SBPIN := equation(i1,segp1)$SBPIN
i2:Symbol := "I2"::Symbol
hp2:PIN := n
segp2:SEGPIN := segment(lp1,hp2)$SEGPIN
segbp2:SBPIN := equation(i2,segp2)$SBPIN
ip1:PIN := i1::PIN
ip2:PIN := i2::PIN
indices:List(PIN) := [ip1,ip2]
fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
fa

zeroMatrix(fname:Symbol,segbp1:SBPIN,segbp2:SBPIN):FC ==
ue:Expression(Integer) := 0
i1:Symbol := variable(segbp1)$SBPIN
i2:Symbol := variable(segbp2)$SBPIN
ip1:PIN := i1::PIN
ip2:PIN := i2::PIN
indices:List(PIN) := [ip1,ip2]
fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
fa

zeroSquareMatrix(fname:Symbol,n:PIN):FC ==
ue:Expression(Integer) := 0
i1:Symbol := "I1"::Symbol
lp1:PIN := 1::PIN
hp1:PIN := n
segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
segbp1:SBPIN := equation(i1,segp1)$SBPIN
i2:Symbol := "I2"::Symbol
segbp2:SBPIN := equation(i2,segp1)$SBPIN
ip1:PIN := i1::PIN
ip2:PIN := i2::PIN
indices:List(PIN) := [ip1,ip2]
fa:FC :=forLoop(segbp1,forLoop(segbp2,assign(fname,indices,ue)$FC)$FC)$FC
fa

identitySquareMatrix(fname:Symbol,n:PIN):FC ==
ue:Expression(Integer) := 0
u1:Expression(Integer) := 1
i1:Symbol := "I1"::Symbol
lp1:PIN := 1::PIN
hp1:PIN := n
segp1:SEGPIN := segment(lp1,hp1)$SEGPIN
segbp1:SBPIN := equation(i1,segp1)$SBPIN
i2:Symbol := "I2"::Symbol
segbp2:SBPIN := equation(i2,segp1)$SBPIN
package FOP FortranOutputStackPackage

— FortranOutputStackPackage.input —

)set break resume
)sys rm -f FortranOutputStackPackage.output
)spool FortranOutputStackPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranOutputStackPackage
--R
--R FortranOutputStackPackage is a package constructor
--R Abbreviation for FortranOutputStackPackage is FOP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FOP
--R
--R------------------------------- Operations --------------------------------
--R popFortranOutputStack : () -> Void  topFortranOutputStack : () -> String
--R clearFortranOutputStack : () -> Stack(String)
--R pushFortranOutputStack : FileName -> Void
--R pushFortranOutputStack : String -> Void
--R showFortranOutputStack : () -> Stack(String)
--R
--E 1

)spool
)lisp (bye)

——

— FortranOutputStackPackage.help —

====================================================================
FortranOutputStackPackage examples
====================================================================

Code to manipulate Fortran Output Stack

See Also:
  o )show FortranOutputStackPackage

——

FortranOutputStackPackage (FOP)

Exports:
  popFortranOutputStack  clearFortranOutputStack  pushFortranOutputStack
  pushFortranOutputStack  showFortranOutputStack  topFortranOutputStack

— package FOP FortranOutputStackPackage —

)abbrev package FOP FortranOutputStackPackage
-- Because of a bug in the compiler:
)bo $noSubsumption:=false
++ Author: Mike Dewar
++ Date Created: October 1992
++ Description:
++ Code to manipulate Fortran Output Stack

FortranOutputStackPackage() : specification == implementation where

specification == with

  clearFortranOutputStack : () -> Stack String
  ++ clearFortranOutputStack() clears the Fortran output stack
  showFortranOutputStack : () -> Stack String
  ++ showFortranOutputStack() returns the Fortran output stack
  popFortranOutputStack : () -> Void
  ++ popFortranOutputStack() pops the Fortran output stack
  pushFortranOutputStack : FileName -> Void
  ++ pushFortranOutputStack(f) pushes f onto the Fortran output stack
  pushFortranOutputStack : String -> Void
  ++ pushFortranOutputStack(f) pushes f onto the Fortran output stack
  topFortranOutputStack : () -> String
  ++ topFortranOutputStack() returns the top element of the Fortran output stack

implementation == add

import MoreSystemCommands

-- A stack of filenames for Fortran output. We are sharing this with
-- the standard Fortran output code, so want to be a bit careful about
-- how we interact with what the user does independently. We get round
-- potential problems by always examining the top element of the stack
-- before we push. If the user has redirected output then we alter our
-- top value accordingly.
fortranOutputStack : Stack String := empty()@(Stack String)

topFortranOutputStack():String == string(_$fortranOutputFile$Lisp)

pushFortranOutputStack(fn:FileName):Void ==
  if empty? fortranOutputStack then
    push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
  else if not(top(fortranOutputStack)=string(_$fortranOutputFile$Lisp)) then
    pop! fortranOutputStack
    push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
    push!( fn::String,fortranOutputStack)
systemCommand concat(['set output fortran quiet ', fn::String])$String
    void()

pushFortranOutputStack(fn:String):Void ==
  if empty? fortranOutputStack then
push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
else if not(top(fortranOutputStack)=string(_$fortranOutputFile$Lisp)) then
  pop! fortranOutputStack
  push!(string(_$fortranOutputFile$Lisp),fortranOutputStack)
push!( fn,fortranOutputStack)
systemCommand concat(["set output fortran quiet ", fn])$String
void()

popFortranOutputStack():Void ==
  if not empty? fortranOutputStack then pop! fortranOutputStack
  if empty? fortranOutputStack then push!(["CONSOLE",fortranOutputStack)
  systemCommand concat(["set output fortran quiet append ", 
                  top fortranOutputStack])$String
void()

clearFortranOutputStack():Stack String ==
  fortranOutputStack := empty()@(Stack String)

showFortranOutputStack():Stack String ==
  fortranOutputStack

———
— FOP.dotabb —

"FOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FOP"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"FOP" -> "STRING"

———

package FORT FortranPackage

— FortranPackage.input —

)set break resume
)sys rm -f FortranPackage.output
)spool FortranPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FortranPackage
---R
--R FortranPackage is a package constructor
--R Abbreviation for FortranPackage is FORT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FORT
--R
--R------------------------------- Operations --------------------------------
--R outputAsFortran : FileName -> Void
--R linkToFortran : (Symbol,List(Union(array: List(Symbol),scalar: Symbol)),List(List(Union(array: List(Symbol),scalar: Symbol))),List(Symbol)) -> SExpression
--R linkToFortran : (Symbol,List(Symbol),TheSymbolTable,List(Symbol)) -> SExpression
--R setLegalFortranSourceExtensions : List(String) -> List(String)
--R
--E 1

)spool
)lisp (bye)

——

FortranPackage.help —

====================================================================
FortranPackage examples
====================================================================

This package provides an interface to the boot code for calling Fortran

See Also:
  o )show FortranPackage

——

FortranPackage (FORT)
Exports:
outputAsFortran  linkToFortran  setLegalFortranSourceExtensions

— package FORT FortranPackage —

)abbrev package FORT FortranPackage
-- Because of a bug in the compiler:
)bo $noSubsumption:=true

++ Author: Mike Dewar
++ Date Created: October 6 1991
++ Date Last Updated: 13 July 1994
++ Description:
++ provides an interface to the boot code for calling Fortran

FortranPackage(): Exports == Implementation where
FST ==> FortranScalarType
SEX ==> SExpression
L ==> List
S ==> Symbol
FOP ==> FortranOutputStackPackage
U ==> Union(array:L S, scalar:S)
Exports ==> with
  linkToFortran: (S, L U, L L U, L S) -> SEX
  ++ linkToFortran(s,l,ll,lv)  undocumented{}
  linkToFortran: (S, L U, L L U, L S, S) -> SEX
  ++ linkToFortran(s,l,lv,t)  undocumented{}
  linkToFortran: (S,L S,TheSymbolTable,L S) -> SEX
  ++ linkToFortran(s,l,t,lv)  undocumented{}
outputAsFortran: FileName -> Void
  ++ outputAsFortran(fn)  undocumented{}
setLegalFortranSourceExtensions: List String -> List String
  ++ setLegalFortranSourceExtensions(l)  undocumented{}

Implementation ==> add

legalFortranSourceExtensions : List String := ["f"]

setLegalFortranSourceExtensions(l:List String):List String ==
  legalFortranSourceExtensions := l

checkExtension(fn : FileName) : String ==
  -- Does it end in a legal extension ?
  stringFn := fn::String
  not member?(extension fn,legalFortranSourceExtensions) =>
    error [stringFn,"is not a legal Fortran Source File."]
  stringFn
outputAsFortran(fn: FileName): Void ==
    -- source : String := checkExtension fn
    source : String := fn::String
    not readable? fn =>
        popFortranOutputStack()$FOP
        error([source," is not readable"]@List(String))
    target : String := topFortranOutputStack()$FOP
    command : String :=
        concat(["sys rm -f ",target," ; cp ",source," ",target]]$String
    systemCommand(command)$MoreSystemCommands
    void()$Void

linkToFortran(name:S, args:L U, decls:L L U, res:L(S)):SEX ==
    makeFort(name, args, decls, res, NIL$Lisp, NIL$Lisp)$Lisp

linkToFortran(name:S, args:L U, decls:L L U, res:L(S), returnType:S):SEX ==
    makeFort(name, args, decls, res, returnType, NIL$Lisp)$Lisp

dimensions(type:FortranType):SEX ==
    convert([convert(convert(u)@InputForm)@SEX _
                for u in dimensionsOf(type)])@SEX

ftype(name:S, type:FortranType):SEX ==
    [name, scalarTypeOf(type), dimensions(type), external? type]$Lisp

makeAspList(asp:S, syms:TheSymbolTable): SExpression==
    symtab : SymbolTable := symbolTableOf(asp, syms)
    [asp, returnTypeOf(asp, syms), argumentListOf(asp, syms), _
        [ftype(u, fortranTypeOf(u, symtab)) for u in parametersOf symtab]]$Lisp

linkToFortran(name:S, aArgs:L S, syms:TheSymbolTable, res:L S):SEX ==
    arguments : L S := argumentListOf(name, syms)$TheSymbolTable
    dummies : L S := setDifference(arguments, aArgs)
    symbolTable:SymbolTable := symbolTableOf(name, syms)
    symbolList := newTypeLists(symbolTable)
    rt: Union(fst: FST, void: "void") := returnTypeOf(name, syms)$TheSymbolTable

    -- Look for arguments which are subprograms
    asps :=[makeAspList(u, syms) for u in externalList(symbolTable)$SymbolTable] $SymbolTable
    rt case fst =>
        makeFort1(name, arguments, aArgs, dummies, symbolList, res, (rt.fst)::S, asps)$Lisp
    makeFort1(name, arguments, aArgs, dummies, symbolList, res, NIL$Lisp, asps)$Lisp

——

— FORT.dotabb —
package FRIDEAL2 FractionalIdealFunctions2

--- FractionalIdealFunctions2.input ---

)set break resume
)sys rm -f FractionalIdealFunctions2.output
)spool FractionalIdealFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FractionalIdealFunctions2
--R
--R FractionalIdealFunctions2(R1: EuclideanDomain,F1: QuotientFieldCategory(R1),U1: UnivariatePolynomialCategory(F1),A1: ... UnivariatePolynomialCategory(F2),A2: Join(FramedAlgebra(F2,U2),RetractableTo(F2))) is a package constructor
--R Abbreviation for FractionalIdealFunctions2 is FRIDEAL2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FRIDEAL2
--R
--R----------------------------------------------------- Operations --------------------------------
--R map : ((R1 -> R2),FractionalIdeal(R1,F1,U1,A1)) -> FractionalIdeal(R2,F2,U2,A2)
--R
--E 1

)spool
)lisp (bye)

--- FractionalIdealFunctions2.help ---

====================================================================
FractionalIdealFunctions2 examples
====================================================================

Lifting of morphisms to fractional ideals.

See Also:
  o )show FractionalIdealFunctions2
FractionalIdealFunctions2 (FRIDEAL2)

Exports:
map

--- package FRIDEAL2 FractionalIdealFunctions2 ---

)abbrev package FRIDEAL2 FractionalIdealFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1 Feb 1989
++ Date Last Updated: 27 Feb 1990
++ Description:
++ Lifting of morphisms to fractional ideals.

FractionalIdealFunctions2(R1, F1, U1, A1, R2, F2, U2, A2):
Exports == Implementation where
  R1, R2: EuclideanDomain
  F1: QuotientFieldCategory R1
  U1: UnivariatePolynomialCategory F1
  A1: Join(FramedAlgebra(F1, U1), RetractableTo F1)
  F2: QuotientFieldCategory R2
  U2: UnivariatePolynomialCategory F2
  A2: Join(FramedAlgebra(F2, U2), RetractableTo F2)

Exports ==> with
  map: (R1 -> R2, FractionalIdeal(R1, F1, U1, A1)) -> FractionalIdeal(R2, F2, U2, A2)
  ++ map(f,i) \undocumented{}

Implementation ==> add
  fmap: (F1 -> F2, A1) -> A2
fmap(f, a) ==
  v := coordinates a
  represents
  [f qelt(v, i) for i in minIndex v .. maxIndex v] Vector(F2)

map(f, i) ==
  b := basis i
  ideal [fmap(s -> f(numer s) / f(denom s), qelt(b, j))
    for j in minIndex b .. maxIndex b] Vector(A2)

---

<table>
<thead>
<tr>
<th>FRIDEAL2.dotabb</th>
</tr>
</thead>
</table>

"FRIDEAL2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRIDEAL2"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"FRIDEAL2" -> "FRAMALG"

---

package FFFG FractionFreeFastGaussian

--- FractionFreeFastGaussian.input ---

)set break resume
)sys rm -f FractionFreeFastGaussian.output
)spool FractionFreeFastGaussian.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FractionFreeFastGaussian

--R
--R FractionFreeFastGaussian(D: Join(IntegralDomain,GcdDomain),V: AbelianMonoidRing(D,NonNegativeInteger)) is a package constructor
--R Abbreviation for FractionFreeFastGaussian is FFFG
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFG
--R
--R---------------------------------------------------------- Operations ----------------------------------
--R DiffAction : (NonNegativeInteger,NonNegativeInteger,V) -> D
--R DiffC : NonNegativeInteger -> List(D)
--R ShiftAction : (NonNegativeInteger,NonNegativeInteger,V) -> D
--R ShiftC : NonNegativeInteger -> List(D)
This package implements the interpolation algorithm proposed in Beckermann, Bernhard and Labahn, George, Fraction-free computation of matrix rational interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and Applications 22.

The packages defined in this file provide fast fraction free rational interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)
FractionFreeFastGaussian (FFFG)

Exports:
- DiffAction
- DiffC
- ShiftAction
- ShiftC
- fffg generalCoefficient
- generalInterpolation
- interpolate
- qShiftAction
- qShiftC

— package FFFG FractionFreeFastGaussian —

\)

Abbrev package FFFG FractionFreeFastGaussian
++ Author: Martin Rubey
++ Description:
++ This package implements the interpolation algorithm proposed in Beckermann,
++ Bernhard and Labahn, George, Fraction-free computation of matrix rational
++ interpolants and matrix GCDs, SIAM Journal on Matrix Analysis and
++ Applications 22.
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FractionFreeFastGaussian(D, V): Exports == Implementation where
D: Join(IntegralDomain, GcdDomain)
V: AbelianMonoidRing(D, NonNegativeInteger) -- for example, SUP D

SUP => SparseUnivariatePolynomial

cFunction => (NonNegativeInteger, Vector SUP D) -> D

CoeffAction => (NonNegativeInteger, NonNegativeInteger, V) -> D

Exports == with

  fffg: (List D, cFunction, List NonNegativeInteger) -> Matrix SUP D
  ++ \spad{fffg} is the general algorithm as proposed by Beckermann and
  ++ Labahn.
  ++
  ++ The first argument is the list of c_{i,i}. These are the only values
++ of C explicitly needed in \spad{fffg}.
++
++ The second argument c, computes c_k(M), i.e., c_k(.) is the dual basis
++ of the vector space V, but also knows about the special multiplication
++ rule as described in Equation (2). Note that the information about f
++ is therefore encoded in c.
++
++ The third argument is the vector of degree bounds n, as introduced in
++ \underline{Definition 2.1}. In particular, the sum of the entries is the order of
++ the Mahler system computed.

interpolate: (List D, List D, NonNegativeInteger) \rightarrow \text{Fraction SUP D}
++ \spad{interpolate(xlist, ylist, deg)} returns the rational function with
++ numerator degree at most \spad{deg} and denominator degree at most
++ \spad{#xlist-deg-1} that interpolates the given points using
++ fraction free arithmetic. Note that rational interpolation does not
++ guarantee that all given points are interpolated correctly:
++ unattainable points may make this impossible.

The following function could be moved to FFFGF, parallel to
\underline{generalInterpolation}. However, the reason for moving
\underline{generalInterpolation} for fractions to a separate package was the need of
a generic signature, hence the extra argument VF to FFFGF. In the
special case of rational interpolation, this extra argument is not necessary,
since we are always returning a fraction of SUPs, and ignore V. In
fact, V is not needed for fffg itself, only if we want to specify a
\underline{CoeffAction}.

Thus, maybe it would be better to move fffg to a separate package?

---

package FFFG FractionFreeFastGaussian ---

interpolate: (List Fraction D, List Fraction D, NonNegativeInteger) \rightarrow \text{Fraction SUP D}
++ \spad{interpolate(xlist, ylist, deg)} returns the rational function with
++ numerator degree \spad{deg} that interpolates the given points using
++ fraction free arithmetic.

generalInterpolation: (List D, CoeffAction,
Vector V, List NonNegativeInteger) \rightarrow \text{Matrix SUP D}
++ \spad{generalInterpolation(C, CA, f, \eta)} performs Hermite-Pade
++ approximation using the given action CA of polynomials on the elements
++ of f. The result is guaranteed to be correct up to order
++ \spad{#eta-1}. Given that \eta is a "normal" point, the degrees on the
++ diagonal are given by \eta. The degrees of column i are in this case
++ eta + e.i - [1,1,...,1], where the degree of zero is -1.
++
++ The first argument C is the list of coefficients c_{k,k} in the
++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
++
++ The second argument, CA(k, l, f), should return the coefficient of x^k
++ in z^l f(x).

generalInterpolation: (List D, CoeffAction, Vector V, NonNegativeInteger, NonNegativeInteger) -> Stream Matrix SUP D
++ \spad{generalInterpolation(C, CA, f, sumEta, maxEta)} applies
++ \spad{generalInterpolation(C, CA, f, eta)} for all possible \spad{eta}
++ with maximal entry \spad{maxEta} and sum of entries at most
++ \spad{sumEta}.
++
++ The first argument C is the list of coefficients c_{k,k} in the
++ expansion <x^k> z g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x).
++
++ The second argument, CA(k, l, f), should return the coefficient of x^k
++ in z^l f(x).

generalCoefficient: (CoeffAction, Vector V, NonNegativeInteger, Vector SUP D) -> D
++ \spad{generalCoefficient(action, f, k, p)} gives the coefficient of
++ x^k in p(z)\dot f(x), where the action of z^l on a polynomial in x is
++ given by action, i.e., action(k, l, f) should return the coefficient
++ of x^k in z^l f(x).

ShiftAction: (NonNegativeInteger, NonNegativeInteger, V) -> D
++ \spad{ShiftAction(k, l, g)} gives the coefficient of x^k in z^l g(x),
++ where \spad{z*(a+bx+c*x^2+d*x^3+...)} = (b*x+2*c*x^2+3*d*x^3+...). In
++ terms of sequences, z*u(n)=n*u(n).

ShiftC: NonNegativeInteger -> List D
++ \spad{ShiftC} gives the coefficients c_{k,k} in the expansion <x^k> z
++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
++ shifting. In fact, the result is [0,1,2,...]

DiffAction: (NonNegativeInteger, NonNegativeInteger, V) -> D
++ \spad{DiffAction(k, l, g)} gives the coefficient of x^k in z^l g(x),
++ where z*(a+bx+c*x^2+d*x^3+...) = (a*x+b*x^2+c*x^3+...), i.e.,
++ multiplication with x.

DiffC: NonNegativeInteger -> List D
++ \spad{DiffC} gives the coefficients c_{k,k} in the expansion <x^k> z
++ g(x) = sum_{i=0}^k c_{k,i} <x^i> g(x), where z acts on g(x) by
++ shifting. In fact, the result is [0,0,0,...]

qShiftAction: (D, NonNegativeInteger, NonNegativeInteger, V) -> D
++ \spad{qShiftAction(q, k, l, g)} gives the coefficient of \(x^k\) in \(z^l\)
++ \(g(x)\), where \(z*(a+b*x+c*x^2+d*x^3+...) = \)
++ \((a+q*b*x+q^2*c*x^2+q^3*d*x^3+...). In terms of sequences,
++ \(z*u(n) = q^n*u(n).\)

\textbf{qShiftC:} \((D, \text{NonNegativeInteger}) \rightarrow \text{List } D\)
++ \spad{qShiftC} gives the coefficients \(c_{k,k}\) in the expansion
++ \(<x^k> z g(x) = \sum_{i=0}^k c_{k,i} <x^i> g(x)\), where \(z\) acts on \(g(x)\) by
++ shifting. In fact, the result is \([1,q,q^2,...]\)

Implementation ==> add

-- Shift Operator

-- ShiftAction(k, l, f) is the CoeffAction appropriate for the shift operator.

\textbf{ShiftAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V)}: D ==
\(k**l*\text{coefficient}(f, k)\)

\textbf{ShiftC(total: NonNegativeInteger): List D ==}
\([i::D \text{ for } i \text{ in } 0..\text{total}-1]\)

-- q-Shift Operator

-- q-ShiftAction(k, l, f) is the CoeffAction appropriate for the q-shift operator.

\textbf{qShiftAction(q: D, k: NonNegativeInteger, l: NonNegativeInteger, f: V)}: D ==
\(q**(k*l)*\text{coefficient}(f, k)\)

\textbf{qShiftC(q: D, total: NonNegativeInteger): List D ==}
\([q**i \text{ for } i \text{ in } 0..\text{total}-1]\)

-- Differentiation Operator

-- DiffAction(k, l, f) is the CoeffAction appropriate for the differentiation
-- operator.

\textbf{DiffAction(k: NonNegativeInteger, l: NonNegativeInteger, f: V)}: D ==
\(\text{coefficient}(f, (k-l)::\text{NonNegativeInteger})\)

\textbf{DiffC(total: NonNegativeInteger): List D ==}
[0 for i in 1..total]

-- general - suitable for functions f

-- get the coefficient of \( z^k \) in the scalar product of \( p \) and \( f \), the action
-- being defined by coeffAction

generalCoefficient(coeffAction: CoeffAction, f: Vector V,
k: NonNegativeInteger, p: Vector SUP D): D ==
res: D := 0
for i in 1..#f repeat
  a := f.i
  b := p.i
  for l in minimumDegree b..degree b repeat
    if not zero? coefficient(b, l) then
      res := res + coefficient(b, l) * coeffAction(k, l, a)
res

generalInterpolation(C: List D, coeffAction: CoeffAction,
f: Vector V,
eta: List NonNegativeInteger): Matrix SUP D ==
c: cFunction := (x,y) +-> generalCoefficient(coeffAction, f,
(x-1)::NonNegativeInteger, y)
fffg(C, c, eta)

The following function returns the lexicographically next vector with non-negative components smaller than \( p \) with the same sum as \( v \).

— package FFFG FractionFreeFastGaussian —

nextVector!(p: NonNegativeInteger, v: List NonNegativeInteger)
  : Union("failed", List NonNegativeInteger) ==
n := #v
pos := position(x -> x < p, v)
zero? pos => return "failed"
if pos = 1 then
  sum: Integer := v.1
  for i in 2..n repeat
    if v.i < p and sum > 0 then
      v.i := v.i + 1
      sum := sum - 1
    for j in 1..i-1 repeat
      if sum > p then
        v.j := p
        sum := sum - p
      else
        v.j := sum::NonNegativeInteger
        sum := 0
    return v
  else
    sum := sum + v.i
    return "failed"
else
  v.pos := v.pos + 1
  v.(pos-1) := (v.(pos-1) - 1)::NonNegativeInteger
  return v

The following function returns the stream of all possible degree vectors, beginning with v, where the degree vectors are sorted in reverse lexicographic order. Furthermore, the entries are all less or equal to p and their sum equals the sum of the entries of v. We assume that the entries of v are also all less or equal to p.

— package FFFG FractionFreeFastGaussian —

vectorStream(p: NonNegativeInteger, v: List NonNegativeInteger) : Stream List NonNegativeInteger == delay
  next := nextVector!(p, copy v)
  (next case "failed") => empty$Stream(List NonNegativeInteger)
  cons(next, vectorStream(p, next))

vectorStream$2 skips every second entry of vectorStream.

— package FFFG FractionFreeFastGaussian —

vectorStream$2(p: NonNegativeInteger, v: List NonNegativeInteger) : Stream List NonNegativeInteger == delay
  next := nextVector!(p, copy v)
This version of `generalInterpolation` returns a stream of solutions, one for each possible degree vector. Thus, it only needs to apply the previously defined `generalInterpolation` to each degree vector. These are generated by `vectorStream` and `vectorStream2` respectively.

If `f` consists of two elements only, we can skip every second degree vector: note that `ffg`, and thus also `generalInterpolation`, returns a matrix with `#f` columns, each corresponding to a solution of the interpolation problem. More precisely, the `i`th column is a solution with degrees `η − (1, 1, . . . , 1) + ε_i`. Thus, in the case of 2 × 2 matrices, `vectorStream` would produce solutions corresponding to `(d, 0), (d − 1, 1), (d − 2, 2), (d − 3, 3), . . .`, i.e., every second matrix is redundant.

Although some redundancy exists also for higher dimensional `f`, the scheme becomes much more complicated, thus we did not implement it.

---

**package FFFG FractionFreeFastGaussian**

```lisp
generalInterpolation(C: List D, coeffAction: CoeffAction, f: Vector V, sumEta: NonNegativeInteger, maxEta: NonNegativeInteger) : Stream Matrix SUP D ==
```

-- generate an initial degree vector --

```lisp
if #f = 2 then
  map(x +-> generalInterpolation(C, coeffAction, f, x),
       cons(eta, vectorStream2(maxEta, eta)))
  $StreamFunctions2(List NonNegativeInteger, Matrix SUP D)
else
  map(x +-> generalInterpolation(C, coeffAction, f, x),
       cons(eta, vectorStream(maxEta, eta)))
  $StreamFunctions2(List NonNegativeInteger, Matrix SUP D)
```

We need to generate an initial degree vector, being the minimal element in reverse lexicographic order, i.e., `m, m, . . . , m, k, 0, 0, . . .`, where `m` is `maxEta` and `k` is the remainder of `sumEta` divided by `maxEta`. This is done by the following code:

---

**generate an initial degree vector**
sum: Integer := sumEta
entry: Integer
eta: List NonNegativeInteger
:= [(if sum < maxEta
then (entry := sum; sum := 0)
else (entry := maxEta; sum := sum - maxEta);
entry::NonNegativeInteger) for i in 1..#f]

We want to generate all vectors with sum of entries being at most sumEta. Therefore the following is incorrect.

— BUG generate an initial degree vector —

-- (sum > 0) => empty()$Stream(Matrix SUP D)

— package FFFG FractionFreeFastGaussian —

-- rational interpolation

interpolate(x: List Fraction D, y: List Fraction D, d: NonNegativeInteger)
: Fraction SUP D ==
gx := splitDenominator(x)$InnerCommonDenominator(D, Fraction D, List D, List Fraction D)
gy := splitDenominator(y)$InnerCommonDenominator(D, Fraction D, List D, List Fraction D)
r := interpolate(gx.num, gy.num, d)
elt(numer r, monomial(gx.den,1))/(gy.den*elt(denom r, monomial(gx.den,1)))

interpolate(x: List D, y: List D, d: NonNegativeInteger): Fraction SUP D ==
-- berechne Interpolante mit Graden d und N-d-1
if (N := #x) ~= #y then
  error "interpolate: number of points and values must match"
if N <= d then
  error "interpolate: numerator degree must be smaller than number of data points"
c: cFunction := (s,u) +-> y.s * elt(u.2, x.s) - elt(u.1, x.s)
etta: List NonNegativeInteger := [d, (N-d)::NonNegativeInteger]
M := fffg(x, c, eta)
if zero?(M.(2,1)) then M.(1,2)/M.(2,2)
else M.(1,1)/M.(2,1)
Because of Lemma 5.3, M.1.(2,1) and M.1.(2,2) cannot both vanish, since $\eta_\sigma$ is always $\sigma$-normal by Theorem 7.2 and therefore also para-normal, see Definition 4.2.

Because of Lemma 5.1 we have that M.1.(*,2) is a solution of the interpolation problem, if M.1.(2,1) vanishes.

--- package FFFG FractionFreeFastGaussian ---

-- ffg

--- recurrence computes the new matrix $M$, according to the following formulas (cf. Table 2 in Beckermann and Labahn):

\[
\begin{align*}
&\text{Increase order} \\
&\text{for } \ell = 1 \ldots m, \ell \neq \pi \\
&M_{\sigma+1}^{(\ell)} := \left( M_{\sigma}^{(\ell)}(\pi) - M_{\sigma}^{(\ell)}(\ell) \right) / d_{\sigma} \\
&\text{Increase order in column } \pi \\
&M_{\sigma+1}^{(\pi)} := (z - c_{\sigma,\tau}) M_{\sigma}^{(\pi)} \\
&\text{Adjust degree constraints:} \\
&M_{\sigma+1}^{(\pi)} := \left( M_{\sigma+1}^{(\pi)}(\pi) - \sum_{\ell \neq \pi} M_{\sigma+1}^{(\ell)}(\ell) \right) / d_{\sigma}
\end{align*}
\]

Since we do not need the matrix $M_{\sigma}$ anymore, we drop the index and update the matrix destructively. In the following, we write $C_k$ for $c_{\sigma,\tau}$.

--- package FFFG FractionFreeFastGaussian ---

-- a major part of the time is spent here
recurrence(M: Matrix SUP D, pi: NonNegativeInteger, m: NonNegativeInteger, 
\[r: \text{Vector D, d: D, z: SUP D, Ck: D, p: Vector D})\]: Matrix SUP D ==
\[rPi: D := \text{qelt}(r, pi)\]
\[\text{polyf: SUP D := } rPi \ast (z - \text{Ck}\times\text{SUP D})\]
\[\text{for } i \text{ in 1..m repeat} \]
\[M_{\text{iPi}}: \text{SUP D := qelt(M, i, pi)}\]
\[\text{newMiPi: SUP D := polyf } \ast \text{MiPi}\]
\[\text{-- update columns } \pi \text{ and calculate their sum} \]
\[\text{for l in 1..m | l }\neq\text{ pi repeat} \]
\[r1: D := \text{qelt}(r, l)\]
\[\text{-- I need the coercion to SUP D, since exquo returns an element of}\]
-- Union("failed", SUP D)...
  Mil: SUP D := ((qelt(M, i, l) * rPi - MiPi * rl) exquo d)::SUP D
  qsetelt!(M, i, l, Mil)

  pl: D := qelt(p, l)
  newMiPi := newMiPi - pl * Mil

-- update column pi
  qsetelt!(M, i, pi, (newMiPi exquo d)::SUP D)

M

fffg(C: List D, c: cFunction, eta: List NonNegativeInteger): Matrix SUP D ==
-- eta is the vector of degrees. We compute M with degrees eta+i-1, i=1..m
  z: SUP D := monomial(1, 1)
  m: NonNegativeInteger := #eta
  M: Matrix SUP D := scalarMatrix(m, 1)
  d: D := 1
  K: NonNegativeInteger := reduce(_+, eta)
  etak: Vector NonNegativeInteger := zero(m)
  r: Vector D := zero(m)
  p: Vector D := zero(m)
  Lambda: List Integer
  lambdaMax: Integer
  lambda: NonNegativeInteger

  for k in 1..K repeat
    -- k = sigma+1
    for l in 1..m repeat r.l := c(k, column(M, l))
    Lambda := [eta.l-etak.l for l in 1..m | r.l ~= 0]

    -- if Lambda is empty, then M, d and etak remain unchanged. Otherwise, we look
    -- for the next closest para-normal point.
    (empty? Lambda) => "iterate"
    lambdaMax := reduce(max, Lambda)
    lambda := 1
    while eta.lambda-etak.lambda < lambdaMax or r.lambda = 0 repeat
      lambda := lambda + 1

    -- Calculate leading coefficients
    for l in 1..m | l ~= lambda repeat
      if etak.l > 0 then
        p.l := coefficient(M.(l, lambda),
                         (etak.l-1)::NonNegativeInteger)
else
  p.l := 0

-- increase order and adjust degree constraints
M := recurrence(M, lambda, m, r, d, z, C.k, p)
d := r.lambda
etak.lambda := etak.lambda + 1
M

— FFFG.dotabb —

"FFFG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFFG"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"FFFG" -> "PFECAT"

— package FFFGF FractionFreeFastGaussianFractions —

package FFFGF FractionFreeFastGaussianFractions

— FractionFreeFastGaussianFractions.input —

)set break resume
)sys rm -f FractionFreeFastGaussianFractions.output
)spool FractionFreeFastGaussianFractions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FractionFreeFastGaussianFractions
--R
--R FractionFreeFastGaussianFractions(D: Join(IntegralDomain,GcdDomain),V: FiniteAbelianMonoidRing(D,NonNegativeInteger),VF: FiniteAbelianMonoidRing(Fraction(D),NonNegativeInteger)) is a package constructor
--R Abbreviation for FractionFreeFastGaussianFractions is FFFGF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFFGF
--R
--R----------------------------------- Operations -----------------------------------
--R generalInterpolation : (List(D),((NonNegativeInteger,NonNegativeInteger,V) -> D),Vector(VF),List(NonNegativeInteger)) -> Matrix(SparseUnivariatePolynomial(D))
--R generalInterpolation : (List(D),((NonNegativeInteger,NonNegativeInteger,V) -> D),Vector(VF),NonNegativeInteger,NonNegativeInteger) -> Stream(Matrix(SparseUnivariatePolynomial(D)))
FractionFreeFastGaussianFractions (FFFGF)

Exports:
  generalInterpolation

--- package FFFGF FractionFreeFastGaussianFractions ---

)abbrev package FFFGF FractionFreeFastGaussianFractions
CHAPTER 7. CHAPTER F

++ Author: Martin Rubey
++ Description:
++ This package lifts the interpolation functions from
++ \spad\texttt{FractionFreeFastGaussian} to fractions.
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

FractionFreeFastGaussianFractions(D, V, VF): Exports == Implementation where
  D: Join(IntegralDomain, GcdDomain)
  V: FiniteAbelianMonoidRing(D, NonNegativeInteger)
  VF: FiniteAbelianMonoidRing(Fraction D, NonNegativeInteger)

F ==> Fraction D
NNI ==> NonNegativeInteger
SUP ==> SparseUnivariatePolynomial
FFFG ==> FractionFreeFastGaussian
FAMR2 ==> FiniteAbelianMonoidRingFunctions2

cFunction ==> (NNI, Vector SUP D) -> D
CoeffAction ==> (NNI, NNI, V) -> D
-- coeffAction(k, l, f) is the coefficient of x^k in z^l f(x)

Exports == with

  generalInterpolation: (List D, CoeffAction, Vector VF, List NNI)
    -> Matrix SUP D
  ++ \spad{generalInterpolation(l, CA, f, eta)} performs Hermite-Pade
  ++ approximation using the given action CA of polynomials on the elements
  ++ of f. The result is guaranteed to be correct up to order
  ++ |eta|-1. Given that eta is a "normal" point, the degrees on the
  ++ diagonal are given by eta. The degrees of column i are in this case
  ++ eta + e.i - [1,1,...,1], where the degree of zero is -1.

  generalInterpolation: (List D, CoeffAction, Vector VF, NNI, NNI)
    -> Stream Matrix SUP D
  ++ \spad{generalInterpolation(l, CA, f, sumEta, maxEta)} applies
  ++ generalInterpolation(l, CA, f, eta) for all possible eta with maximal
  ++ entry maxEta and sum of entries sumEta

Implementation == add

  multiplyRows!(v: Vector D, M: Matrix SUP D): Matrix SUP D ==
  n := #v
  for i in 1..n repeat
    for j in 1..n repeat
      M.(i,j) := v.i*M.(i,j)

  M
generalInterpolation(C: List D, coeffAction: CoeffAction, 
f: Vector VF, eta: List NNI): Matrix SUP D ==

n := #f
g: Vector V := new(n, 0)
den: Vector D := new(n, 0)

for i in 1..n repeat
  c := coefficients(f.i)
  den.i := commonDenominator(c)$CommonDenominator(D, F, List F)
  g.i :=
    map(x +-> retract(x*den.i)@D, f.i)$FAMR2(NNI, Fraction D, VF, D, V)

M := generalInterpolation(C, coeffAction, g, eta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not 
-- each column. Possibly I could factor out gcd den, but I'm not sure whether 
-- this is efficient.

multiplyRows!(den, M)

generalInterpolation(C: List D, coeffAction: CoeffAction, 
f: Vector VF, sumEta: NNI, maxEta: NNI): Stream Matrix SUP D ==

n := #f
g: Vector V := new(n, 0)
den: Vector D := new(n, 0)

for i in 1..n repeat
  c := coefficients(f.i)
  den.i := commonDenominator(c)$CommonDenominator(D, F, List F)
  g.i :=
    map(x +-> retract(x*den.i)@D, f.i)$FAMR2(NNI, Fraction D, VF, D, V)
  c: cFunction :=
    (x,y) +-> generalCoefficient(coeffAction, g, (x-1)::NNI, y)$FFFG(D, V)

MS: Stream Matrix SUP D
  := generalInterpolation(C, coeffAction, g, sumEta, maxEta)$FFFG(D, V)

-- The following is necessary since I'm multiplying each row with a factor, not 
-- each column. Possibly I could factor out gcd den, but I'm not sure whether 
-- this is efficient.

map(x +-> multiplyRows!(den, x), MS)$Stream(Matrix SUP D)
package FRAC2 FractionFunctions2

FractionFunctions2(A: IntegralDomain,B: IntegralDomain) is a package constructor
Abbreviation for FractionFunctions2 is FRAC2
This constructor is exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for FRAC2

------------------------------------------------ Operations --------------------------------
map : ((A -> B),Fraction(A)) -> Fraction(B)

FractionFunctions2 examples
This package extends a map between integral domains to a map between
Fractions over those domains by applying the map to the numerators and denominators.

See Also:
o )show FractionFunctions2

FractionFunctions2 (FRAC2)

Exports:
map

— package FRAC2 FractionFunctions2 —

)abbrev package FRAC2 FractionFunctions2
++ Description:
++ This package extends a map between integral domains to
++ a map between Fractions over those domains by applying the map to the
++ numerators and denominators.

FractionFunctions2(A, B): Exports == Impl where
A, B: IntegralDomain

R ==> Fraction A
S ==> Fraction B

Exports ==> with
map: (A -> B, R) -> S
  ++ map(func,frac) applies the function func to the numerator
  ++ and denominator of the fraction frac.

Impl ==> add
map(f, r) == map(f, r)$\text{QuotientFieldCategoryFunctions2}(A, B, R, S)

——

— FRAC2.dotabb —

"FRAC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRAC2"]
"ALGEBRA" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ALGEBRA"]
"FRAC2" -> "ALGEBRA"

——

package FRNAAF2 FramedNonAssociativeAlgebraFunctions2

—— FramedNonAssociativeAlgebraFunctions2.input ——

)set break resume
)sys rm -f FramedNonAssociativeAlgebraFunctions2.output
)spool FramedNonAssociativeAlgebraFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FramedNonAssociativeAlgebraFunctions2

--R
--R FramedNonAssociativeAlgebraFunctions2(AR: FramedNonAssociativeAlgebra(R),R: CommutativeRing,AS: FramedNonAssociativeAlgebra(S),S: CommutativeRing) is a package constructor
--R Abbreviation for FramedNonAssociativeAlgebraFunctions2 is FRNAAF2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FRNAAF2
--R
--R-------------------------------- Operations --------------------------------
--R map : ((R -> S),AR) -> AS
--R
--E 1

)spool
)lisp (bye)

——

— FramedNonAssociativeAlgebraFunctions2.help ——
FramedNonAssociativeAlgebraFunctions2 examples

FramedNonAssociativeAlgebraFunctions2 implements functions between two framed non associative algebra domains defined over different rings. The function map is used to coerce between algebras over different domains having the same structural constants.

See Also:
o )show FramedNonAssociativeAlgebraFunctions2

====================================================================

FramedNonAssociativeAlgebraFunctions2 (FRNAAF2)

Exports:
map

FRNAAF2

FRNAALG

)abbrev package FRNAAF2 FramedNonAssociativeAlgebraFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 28 February 1992
++ Date Last Updated: 28 February 1992
++ Description:
++ FramedNonAssociativeAlgebraFunctions2 implements functions between
++ two framed non associative algebra domains defined over different rings.
++ The function map is used to coerce between algebras over different
++ domains having the same structural constants.

FramedNonAssociativeAlgebraFunctions2(AR,R,AS,S) : Exports ==
Implementation where
R: CommutativeRing
S: CommutativeRing
AR: FramedNonAssociativeAlgebra R
AS: FramedNonAssociativeAlgebra S
V -> Vector
Exports => with
map: (R -> S, AR) -> AS
++ map(f,u) maps f onto the coordinates of u to get an element
++ in \spad{AS} via identification of the basis of \spad{AR}
++ as beginning part of the basis of \spad{AS}.
Implementation => add
map(fn : R -> S, u : AR): AS ==
  rank()$AR > rank()$AS => error(“map: ranks of algebras do not fit”)
  vr : V R := coordinates u
  vs : V S := map(fn,vr)$VectorFunctions2(R,S)

This line used to read:

rank()$AR = rank()$AR => represents(vs)$AS

but the test is clearly always true and cannot be what was intended. Gregory Vanuxem supplied the fix below.

— package FRNAAF2 FramedNonAssociativeAlgebraFunctions2 —

rank()$AR = rank()$AS => represents(vs)$AS
ba := basis()$AS
represents(vs,[ba.i for i in 1..rank()$AR])

— FRNAAF2.dotabb —

"FRNAAF2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FRNAAF2"]
"FRNAALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRNAALG"]
"FRNAAF2" -> "FRNAALG"

— FSPECF FunctionalSpecialFunction.input —
### FunctionalSpecialFunction.help

Provides some special functions over an integral domain.

See Also:
  o )show FunctionalSpecialFunction
FunctionalSpecialFunction (FSPECF)

Exports:
- Beta
- Gamma
- abs
- airyAi
- airyBi
- belong?
- besselI
- besselJ
- besselK
- besselY
- digamma
- iiAiryAi
- iiAiryBi
- iiBesselI
- iiBesselJ
- iiBesselK
- iiBesselY
- iiBeta
- iiGamma
- iiabs
- iidigamma
- iioperator
- ipolygamma
- operator
- polygamma

--- package FSPECF FunctionalSpecialFunction ---

)abbrev package FSPECF FunctionalSpecialFunction
++ Author: Manuel Bronstein
++ Date Created: 18 Apr 1989
++ Date Last Updated: 4 October 1993
++ Description:
++ Provides some special functions over an integral domain.

FunctionalSpecialFunction(R, F): Exports == Implementation where
R: Join(OrderedSet, IntegralDomain)
F: FunctionSpace R

OP   ==> BasicOperator
K    ==> Kernel F
SE   ==> Symbol
SPECIALDIFF ==">%specialDiff"

Exports ==>
- belong?: OP -> Boolean
  ++ belong?(op) is true if op is a special function operator;
  operator: OP -> OP
  ++ operator(op) returns a copy of op with the domain-dependent
  ++ properties appropriate for F;
  ++ error if op is not a special function operator
abs : F -> F
  ++ abs(f) returns the absolute value operator applied to f
Gamma : F -> F
  ++ Gamma(f) returns the formal Gamma function applied to f
Gamma : (F,F) -> F
  ++ Gamma(a,x) returns the incomplete Gamma function applied to a and x
Beta: (F,F) -> F
  ++ Beta(x,y) returns the beta function applied to x and y
digamma: F->F
  ++ digamma(x) returns the digamma function applied to x
polygamma: (F,F) -> F
  ++ polygamma(x,y) returns the polygamma function applied to x and y
besselJ: (F,F) -> F
  ++ besselJ(x,y) returns the besselj function applied to x and y
besselY: (F,F) -> F
  ++ besselY(x,y) returns the bessel function applied to x and y
besselI: (F,F) -> F
  ++ besselI(x,y) returns the besseli function applied to x and y
besselK: (F,F) -> F
  ++ besselK(x,y) returns the besselk function applied to x and y
airyAi: F -> F
  ++ airyAi(x) returns the airyai function applied to x
airyBi: F -> F
  ++ airyBi(x) returns the airybi function applied to x

In case we want to have more special function operators here, do not forget to add them to the list specop in CommonOperators. Otherwise they will not have the 'special' attribute and will not be recognised here. One effect could be that

myNewSpecOp(1::Expression Integer)::Expression DoubleFloat

might not re-evaluate the operator.

— package FSPECF FunctionalSpecialFunction —
++ iiBesselJ(x) should be local but conditional;
iiBesselY : List F -> F
++ iiBesselY(x) should be local but conditional;
iiBesselI : List F -> F
++ iiBesselI(x) should be local but conditional;
iiBesselK : List F -> F
++ iiBesselK(x) should be local but conditional;
iiAiryAi : F -> F
++ iiAiryAi(x) should be local but conditional;
iiAiryBi : F -> F
++ iiAiryBi(x) should be local but conditional;

Implementation ==> add
iabs : F -> F
iGamma : F -> F
iBeta : (F, F) -> F
idigamma : F -> F
iiipolygamma: (F, F) -> F
iiiBesselJ : (F, F) -> F
iiiBesselY : (F, F) -> F
iiiBesselI : (F, F) -> F
iiiBesselK : (F, F) -> F
iiAiryAi : F -> F
iiAiryBi : F -> F

opabs := operator("abs":Symbol)$CommonOperators
opGamma := operator("Gamma":Symbol)$CommonOperators
opGamma2 := operator("Gamma2":Symbol)$CommonOperators
opBeta := operator("Beta":Symbol)$CommonOperators
opdigamma := operator("digamma":Symbol)$CommonOperators
oppolygamma := operator("polygamma":Symbol)$CommonOperators
opBesselJ := operator("besselJ":Symbol)$CommonOperators
opBesselY := operator("besselY":Symbol)$CommonOperators
opBesselI := operator("besselI":Symbol)$CommonOperators
opBesselK := operator("besselK":Symbol)$CommonOperators
opAiryAi := operator("airyAi":Symbol)$CommonOperators
opAiryBi := operator("airyBi":Symbol)$CommonOperators

abs x == opabs x
Gamma(x) == opGamma(x)
Gamma(a,x) == opGamma2(a,x)
Beta(x,y) == opBeta(x,y)
digamma x == opdigamma(x)
polygamma(k,x)== oppolygamma(k,x)
besselJ(a,x) == opBesselJ(a,x)
besselY(a,x) == opBesselY(a,x)
besselI(a,x) == opBesselI(a,x)
besselK(a,x) == opBesselK(a,x)
airyAi(x) == opAiryAi(x)
airyBi(x) == opAiryBi(x)
belong? op == has?(op, "special")

operator op ==
  is?(op, "abs"::Symbol) => opabs
  is?(op, "Gamma"::Symbol) => opGamma
  is?(op, "Gamma2"::Symbol) => opGamma2
  is?(op, "Beta"::Symbol) => opBeta
  is?(op, "digamma"::Symbol) => opdigamma
  is?(op, "polygamma"::Symbol) => oppolygamma
  is?(op, "besselJ"::Symbol) => opBesselJ
  is?(op, "besselY"::Symbol) => opBesselY
  is?(op, "besselI"::Symbol) => opBesselI
  is?(op, "besselK"::Symbol) => opBesselK
  is?(op, "airyAi"::Symbol) => opAiryAi
  is?(op, "airyBi"::Symbol) => opAiryBi

  error "Not a special operator"

-- Could put more unconditional special rules for other functions here
iGamma x ==
  (x = 1) => x
  kernel(opGamma, x)

iabs x ==
  zero? x => 0
  is?(x, opabs) => x
  x < 0 => kernel(opabs, -x)
  kernel(opabs, x)

iBeta(x, y) == kernel(opBeta, [x, y])
idigamma x == kernel(opdigamma, x)
dpolygamma(n, x) == kernel(oppolygamma, [n, x])
diBesselJ(x, y) == kernel(opBesselJ, [x, y])
diBesselY(x, y) == kernel(opBesselY, [x, y])
diBesselI(x, y) == kernel(opBesselI, [x, y])
diBesselK(x, y) == kernel(opBesselK, [x, y])
dAiryAi x == kernel(opAiryAi, x)
dAiryBi x == kernel(opAiryBi, x)

-- Could put more conditional special rules for other functions here

if R has abs : R -> R then
  iabs x ==
    (r := retractIfCan(x)@Union(Fraction Polynomial R, "failed"))
    case "failed" => iabs x
    f := r::Fraction Polynomial R
    (a := retractIfCan(numer f)@Union(R, "failed")) case "failed" or
(b := retractIfCan(denom f)@Union(R,"failed")) case "failed" => iabs x
abs(a::R)::F / abs(b::R)::F

else iabs x == iabs x

if R has SpecialFunctionCategory then

iiGamma x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => iGamma x
Gamma(r::R)::F

iiBeta l ==
(r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
=> iiBeta(first l, second l)
Beta(r::R, s::R)::F

iidigamma x ==
(r:=retractIfCan(x)@Union(R,"failed")) case "failed" => idigamma x
digamma(r::R)::F

iiPolygamma l ==
(s:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(r:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
=> iiPolygamma(first l, second l)
polygamma(s::R, r::R)::F

iiBesselJ l ==
(r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
=> iiiBesselJ(first l, second l)
besselJ(r::R, s::R)::F

iiBesselY l ==
(r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
=> iiiBesselY(first l, second l)
besselY(r::R, s::R)::F

iiBesselI l ==
(r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
=> iiiBesselI(first l, second l)
besselI(r::R, s::R)::F

iiBesselK l ==
(r:=retractIfCan(first l)@Union(R,"failed")) case "failed" or _
(s:=retractIfCan(second l)@Union(R,"failed")) case "failed" _
=> iiiBesselK(first l, second l)
besselK(r::R, s::R)::F
iiAiryAi x ==
   (r := retractIfCan(x) @ Union(R, "failed")) case "failed" => iAiryAi x
   airyAi(r::R)::F

iiAiryBi x ==
   (r := retractIfCan(x) @ Union(R, "failed")) case "failed" => iAiryBi x
   airyBi(r::R)::F

else
   if R has RetractableTo Integer then
      iiGamma x ==
         (r := retractIfCan(x) @ Union(Integer, "failed")) case Integer
            and (r::Integer >= 1) => factorial(r::Integer - 1)::F
         iGamma x
   else
      iiGamma x == iGamma x

iiBeta l == iBeta(first l, second l)
iiigamma x == idigamma x
iipolygamma l == iiipolygamma(first l, second l)
iiBesselJ l == iiIBesselJ(first l, second l)
iiBesselY l == iiIBesselY(first l, second l)
iiBesselI l == iiIBesselI(first l, second l)
iiBesselK l == iiIBesselK(first l, second l)
iiAiryAi x == iAiryAi x
iiAiryBi x == iAiryBi x

-- Default behaviour is to build a kernel
evaluate(opGamma, iiGamma)$BasicOperatorFunctions1(F)
evaluate(opabs, iiabs)$BasicOperatorFunctions1(F)
-- evaluate(opGamma2, iiGamma2)$BasicOperatorFunctions1(F)
evaluate(opBeta, iiBeta)$BasicOperatorFunctions1(F)
evaluate(opdigamma, iiigamma)$BasicOperatorFunctions1(F)
evaluate(oppolygamma, iipolygamma)$BasicOperatorFunctions1(F)
evaluate(opBesselJ, iiBesselJ)$BasicOperatorFunctions1(F)
evaluate(opBesselY, iiBesselY)$BasicOperatorFunctions1(F)
evaluate(opBesselI, iiBesselI)$BasicOperatorFunctions1(F)
evaluate(opBesselK, iiBesselK)$BasicOperatorFunctions1(F)
evaluate(opAiryAi, iiAiryAi)$BasicOperatorFunctions1(F)
evaluate(opAiryBi, iiAiryBi)$BasicOperatorFunctions1(F)

___

differentiation of special functions

In the following we define the symbolic derivatives of the special functions we provide. The formulas we use for the Bessel functions can be found in Milton Abramowitz and Irene A. Stegun, eds. (1965). Handbook of Mathematical Functions with Formulas, Graphs, and
CHAPTER 7. CHAPTER F

Up to patch-50 the formula for $K$ missed the minus sign. (Issue #355)

We do not attempt to provide formulas for the derivative with respect to the first argument currently. Instead, we leave such derivatives unevaluated.

--- package FSPECF FunctionalSpecialFunction ---

import Fraction Integer
ahalf: F := recip(2::F)::F
athird: F := recip(2::F)::F
twothirds: F := 2*recip(3::F)::F

---

We need to get hold of the differentiation operator as modified by FunctionSpace. Otherwise, for example, display will be ugly. We accomplish that by differentiating an operator, which will certainly result in a single kernel only.

--- package FSPECF FunctionalSpecialFunction ---

dummyArg: SE := new()$SE
opdiff := operator first kernels D((operator(new()$SE)$BasicOperator)(dummyArg::F), dummyArg)

---

The differentiation operator opdiff takes three arguments corresponding to

$$F_{i}(a_1, a_2, \ldots, a_n):$$

1. $F(a_1, \ldots, dm, \ldots a_n)$, where the $i^{th}$ argument is a dummy variable,
2. $dm$, the dummy variable, and
3. $a_i$, the point at which the differential is evaluated.

In the following, it seems to be safe to use the same dummy variable throughout. At least, this is done also in FunctionSpace, and did not cause problems.

The operation symbolicGrad returns the first component of the gradient of op l.

--- package FSPECF FunctionalSpecialFunction ---

dm := new()$SE :: F
iBesselJ(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t) * kernel(opdiff, [opBesselJ [dm, x], dm, n])
+ differentiate(x, t) * ahalf * (besselJ (n-1,x) - besselJ (n+1,x))

iBesselY(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t) * kernel(opdiff, [opBesselY [dm, x], dm, n])
+ differentiate(x, t) * ahalf * (besselY (n-1,x) - besselY (n+1,x))

iBesselI(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t) * kernel(opdiff, [opBesselI [dm, x], dm, n])
+ differentiate(x, t) * ahalf * (besselI (n-1,x) + besselI (n+1,x))

iBesselK(l: List F, t: SE): F ==
n := first l; x := second l
differentiate(n, t) * kernel(opdiff, [opBesselK [dm, x], dm, n])
- differentiate(x, t) * ahalf * (besselK (n-1,x) + besselK (n+1,x))

For the moment we throw an error if we try to differentiate polygamma with respect to the first argument.

— package FSPECF FunctionalSpecialFunction —
ipolygamma(l: List F, x: SE): F ==
member?(x, variables first l) =>
  error "cannot differentiate polygamma with respect to the first argument"
n := first l; y := second l
differentiate(y, x) * polygamma(n+1, y)

iBetaGrad1(l: List F): F ==
x := first l; y := second l
Beta(x,y) *(digamma x - digamma(x+y))

iBetaGrad2(l: List F): F ==
x := first l; y := second l
Beta(x,y) *(digamma y - digamma(x+y))

if F has ElementaryFunctionCategory then
  iGamma2(l: List F, t: SE): F ==
a := first l; x := second l
differentiate(a, t) * kernel(opdiff, [opGamma2 [dm, x], dm, a])
- differentiate(x, t) * x ** (a - 1) * exp(-x)
  setProperty(opGamma2, SPECIALDIFF, iGamma2@((List F, SE)->F)
  pretend None)

Finally, we tell Axiom to use these functions for differentiation. Note that up to patch-50, the properties for the Bessel functions were set using derivative(oppolygamma, [lzero,
ipolygammaGrad]), where 0zero returned zero always. Trying to replace 0zero by a function
that returns the first component of the gradient failed, it resulted in an infinite loop for
integrate(D(besselJ(a,x),a),a).

— package FSPECF FunctionalSpecialFunction —


derivative(opabs, (x:F):F +-> abs(x) * inv(x))
derivative(opGamma, (x:F):F +-> digamma x * Gamma x)
derivative(opBeta, [iBetaGrad1, iBetaGrad2])
derivative(opdigamma, (x:F):F +-> polygamma1, x))
setProperty(oppolygamma, SPECIALDIFF, ipolygamma((List F, SE)->F)
pretend None)
setProperty(opBesselJ, SPECIALDIFF, iBesselJ((List F, SE)->F)
pretend None)
setProperty(opBesselY, SPECIALDIFF, iBesselY((List F, SE)->F)
pretend None)
setProperty(opBesselI, SPECIALDIFF, iBesselI((List F, SE)->F)
pretend None)
setProperty(opBesselK, SPECIALDIFF, iBesselK((List F, SE)->F)
pretend None)

— FSPECF.dotabb —

"FSPECF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSPECF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSPECF" -> "FS"

— package FFCAT2 FunctionFieldCategoryFunctions2 —

)set break resume
)sys rm -f FunctionFieldCategoryFunctions2.output
)spool FunctionFieldCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionFieldCategoryFunctions2
--R
--R FunctionFieldCategoryFunctions2(R1: UniqueFactorizationDomain,UP1: UnivariatePolynomialCategory(R1),UPUP1: UnivariatePolynomialCategory(Fraction(UP2)),F2: FunctionFieldCategory(R2,UP2,UPUP2)) is a package constructor
--R Abbreviation for FunctionFieldCategoryFunctions2 is FFCAT2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FFCAT2
--R
--R------------------------------- Operations --------------------------------
--R map : ((R1 -> R2),F1) -> F2
--R
--E 1

)spool
)lisp (bye)

---------

| FunctionFieldCategoryFunctions2.help |

| FunctionFieldCategoryFunctions2 examples |

Lifts a map from rings to function fields over them.

See Also:
| o )show FunctionFieldCategoryFunctions2 |

---------

FunctionFieldCategoryFunctions2 (FFCAT2)

Exports:
map
— package FFCAT2 FunctionFieldCategoryFunctions2 —

)abbrev package FFCAT2 FunctionFieldCategoryFunctions2  
++ Author: Manuel Bronstein  
++ Date Created: May 1988  
++ Date Last Updated: 26 Jul 1988  
++ Description:  
++ Lifts a map from rings to function fields over them.

FunctionFieldCategoryFunctions2(R1, UP1, UPUP1, F1, R2, UP2, UPUP2, F2):  
Exports == Implementation where  
R1 : UniqueFactorizationDomain  
UP1 : UnivariatePolynomialCategory R1  
UPUP1: UnivariatePolynomialCategory Fraction UP1  
F1 : FunctionFieldCategory(R1, UP1, UPUP1)  
R2 : UniqueFactorizationDomain  
UP2 : UnivariatePolynomialCategory R2  
UPUP2: UnivariatePolynomialCategory Fraction UP2  
F2 : FunctionFieldCategory(R2, UP2, UPUP2)  
Exports ==> with  
  map: (R1 -> R2, F1) -> F2  
  ++ map(f, p) lifts f to F1 and applies it to p.

Implementation ==> add  
map(f, f1) ==  
  reduce(map(f, lift f1)$MultipleMap(R1, UP1, UPUP1, R2, UP2, UPUP2))

— FFCAT2.dotabb —

"FFCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FFCAT2"]  
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]  
"FFCAT2" -> "FFCAT"

package FFINTBAS FunctionFieldIntegralBasis

— FunctionFieldIntegralBasis.input —

)set break resume
Integral bases for function fields of dimension one
In this package \( R \) is a Euclidean domain and \( F \) is a framed algebra over \( R \). The package provides functions to compute the integral closure of \( R \) in the quotient field of \( F \). It is assumed that \( \text{char}(R/P) = \text{char}(R) \) for any prime \( P \) of \( R \). A typical instance of this is when \( R = K[x] \) and \( F \) is a function field over \( R \).

See Also:
- \)>show FunctionFieldIntegralBasis
FunctionFieldIntegralBasis (FFINTBAS)

Exports:
integralBasis  localIntegralBasis

— package FFINTBAS FunctionFieldIntegralBasis —

)abbrev package FFINTBAS FunctionFieldIntegralBasis
++ Author: Victor Miller
++ Date Created: 9 April 1990
++ Date Last Updated: 20 September 1994
++ Description:
++ Integral bases for function fields of dimension one
++ In this package R is a Euclidean domain and F is a framed algebra
++ over R. The package provides functions to compute the integral
++ closure of R in the quotient field of F. It is assumed that
++ \spad{char(R/P) = char(R)} for any prime P of R. A typical instance of
++ this is when \spad{R = K[x]} and F is a function field over R.

FunctionFieldIntegralBasis(R,UP,F): Exports == Implementation where
R : EuclideanDomain with
  squareFree: $ -> Factored $
    ++ \spad{squareFree(x)} returns a square-free factorisation of x
UP : UnivariatePolynomialCategory R
F : FramedAlgebra(R,UP)

I  ==> Integer
Mat ==> Matrix R
NNI ==> NonNegativeInteger

Exports == with
  integralBasis : () -> Record(basis: Mat, basisDen: R, basisInv:Mat)
    ++ \spad{integralBasis()} returns a record
    ++ \spad{[basis,basisDen,basisInv]} containing information regarding
    ++ the integral closure of R in the quotient field of F, where
    ++ F is a framed algebra with R-module basis \spad{\{w1,w2,...,wn\}}.
    ++ If \spad{basis} is the matrix \spad{\{(aij, i = 1..n, j = 1..n)\}, then

++ the \spad{i}th element of the integral basis is
++ \spad{vi} = \frac{1}{basisDen} \times \sum aij * wj, j = 1..n}, i.e. the
++ \spad{i}th row of \spad{basis} contains the coordinates of the
++ \spad{i}th basis vector. Similarly, the \spad{i}th row of the
++ matrix \spad{basisInv} contains the coordinates of \spad{vi} with
++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
++ matrix \spad{bij, i = 1..n, j = 1..n)}, then
++ \spad{vi} = \sum \spad{bij} * \spad{vj}, j = 1..n}

localIntegralBasis : R -> Record(basis: Mat, basisDen: R, basisInv:Mat)
++ \spad{integralBasis(p)} returns a record
++ \spad{\[basis, basisDen, basisInv\]} containing information regarding
++ the local integral closure of \spad{R} at the prime \spad{p} in the quotient
++ field of \spad{F}, where \spad{F} is a framed algebra with \spad{R}-module basis
++ \spad{\[v1, w2, ..., wn\].}
++ If \spad{basis} is the matrix \spad{\((aij, i = 1..n, j = 1..n)\), then
++ the \spad{i}th element of the local integral basis is
++ \spad{vi} = \frac{1}{basisDen} \times \sum aij * wj, j = 1..n}, i.e. the
++ \spad{i}th row of \spad{basis} contains the coordinates of the
++ \spad{i}th basis vector. Similarly, the \spad{i}th row of the
++ matrix \spad{basisInv} contains the coordinates of \spad{vi} with
++ respect to the basis \spad{v1,...,vn}: if \spad{basisInv} is the
++ matrix \spad{bij, i = 1..n, j = 1..n)}, then
++ \spad{vi} = \sum \spad{bij} * \spad{vj}, j = 1..n}

Implementation ==> add
import IntegralBasisTools(R, UP, F)
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

squaredFactors: R -> R
squaredFactors px ==
  */\[(if ffe.exponent > 1 then ffe.factor else 1$R)
             for ffe in factors squareFree px]

iIntegralBasis: (Mat,R,R) -> Record(basis: Mat, basisDen: R, basisInv:Mat)
iIntegralBasis(tfm,disc,sing) ==
  -- tfm = trace matrix of current order
  n := rank()$F; tfm0 := copy tfm; disc0 := disc
  rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
  -- rb = basis matrix of current order
  -- rbinv = inverse basis matrix of current order
  -- these are wrt the original basis for \spad{F}
  rbden : R := 1; index : R := 1; oldIndex : R := 1
  -- rbden = denominator for current basis matrix
  -- index = index of original order in current order
  not sizeLess?(1, sing) => [rb, rbden, rbinv]
  repeat
    -- compute the p-radical
    idinv := transpose squareTop rowEchelon(tfm, sing)
    -- \[u1, ..., un\] are the coordinates of an element of the p-radical
-- iff \([u_1, \ldots, u_n] \ast \text{idinv} \text{ is in sing} \ast R^n\)

\[
id := \text{rowEchelon LowTriBddDenomInv(idinv, sing)}
\]

\[
\text{id} = \text{basis matrix of the p-radical}
\]

\[
idinv := \text{UpTriBddDenomInv(id, sing)}
\]

\[
\text{id} \ast \text{idinv} = \text{sing} \ast \text{identity}
\]

\[
\text{no need to check for inseparability in this case}
\]

\[
rbinv := \text{idealiser(id} \ast \text{rb, rbinv} \ast \text{idinv, sing} \ast \text{rbden})
\]

\[
\text{index} := \text{diagonalProduct rbinv}
\]

\[
rb := \text{rowEchelon LowTriBddDenomInv(rbinv, rbden} \ast \text{sing)}
\]

\[
g := \text{matrixGcd(rb, sing, n)}
\]

\[
\text{if sizeLess?(1,g) then rb := (rb exquo g) :: Mat}
\]

\[
rbden := \text{rbden} \ast (\text{sing quo g})
\]

\[
rbinv := \text{UpTriBddDenomInv(rb, rbden)}
\]

\[
disc := \text{disc0 quo (index} \ast \text{index)}
\]

\[
\text{indexChange} := \text{index quo oldIndex; oldIndex := index}
\]

\[
\text{sing} := \text{gcd(indexChange, squaredFactors disc)}
\]

\[
\text{not sizeLess?(1, sing)} \Rightarrow \text{return [rb, rbden, rbinv]}
\]

\[
tfm := ((rb \ast \text{tfm0} \ast \text{transpose} \text{rb}) \text{exquo (rbden} \ast \text{rbden)}) :: \text{Mat}
\]

\[
\text{integralBasis()} =
\]

\[
n := \text{rank()}\_F; \ p := \text{characteristic()}\_F
\]

\[
\text{(not zero? p) and (n} \geq p) \Rightarrow
\]

\[
\text{error "integralBasis: possible wild ramification"}
\]

\[
\text{tfm := traceMatrix()}\_F; \ \text{disc := determinant tfm}
\]

\[
\text{sing := squaredFactors disc} \quad \text{-- singularities of relative Spec}
\]

\[
iIntegralBasis(tfm, disc, sing)
\]

\[
\text{localIntegralBasis prime} =
\]

\[
n := \text{rank()}\_F; \ p := \text{characteristic()}\_F
\]

\[
\text{(not zero? p) and (n} \geq p) \Rightarrow
\]

\[
\text{error "integralBasis: possible wild ramification"}
\]

\[
\text{tfm := traceMatrix()}\_F; \ \text{disc := determinant tfm}
\]

\[
\text{(disc exquo (prime} \ast \text{prime} \text{)) case "failed" } \Rightarrow
\]

\[
[\text{scalarMatrix(n,1),1,scalarMatrix(n,1)}]
\]

\[
iIntegralBasis(tfm, disc, prime)
\]

\[
\text{——}
\]

— FFINTBAS.dotabb —

"FFINTBAS" [color="#FF4488",href=bookvol10.4.pdf#nameddest=FFINTBAS"

"FRAMALG" [color="#4488FF",href=bookvol10.2.pdf#nameddest=FRAMALG"

"FFINTBAS" -> "FRAMALG"

\[
\text{——}
\]
package PMASSFS FunctionSpaceAssertions

--- FunctionSpaceAssertions.input ---

)set break resume
)sys rm -f FunctionSpaceAssertions.output
)spool FunctionSpaceAssertions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceAssertions
--R
--R FunctionSpaceAssertions(R: OrderedSet,F: FunctionSpace(R)) is a package constructor
--R Abbreviation for FunctionSpaceAssertions is PMASSFS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMASSFS
--R
--R-------------------------------- Operations --------------------------------
--R assert : (F,String) -> F
--R multiple : F -> F
--R optional : F -> F
--R
--E 1

)spool
)lisp (bye)

---

--- FunctionSpaceAssertions.help ---

====================================================================
FunctionSpaceAssertions examples
====================================================================

Attaching assertions to symbols for pattern matching;

See Also:
o )show FunctionSpaceAssertions

---
FunctionSpaceAssertions (PMASSFS)

Exports:
assert constant multiple optional

— package PMASSFS FunctionSpaceAssertions —

)abbrev package PMASSFS FunctionSpaceAssertions
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching assertions to symbols for pattern matching;

FunctionSpaceAssertions(R, F): Exports == Implementation where
  R: OrderedSet
  F: FunctionSpace R

K ==> Kernel F
PمونOPT ==> "%pmoptional"
PمونMULT ==> "%pmmultiple"
PمونCUNST ==> "%pmconstant"

Exports ==> with
  assert : (F, String) -> F
  ++ assert(x, s) makes the assertion s about x.
  ++ Error: if x is not a symbol.
  constant: F -> F
  ++ constant(x) tells the pattern matcher that x should
  ++ match only the symbol 'x and no other quantity.
  ++ Error: if x is not a symbol.
  optional: F -> F
  ++ optional(x) tells the pattern matcher that x can match
  ++ an identity (0 in a sum, 1 in a product or exponentiation).
  ++ Error: if x is not a symbol.
  multiple: F -> F
  ++ multiple(x) tells the pattern matcher that x should
preferably match a multi-term quantity in a sum or product. For matching on lists, multiple(x) tells the pattern matcher that x should match a list instead of an element of a list. Error: if x is not a symbol.

Implementation ==> add

\[
\begin{align*}
\text{ass} &: (K, \text{String}) \rightarrow \mathbb{F} \\
\text{asst} &: (K, \text{String}) \rightarrow \mathbb{F} \\
\text{mkk} &: \text{BasicOperator} \rightarrow \mathbb{F} \\
\text{mkk} \text{ op} &= \text{kernel}(\text{op, empty()}$\mathbb{List}(\mathbb{F})) \\
\text{ass}(x, s) &= \text{has?}(\text{op} := \text{operator} k, s) \Rightarrow k :: \mathbb{F} \\
\text{asst}(x, s) &= \text{has?}(\text{op} := \text{operator} k, s) \Rightarrow k :: \mathbb{F} \\
\text{assert}(x, s) &= \text{retractIfCan}(x)@\text{Union}(\text{Symbol, "failed"}) \text{ case Symbol} \Rightarrow \\
&\quad \text{asst}(\text{retract}(x)@K, s) \\
&\quad \text{error "assert must be applied to symbols only"} \\
\text{constant} x &= \text{retractIfCan}(x)@\text{Union}(\text{Symbol, "failed"}) \text{ case Symbol} \Rightarrow \\
&\quad \text{ass}(\text{retract}(x)@K, \text{PMCONST}) \\
&\quad \text{error "constant must be applied to symbols only"} \\
\text{optional} x &= \text{retractIfCan}(x)@\text{Union}(\text{Symbol, "failed"}) \text{ case Symbol} \Rightarrow \\
&\quad \text{ass}(\text{retract}(x)@K, \text{PMOPT}) \\
&\quad \text{error "optional must be applied to symbols only"} \\
\text{multiple} x &= \text{retractIfCan}(x)@\text{Union}(\text{Symbol, "failed"}) \text{ case Symbol} \Rightarrow \\
&\quad \text{ass}(\text{retract}(x)@K, \text{PMMULT}) \\
&\quad \text{error "multiple must be applied to symbols only"}
\end{align*}
\]

— PMASSFS.dotabb —

"PMASSFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMASSFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMASSFS" -> "FS"
package PMPREDFS FunctionSpaceAttachPredicates

--- FunctionSpaceAttachPredicates.input ---

)set break resume
)sys rm -f FunctionSpaceAttachPredicates.output
)spool FunctionSpaceAttachPredicates.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceAttachPredicates
--R
--R FunctionSpaceAttachPredicates(R: OrderedSet,F: FunctionSpace(R),D: Type) is a package constructor
--R Abbreviation for FunctionSpaceAttachPredicates is PMPREDFS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMPREDFS
--R
--R----------------------------------- Operations -----------------------------------
--R suchThat : (F,(D -> Boolean)) -> F
--R suchThat : (F,List((D -> Boolean))) -> F
--R
--E 1

)spool
)lisp (bye)

---

--- FunctionSpaceAttachPredicates.help ---

====================================================================
FunctionSpaceAttachPredicates examples
====================================================================

Attaching predicates to symbols for pattern matching.

See Also:
 o )show FunctionSpaceAttachPredicates
FunctionSpaceAttachPredicates (PMPREDFS)

Exports:
suchThat

— package PMPREDFS FunctionSpaceAttachPredicates —

)abbrev package PMPREDFS FunctionSpaceAttachPredicates
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching predicates to symbols for pattern matching.

FunctionSpaceAttachPredicates(R, F, D): Exports == Implementation where
R: OrderedSet
F: FunctionSpace R
D: Type

K ==> Kernel F
PMPRED ==> "%pmpredicate"

Exports ==> with
 suchThat: (F, D -> Boolean) -> F
   ++ suchThat(x, foo) attaches the predicate foo to x;
   ++ error if x is not a symbol.
suchThat: (F, List(D -> Boolean)) -> F
   ++ suchThat(x, [f1, f2, ..., fn]) attaches the predicate
   ++ f1 and f2 and ... and fn to x.
   ++ Error: if x is not a symbol.

Implementation ==> add
import AnyFunctions1(D -> Boolean)

st : (K, List Any) -> F
preds: K -> List Any
mkk : BasicOperator -> F
suchThat(p:F, f:D -> Boolean) == suchThat(p, [f])
mkk op == kernel(op, empty()$List(F))

preds k ==
  (u := property(operator k, PMPRED)) case "failed" => empty()
  (u::None) pretend List(Any)

st(k, l) ==
  mkk assert(setProperty(copy operator k, PMPRED,
    concat(preds k, l pretend None), string(new()$Symbol))

suchThat(p:F, l:List(D -> Boolean)) ==
  retractIfCan(p)@Union(Symbol, "failed") case Symbol =>
    st(retract(p)@K, [f::Any for f in l])
  error "suchThat must be applied to symbols only"

— PMPREDFS.dotabb —

"PMPREDFS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMPREDFS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"PMPREDFS" -> "FS"

— package FSCINT FunctionSpaceComplexIntegration —

package FSCINT FunctionSpaceComplexIntegration

— FunctionSpaceComplexIntegration.input —

)set break resume
)sys rm -f FunctionSpaceComplexIntegration.output
)spool FunctionSpaceComplexIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceComplexIntegration

--R
--R FunctionSpaceComplexIntegration(R: Join(EuclideanDomain,OrderedSet,CharacteristicZero,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),F: Join(TranscendentalFunctionCategory,AlgebraicallyClosedFunctionSpace(R))) is a package constructor
--R Abbreviation for FunctionSpaceComplexIntegration is FSCINT
--R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for FSCINT
---R
---R---------------------------------------- Operations ----------------------------------------
---R complexIntegrate : (F,Symbol) -> F
---R internalIntegrate : (F,Symbol) -> IntegrationResult(F)
---R internalIntegrate0 : (F,Symbol) -> IntegrationResult(F)
---R
---E 1

)spool
)lisp (bye)

----------

— FunctionSpaceComplexIntegration.help —

====================================================================
FunctionSpaceComplexIntegration examples
====================================================================

Top-level complex function integration FunctionSpaceComplexIntegration provides functions for the indefinite integration of complex-valued functions.

See Also:
o )show FunctionSpaceComplexIntegration

----------

FunctionSpaceComplexIntegration (FSCINT)

Exports:

complexIntegrate  internalIntegrate  internalIntegrate0
package FSCINT FunctionSpaceComplexIntegration

)abbrev package FSCINT FunctionSpaceComplexIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 11 June 1993
++ Description:
++ Top-level complex function integration
++ \spadtype{FunctionSpaceComplexIntegration} provides functions for the
++ indefinite integration of complex-valued functions.

FunctionSpaceComplexIntegration(R, F): Exports == Implementation where
R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
        RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory,
         AlgebraicallyClosedFunctionSpace R)

SE ==> Symbol
G ==> Complex R
FG ==> Expression G
IR ==> IntegrationResult F

Exports == with
  internalIntegrate : (F, SE) -> IR
  ++ internalIntegrate(f, x) returns the integral of \spad{f(x)dx}
  ++ where x is viewed as a complex variable.
  internalIntegrate0: (F, SE) -> IR
  ++ internalIntegrate0 should be a local function, but is conditional.
  complexIntegrate : (F, SE) -> F
  ++ complexIntegrate(f, x) returns the integral of \spad{f(x)dx}
  ++ where x is viewed as a complex variable.

Implementation ==> add
  import IntegrationTools(R, F)
  import ElementaryIntegration(R, F)
  import ElementaryIntegration(G, FG)
  import AlgebraicManipulations(R, F)
  import AlgebraicManipulations(G, FG)
  import TrigonometricManipulations(R, F)
  import IntegrationResultToFunction(R, F)
  import IntegrationResultFunctions2(FG, F)
  import ElementaryFunctionStructurePackage(R, F)
  import ElementaryFunctionStructurePackage(G, FG)
  import InnerTrigonometricManipulations(R, F, FG)

K2KG: Kernel F -> Kernel FG

K2KG k == retract(tan F2FG first argument k)@Kernel(FG)
complexIntegrate(f, x) ==
    removeConstantTerm(complexExpand internalIntegrate(f, x), x)

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
    and F has Join(LiouvilleanFunctionCategory, RetractableTo SE)
    then import PatternMatchIntegration(R, F)
    internalIntegrate0(f, x) ==
        intPatternMatch(f, x, lfinegrate, pmComplexintegrate)
    else internalIntegrate0(f, x) == lfinegrate(f, x)

internalIntegrate(f, x) ==
    f := distribute(f, x::F)
    any?(x1+->has?(operator x1, "rtrig"),
        [k for k in tower(g := realElementary(f, x))
        | member?(x, variables(k::F))]
    )$List(Kernel F))$List(Kernel F) =>
        h := trigs2explogs(F2FG g, [K2KG k for k in tower f
        | is?(k, "tan"::SE) or is?(k, "cot"::SE)], [x])
        real?(g := FG2F h) =>
            internalIntegrate0(rootSimp(rischNormalize(g, x).func), x)
        real?(g := FG2F(h := rootSimp(rischNormalize(h, x).func))) =>
            map(FG2F, lfinegrate(h, x))
        internalIntegrate0(rootSimp(rischNormalize(g, x).func), x)

| FSCINT.dotabb |
"FSCINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSCINT"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"FSCINT" -> "ACFS"

package FS2 FunctionSpaceFunctions2

| FunctionSpaceFunctions2.input |
)set break resume
)sys rm -f FunctionSpaceFunctions2.output
)spool FunctionSpaceFunctions2.output
)set message test on
)set message auto off
FunctionSpaceFunctions2.help

Lifting of maps to function spaces. This package allows a mapping $R \to S$ to be lifted to a mapping from a function space over $R$ to a function space over $S$.

See Also:
  o )show FunctionSpaceFunctions2
FUNCTIONSPACEFUNCTIONS2 (FS2)

Exports:
map

— package FS2 FunctionSpaceFunctions2 —

)abbrev package FS2 FunctionSpaceFunctions2
++ Author: Manuel Bronstein
++ Date Created: 22 March 1988
++ Date Last Updated: 3 May 1994
++ Description:
++ Lifting of maps to function spaces
++ This package allows a mapping R -> S to be lifted to a mapping
++ from a function space over R to a function space over S;

FunctionSpaceFunctions2(R, A, S, B): Exports == Implementation where
  R, S: Join(Ring, OrderedSet)
  A : FunctionSpace R
  B : FunctionSpace S

  K ==> Kernel A
  P ==> SparseMultivariatePolynomial(R, K)

Exports ==> with
  map: (R -> S, A) -> B
  ++ map(f, a) applies f to all the constants in R appearing in \spad{a}.

Implementation ==> add
  smpmap: (R -> S, P) -> B

  smpmap(fn, p) ==
    map(x+->map(z+->map(fn, z),x)$ExpressionSpaceFunctions2(A,B),
        y+->fn(y)::B,p)
    $PolynomialCategoryLifting(IndexedExponents K, K, R, P, B)

  if R has IntegralDomain then
if $S$ has IntegralDomain then
    $\text{map}(f, x) = \frac{\text{smmap}(f, \text{numer} x)}{\text{smmap}(f, \text{denom} x)}$
else
    $\text{map}(f, x) = \text{smmap}(f, \text{numer} x) \times (\text{recip}(\text{smmap}(f, \text{denom} x))::B)$
else
    $\text{map}(f, x) = \text{smmap}(f, \text{numer} x)$

———

package FSINT FunctionSpaceIntegration

———

FunctionSpaceIntegration.input

)set break resume
)sys rm -f FunctionSpaceIntegration.output
)spool FunctionSpaceIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceIntegration
--R
--R FunctionSpaceIntegration(R: Join(EuclideanDomain,OrderedSet,CharacteristicZero,RetractableTo)
--R Abbreviation for FunctionSpaceIntegration is FSINT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FSINT
--R
--R-------------------------------- Operations -----------------------------------
--R integrate : (F,Symbol) -> Union(F,List(F))
--R
--E 1

)spool
)lisp (bye)
---

--- FunctionSpaceIntegration.help ---

====================================================================
FunctionSpaceIntegration examples
====================================================================

Top-level real function integration FunctionSpaceIntegration provides functions for the indefinite integration of real-valued functions.

See Also:
o )show FunctionSpaceIntegration

---

FunctionSpaceIntegration (FSINT)

---

Exports:
integrate

--- package FSINT FunctionSpaceIntegration ---

)abbrev package FSINT FunctionSpaceIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 11 June 1993
++ Description:
++ Top-level real function integration
++ \spadtype{FunctionSpaceIntegration} provides functions for the
++ indefinite integration of real-valued functions.

FunctionSpaceIntegration(R, F):Exports == Implementation where
R : Join(EuclideanDomain, OrderedSet, CharacteristicZero, RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory, AlgebraicallyClosedFunctionSpace R)

B ==> Boolean
G ==> Complex R
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
SE ==> Symbol
IR ==> IntegrationResult F
FG ==> Expression G
ALGOP ==> "%alg"
TANTEMP ==> "%temptan":SE

Exports ==> with
  integrate: (F, SE) -> Union(F, List F)
  ++ integrate(f, x) returns the integral of \spad{f(x)dx}
  ++ where x is viewed as a real variable.

Implementation ==> add
  import IntegrationTools(R, F)
  import ElementaryIntegration(R, F)
  import ElementaryIntegration(G, FG)
  import AlgebraicManipulations(R, F)
  import TrigonometricManipulations(R, F)
  import IntegrationResultToFunction(R, F)
  import TranscendentalManipulations(R, F)
  import IntegrationResultFunctions2(FG, F)
  import FunctionSpaceComplexIntegration(R, F)
  import ElementaryFunctionStructurePackage(R, F)
  import InnerTrigonometricManipulations(R, F, FG)
  import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, SparseMultivariatePolynomial(R, K), F)

K2KG : K -> Kernel FG
postSubst : (F, List F, List K, B, List K, SE) -> F
rinteg : (IR, F, SE, B, B) -> Union(F, List F)
mkPrimh : (F, SE, B, B) -> F
trans? : F -> B
goComplex? : (B, List K, List K) -> B
halfangle : F -> F
Khalf : K -> F
tan2temp : K -> K

optemp:BasicOperator := operator(TANTEMP, 1)

K2KG k == retract(tan F2FG first argument k)@Kernel(FG)
tan2temp k == kernel(optemp, argument k, height k)$K
trans f ==
  any?(x1->is?(x1,"log"::SE) or is?(x1,"exp"::SE) or is?(x1,"atan"::SE),
    operators f)$List(BasicOperator)

mkPrimh(f, x, h, comp) ==
  f := real f
  if comp then f := removeSinSq f
  g := mkPrim(f, x)
  h and trans? g => htrigs g
  g

rinteg(i, f, x, h, comp) ==
  not elem? i => integral(f, x)$F
  empty? rest(l := [mkPrimh(f, x, h, comp) for f in expand i]) => first l
  l

-- replace tan(a/2)**2 by (1-cos a)/(1+cos a) if tan(a/2) is in ltan
halfangle a ==
  a := 2 * a
  (1 - cos a) / (1 + cos a)

Khalf k ==
  a := 2 * first argument k
  sin(a) / (1 + cos a)

-- ltan = list of tangents in the integrand after real normalization
postSubst(f, lv, lk, comp, ltan, x) ==
  for v in lv for k in lk repeat
    if ((u := retractIfCan(v)@Union(K, "failed")) case K) then
      if has?(operator(kk := u::K), ALGOP) then
        f := univariate(f, kk, minPoly kk) (kk::F)
        f := eval(f, [u::K], [k::F])
      if not(comp or empty? ltan) then
        ltemp := [tan2temp k for k in ltan]
        f := eval(f, ltan, [k::F for k in ltemp])
        f := eval(f, TANTEMP, 2, halfangle)
        f := eval(f, ltemp, [Khalf k for k in ltemp])
        removeConstantTerm(f, x)
  -- can handle a single unnested tangent directly, otherwise go complex for now
  -- l is the list of all the kernels containing x
  -- ltan is the list of all the tangents in l
  goComplex?(rt, l, ltan) ==
    empty? ltan => rt
    not empty? rest rest l

integrate(f, x) ==
  not real? f => complexIntegrate(f, x)
  f := distribute(f, x::F)
  tf := [k for k in tower f | member?(x,variables(k::F)@List(SE))]|$List(K)
ltf := select(x1+->is?(operator x1, "tan::SE), tf)
h := any?(x1+->has?(operator x1, "htrig"), tf)
rec := rischNormalize(realElementary(f, x), x)
g := rootSimp(rec.func)
tg := [k for k in tower g | member?(x, variables(k::F))]|$List(K)
rtg := select(x1+->is?(operator x1, "tan::SE), rtg)
el := any?(x1+->has?(operator x1, "elem"), el)
i:IR
if (comp := goComplex?(rtg, tg, ltf)) then
    i := map(FG2F, lffintegrate(trigs2explogs(F2FG g,
        [K2KG k for k in tf | is?(k, "tan::SE" or
        is?(k, "cot":SE)], [x]), x))
else i := lffintegrate(g, x)
ltg := setDifference(ltg, ltf) -- tan's added by normalization
(u := rinteg(i, f, x, el and ht, comp)) case F =>
    postSubst(u::F, rec.vals, rec.kers, comp, ltf, x)
    [postSubst(h, rec.vals, rec.kers, comp, ltg, x) for h in u::List(F)]
--R Abbreviation for FunctionSpacePrimitiveElement is FSPRMELT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FSPRMELT
--R
--R----------------------------------- Operations --------------------------------
--R primitiveElement : List(F) -> Record(primelt: F,poly: List(SparseUnivariatePolynomial(F)),prim: SparseUnivariatePolynomial(F))
--R primitiveElement : (F,F) -> Record(primelt: F,pol1: SparseUnivariatePolynomial(F),pol2: SparseUnivariatePolynomial(F),prim: SparseUnivariatePolynomial(F)) if F has ACF
--R

)spool
)lisp (bye)

---

FunctionSpacePrimitiveElement.help

FunctionsSpacePrimitiveElement provides functions to compute primitive elements in functions spaces;

See Also:
o )show FunctionSpacePrimitiveElement

---

FunctionSpacePrimitiveElement (FSPRMELT)

Exports:
primitiveElement
chapter 7. chapter F

abbrev package FSPRMELT FunctionSpacePrimitiveElement
++ Author: Manuel Bronstein
++ Date Created: 6 Jun 1990
++ Date Last Updated: 25 April 1991
++ Description:
++ FunctionsSpacePrimitiveElement provides functions to compute
++ primitive elements in functions spaces;

FunctionSpacePrimitiveElement(R, F): Exports == Implementation where
  R: Join(IntegralDomain, OrderedSet, CharacteristicZero)
  F: FunctionSpace R

SY ==> Symbol
P ==> Polynomial F
K ==> Kernel F
UP ==> SparseUnivariatePolynomial F
REC ==> Record(primelt:F, poly:List UP, prim:UP)

Exports ==
  primitiveElement: List F -> Record(primelt:F, poly:List UP, prim:UP)
  ++ primitiveElement([a1,...,an]) returns \spad{[a, [q1,...,qn], q]}
  ++ such that then \spad{k(a1,...,an) = k(a)},
  ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
  ++ This operation uses the technique of
  ++ \spad{groebner bases}\{Groebner basis\}.
  if F has AlgebraicallyClosedField then
    primitiveElement: (F,F)->Record(primelt:F,pol1:UP,pol2:UP,prim:UP)
    ++ primitiveElement(a1, a2) returns \spad{[a, q1, q2, q]}
    ++ such that \spad{k(a1, a2) = k(a)},
    ++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
    ++ The minimal polynomial for a2 may involve a1, but the
    ++ minimal polynomial for a1 may not involve a2;
    ++ This operations uses \spad{resultant}.

Implementation ==
  import PrimitiveElement(F)
  import AlgebraicManipulations(R, F)
  import PolynomialCategoryLifting(IndexedExponents K,
    K, R, SparseMultivariatePolynomial(R, K), P)

F2P: (F, List SY) -> P
K2P: (K, List SY) -> P

F2P(f, l) ==
  inv(denom(f)::F)*map((k1:K):P+->K2P(k1,l),(r1:R):P+->r1::F::P, numer f)

K2P(k, l) ==
((v := symbolIfCan k case SY) and member?(v::SY, 1) => v::SY::P
k::F::P

primitiveElement 1 ==
  u := string(uu := new()$SY)
  vars := [concat(u, string i)::SY for i in 1..#l]
  vv := [kernel(v)$K :: F for v in vars]
  kers := [retract(a)@K for a in l]
  pols := [F2P(subst(ratDenom((minPoly k) v, kers), kers, vv), vars)
            for k in kers for v in vv]
  rec := primitiveElement(pols, vars, uu)
  [+/[c * a for c in rec.coef for a in l], rec.poly, rec.prim]

if F has AlgebraicallyClosedField then
  import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, SparseMultivariatePolynomial(R, K), F)

F2UP: (UP, K, UP) -> UP
getpoly: (UP, F) -> UP

F2UP(p, k, q) ==
  ans:UP := 0
  while not zero? p repeat
    f := univariate(leadingCoefficient p, k)
    ans := ans + ((numer f) q)
    * monomial(inv(retract(denom f)@F), degree p)
    p := reductum p
  ans

primitiveElement(a1, a2) ==
  a := (aa := new()$SY)::F
  b := (bb := new()$SY)::F
  l := [aa, bb]$List(SY)
  p1 := minPoly(k1 := retract(a1)@K)
  p2 := map((z1:F):F+->subst(ratDenom(z1, [k1]), [k1], [a]),
            minPoly(retract(a2)@K))
  rec := primitiveElement(F2P(p1 a, l), aa, F2P(p2 b, l), bb)
  w := rec.coef1 * a1 + rec.coef2 * a2
  g := rootOf(rec.prim)
  zero?(rec.coef1) =>
    c2g := inv(rec.coef2 :: F) * g
    r := gcd(p1, univariate(p2 c2g, retract(a)@K, p1))
    q := getpoly(r, g)
    [w, q, rec.coef2 * monomial(1, 1)$UP, rec.prim]
  ic1 := inv(rec.coef1 :: F)
  gg := (ic1 * g)::UP - monomial(rec.coef2 * ic1, 1)$UP
  kg := retract(g)@K
  r := gcd(p1 gg, F2UP(p2, retract(a)@K, gg))
  q := getpoly(r, g)
  [w, monomial(ic1, 1)$UP - rec.coef2 * ic1 * q, q, rec.prim]
getpoly(r, g) ==
-- one? degree r =>
(degree r = 1) =>
  k := retract(g)@K
  univariate(-coefficient(r,0)/leadingCoefficient r,k,minPoly k)
error "GCD not of degree 1"

— FSPRMELT.dotabb —
"FSPRMELT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FSPRMELT"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FSPRMELT" -> "FS"

package FSRED FunctionSpaceReduce

— FunctionSpaceReduce.input —

)set break resume
)sys rm -f FunctionSpaceReduce.output
)spool FunctionSpaceReduce.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceReduce
--R
--R FunctionSpaceReduce(R: Join(OrderedSet,IntegralDomain,RetractableTo(Integer)),F: FunctionSpace(R)) is a package constructor
--R Abbreviation for FunctionSpaceReduce is FSRED
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FSRED
--R
--R----------------------------------- Operations -----------------------------------
--R bringDown : F -> Fraction(Integer)  newReduc : () -> Void
--R bringDown : (F,Kernel(F)) -> SparseUnivariatePolynomial(Fraction(Integer))
--R
--E 1

)spool
)lisp (bye)
Reduction from a function space to the rational numbers
This package provides function which replaces transcendental kernels
in a function space by random integers. The correspondence between
the kernels and the integers is fixed between calls to new().

See Also:
o )show FunctionSpaceReduce
++ the kernels and the integers is fixed between calls to new().

FunctionSpaceReduce(R, F): Exports == Implementation where
R: Join(OrderedSet, IntegralDomain, RetractableTo Integer)
F: FunctionSpace R
Z ==> Integer
Q ==> Fraction Integer
UP ==> SparseUnivariatePolynomial Q
K ==> Kernel F
ALGOP ==> "%alg"

Exports ==> with
bringDown: F -> Q
++ bringDown(f) \undocumented
bringDown: (F, K) -> UP
++ bringDown(f,k) \undocumented
newReduc : () -> Void
++ newReduc() \undocumented

Implementation ==> add
import SparseUnivariatePolynomialFunctions2(F, Q)
import PolynomialCategoryQuotientFunctions(IndexedExponents K,
K, R, SparseMultivariatePolynomial(R, K), F)

K2Z : K -> F
redmap := table()$AssociationList(K, Z)

newReduc() ==
  for k in keys redmap repeat remove_!(k, redmap)
  void

bringDown(f, k) ==
  ff := univariate(f, k)
  (bc := extendedEuclidean(map(bringDown, denom ff),
    m := map(bringDown, minPoly k), 1)) case "failed" =>
      error "denominator is 0"
    (map(bringDown, numer ff) * bc.coef1) rem m

bringDown f ==
  retract(eval(f, lk := kernels f, [K2Z k for k in lk])@Q

K2Z k ==
  has?(operator k, ALGOP) => error "Cannot reduce constant field"
  (u := search(k, redmap)) case "failed" =>
    setelt(redmap, k, random()$Z)::F
  u::Z::F
package SUMFS FunctionSpaceSum

--- FunctionSpaceSum.input ---

)set break resume
)sys rm -f FunctionSpaceSum.output
)spool FunctionSpaceSum.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceSum

--R FunctionSpaceSum(R: Join(IntegralDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),F: Join(FunctionSpace(R),CombinatorialOpsCategory,AlgebraicallyClosedField,TranscendentalFunctionCategory)) is a package constructor
--R Abbreviation for FunctionSpaceSum is SUMFS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SUMFS

--R Operations

--R sum : (F,Symbol) -> F
--R sum : (F,SegmentBinding(F)) -> F

)spool
)lisp (bye)

--- FunctionSpaceSum.help ---

====================================================================
FunctionSpaceSum examples
====================================================================
Computes sums of top-level expressions

See Also:
  o )show FunctionSpaceSum

---

FunctionSpaceSum (SUMFS)

Exports:
  sum

— package SUMFS FunctionSpaceSum —

)abbrev package SUMFS FunctionSpaceSum
++ Author: Manuel Bronstein
++ Date Last Updated: 19 April 1991
++ Description:
  ++ Computes sums of top-level expressions

FunctionSpaceSum(R, F): Exports == Implementation where
  R: Join(IntegralDomain, OrderedSet,
            RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F: Join(FunctionSpace R, CombinatorialOpsCategory,
            AlgebraicallyClosedField, TranscendentalFunctionCategory)

SE ==> Symbol
K ==> Kernel F

Exports ==> with
  sum: (F, SE) -> F
  ++ sum(a(n), n) returns A(n) such that A(n+1) - A(n) = a(n);
sum: (F, SegmentBinding F) -> F
++ sum(f(n), n = a..b) returns f(a) + f(a+1) + ... + f(b);

Implementation ==> add
import ElementaryFunctionStructurePackage(R, F)
import GosperSummationMethod(IndexedExponents K, K, R,
SparseMultivariatePolynomial(R, K), F)

innersum: (F, K) -> Union(F, "failed")
notRF? : (F, K) -> Boolean
newk : () -> K

newk() == kernel(new()$SE)

sum(x:F, s:SegmentBinding F) ==
  k := kernel(variable s)$K
  (u := innersum(x, k)) case "failed" => summation(x, s)
  eval(u::F, k, 1 + hi segment s) - eval(u::F, k, lo segment s)

sum(x:F, v:SE) ==
  (u := innersum(x, kernel(v)$K)) case "failed" => summation(x,v)
  u::F

notRF?(f, k) ==
  for kk in tower f repeat
    member?(k, tower(kk::F)) and (symbolIfCan(kk) case "failed") =>
      return true
    false

innersum(x, k) ==
  zero? x => 0
  notRF?(f := normalize(x / (x1 := eval(x, k, k::F - 1))), k) =>
    "failed"
  (u := GospersMethod(f, k, newk)) case "failed" => "failed"
  x1 * eval(u::F, k, k::F - 1)
package FS2EXPXP FunctionSpaceToExponentialExpansion

--- FunctionSpaceToExponentialExpansion.input ---

)set break resume
)sys rm -f FunctionSpaceToExponentialExpansion.output
)spool FunctionSpaceToExponentialExpansion.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show FunctionSpaceToExponentialExpansion
--R
--R FunctionSpaceToExponentialExpansion(R: Join(GcdDomain,OrderedSet,RetractableTo(Integer),Line)
--R Abbreviation for FunctionSpaceToExponentialExpansion is FS2EXPXP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FS2EXPXP
--R
--R------------------------------ Operations -------------------------------
--R localAbs : FE -> FE
--R exprToXXP : (FE,Boolean) -> Union(Expansion: ExponentialExpansion(R,FE,x,cen),%problem: Record(func: String,prob: String))
--E 1

)spool
)lisp (bye)

---

--- FunctionSpaceToExponentialExpansion.help ---

====================================================================
FunctionSpaceToExponentialExpansion examples
====================================================================

This package converts expressions in some function space to exponential expansions.

See Also:
  o )show FunctionSpaceToExponentialExpansion

---
FunctionSpaceToExponentialExpansion (FS2EXPXP)

Exports:
  localAbs  exprToXXP

— package FS2EXPXP FunctionSpaceToExponentialExpansion —

)abbrev package FS2EXPXP FunctionSpaceToExponentialExpansion
++ Author: Clifton J. Williamson
++ Date Created: 17 August 1992
++ Date Last Updated: 2 December 1994
++ Description:
  This package converts expressions in some function space to exponential
  expansions.

FunctionSpaceToExponentialExpansion(R,FE,x,cen):=
  Exports == Implementation where
  R : Join(GcdDomain,OrderedSet,RetractableTo Integer,_
    LinearlyExplicitRingOver Integer)
  FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
    FunctionSpace R)
  x  : Symbol
  cen : FE
  B  ==> Boolean
  BOP ==> BasicOperator
  Expon ==> Fraction Integer
  I  ==> Integer
  NNI ==> NonNegativeInteger
  K  ==> Kernel FE
  L  ==> List
  RN ==> Fraction Integer
  S  ==> String
  SY  ==> Symbol
  PCL ==> PolynomialCategoryLifting(IndexedExponents K,K,R,SMP,FE)
  POL ==> Polynomial R
  SMP ==> SparseMultivariatePolynomial(R,K)
  SUP ==> SparseUnivariatePolynomial Polynomial Polynomial R
UTS ==> UnivariateTaylorSeries(FE,x,cen)
ULS ==> UnivariateLaurentSeries(FE,x,cen)
UPXS ==> UnivariatePuiseuxSeries(FE,x,cen)
EFULS ==> ElementaryFunctionsUnivariateLaurentSeries(FE,UTS,ULS)
EFUPXS ==> ElementaryFunctionsUnivariatePuiseuxSeries(FE,ULS,UPXS,EFULS)
FS2UPS ==> FunctionSpaceToUnivariatePowerSeries(R,FE,RN,UPXS,EFUPXS,x)
EXPUPXS ==> ExponentialOfUnivariatePuiseuxSeries(FE,x,cen)
UPXSSING ==> UnivariatePuiseuxSeriesWithExponentialSingularity(R,FE,x,cen)
XXP ==> ExponentialExpansion(R,FE,x,cen)
Problem ==> Record(func:String,prob:String)
Result ==> Union(%series:UPXS,%problem:Problem)
XResult ==> Union(%expansion:XXP,%problem:Problem)
SIGNEF ==> ElementaryFunctionSign(R,FE)

Exports ==> with
  exprToXXP : (FE,B) -> XResult
  ++ exprToXXP(fcn,posCheck?) converts the expression \spad{fcn} to
  ++ an exponential expansion. If \spad{posCheck?} is true,
  ++ log’s of negative numbers are not allowed nor are nth roots of
  ++ negative numbers with n even. If \spad{posCheck?} is false,
  ++ these are allowed.
  localAbs: FE -> FE
  ++ localAbs(fcn) = \spad{abs(fcn)} or \spad{sqrt(fcn**2)} depending
  ++ on whether or not FE has a function \spad{abs}. This should be
  ++ a local function, but the compiler won’t allow it.

Implementation ==> add

import FS2UPS -- conversion of functional expressions to Puiseux series
import EFUPXS -- partial transcendental functions on UPXS

ratIfCan : FE -> Union(RN,"failed")
stateSeriesProblem : (S,S) -> Result
stateProblem : (S,S) -> XResult
newElem : FE -> FE
smpElem : SMP -> FE
k2Elem : K -> FE
iExprToXXP : (FE,B) -> XResult
listToXXP : (L FE,B,XXP,(XXP,XXP) -> XXP) -> XResult
isNonTrivPower : FE -> Union(Record(val:FE,exponent:I),"failed")
negativePowerOK? : UPXS -> Boolean
powerToXXP : (FE,I,B) -> XResult
carefulNthRootIfCan : (UPXS,NNI,B) -> Result
nthRootXXPfCan : (XXP,NNI,B) -> XResult
nthRootToXXP : (FE,NNI,B) -> XResult
genPowerToXXP : (L FE,B) -> XResult
kernelToXXP : (K,B) -> XResult
genExp : (UPXS,B) -> Result
exponential : (UPXS,B) -> XResult
expToXXP : (FE,B) -> XResult
genLog : (UPXS,B) -> Result
logToXXP : (FE,B) -> XResult
applyIfCan : (UPXS -> Union(UPXS,"failed"),FE,S,B) -> XResult
applyBddIfCan : (FE,UPXS -> Union(UPXS,"failed"),FE,S,B) -> XResult
tranToXXP : (K,FE,B) -> XResult
contOnReals? : S -> B
bddOnReals? : S -> B
opsInvolvingX : FE -> L BOP
opInOpList? : (SY,L BOP) -> B
exponential? : FE -> B
productOfNonZeros? : FE -> B
atancotToXXP : (FE,FE,B,I) -> XResult

ZEROCOUNT : RN := 1000/1
-- number of zeroes to be removed when taking logs or nth roots

--% retractions
ratIfCan fcn == retractIfCan(fcn)@Union(RN,"failed")

--% 'problems' with conversion

stateSeriesProblem(function,problem) ==
  -- records the problem which occurred in converting an expression
  -- to a power series
  [[function,problem]]

stateProblem(function,problem) ==
  -- records the problem which occurred in converting an expression
  -- to an exponential expansion
  [[function,problem]]

--% normalizations

newElem f ==
  -- rewrites a functional expression; all trig functions are
  -- expressed in terms of sin and cos; all hyperbolic trig
  -- functions are expressed in terms of exp; all inverse
  -- hyperbolic trig functions are expressed in terms of exp
  -- and log
  smpElem(numer f) / smpElem(denom f)

smpElem p == map(k2Elem,(x1:R):FE+->x1::FE,p)$PCL

k2Elem k ==
  -- rewrites a kernel; all trig functions are
  -- expressed in terms of sin and cos; all hyperbolic trig
  -- functions are expressed in terms of exp
  null(args := [newElem a for a in argument k]) => k :: FE
  iez := inv(ez := exp(z := first args))
\[ \sin z := \sin z; \cos z := \cos z \]
\[ \text{is}(k, \text{"tan" :: SY}) \Rightarrow \sin z / \cos z \]
\[ \text{is}(k, \text{"cot" :: SY}) \Rightarrow \cos z / \sin z \]
\[ \text{is}(k, \text{"sec" :: SY}) \Rightarrow \text{inv } \cos z \]
\[ \text{is}(k, \text{"csc" :: SY}) \Rightarrow \text{inv } \sin z \]
\[ \text{is}(k, \text{"sinh" :: SY}) \Rightarrow (e^z - i e^z) / (2 :: \text{FE}) \]
\[ \text{is}(k, \text{"cosh" :: SY}) \Rightarrow (e^z + i e^z) / (2 :: \text{FE}) \]
\[ \text{is}(k, \text{"tanh" :: SY}) \Rightarrow (e^z - i e^z) / (e^z + i e^z) \]
\[ \text{is}(k, \text{"coth" :: SY}) \Rightarrow (e^z + i e^z) / (e^z - i e^z) \]
\[ \text{is}(k, \text{"sech" :: SY}) \Rightarrow 2 * \text{inv } (e^z + i e^z) \]
\[ \text{is}(k, \text{"csch" :: SY}) \Rightarrow 2 * \text{inv } (e^z - i e^z) \]
\[ \text{is}(k, \text{"acosh" :: SY}) \Rightarrow \text{log } (\sqrt{z^2 - 1} + z) \]
\[ \text{is}(k, \text{"atanh" :: SY}) \Rightarrow \text{log } ((z + 1) / (1 - z)) / (2 :: \text{FE}) \]
\[ \text{is}(k, \text{"acoth" :: SY}) \Rightarrow \text{log } ((1 + \text{inv } z) + \text{sqrt}(\text{inv } (z^2 - 1))) \]
\[ \text{is}(k, \text{"asech" :: SY}) \Rightarrow \text{log } ((\text{inv } z) + \text{sqrt}(1 + \text{inv } (z^2))) \]
\[ \text{is}(k, \text{"acsch" :: SY}) \Rightarrow \text{log } ((\text{inv } z) + \text{sqrt}(1 + \text{inv } (z^2))) \]

(\text{operator } k) \text{ args}

```
-- % general conversion function
exprToXXP(fcn,posCheck?) == iExprToXXP(newElem fcn,posCheck?)

iExprToXXP(fcn,posCheck?) ==
  -- converts a functional expression to an exponential expansion
  -- The following line is commented out so that expressions of
  -- the form a**b will be normalized to exp(b * log(a)) even if
  -- 'a' and 'b' do not involve the limiting variable 'x'.
  --
  -- not member?(x,variables fcn) => \{\text{monomial}(fcn,0) \UPXS :: XXP\}
  (poly := retractIfCan(fcn)@Union(POL,\"failed\")) case POL =>
    [exprToUPS(fcn,false,\"real:two sides\").%series :: XXP]
  (sum := isPlus fcn) case L(FE) =>
    listToXXP(sum::L(FE),posCheck?,0,(y1:XXP,y2:XXP):XXP +-> y1+y2)
  (prod := isTimes fcn) case L(FE) =>
    listToXXP(prod :: L(FE),posCheck?,1,(y1:XXP,y2:XXP):XXP +-> y1*y2)
  (expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
    power := expt :: Record(val:FE,exponent:1)
    powerToXXP(power.val,power.exponent,posCheck?)
  (ker := retractIfCan(fcn)@Union(K,\"failed\")) case K =>
    kernelToXXP(ker :: K,posCheck?)
  error \"exprToXXP: neither a sum, product, power, nor kernel\"

-- % sums and products
listToXXP(list,posCheck?,ans,op) ==
  -- converts each element of a list of expressions to an exponential
  -- expansion and returns the sum of these expansions, when 'op' is +
  -- and 'ans' is 0, or the product of these expansions, when 'op' is *
  -- and 'ans' is 1
  while not null list repeat
(term := iExprToXXP(first list,posCheck?)) case %problem =>
    return term
ans := op(ans,term.%expansion)
list := rest list
[ans]

-- % nth roots and integral powers

isNonTrivPower fcn ==
    -- is the function a power with exponent other than 0 or 1?
    (expt := isPower fcn) case "failed" => "failed"
    power := expt :: Record(val:FE,exponent:I)
    -- one? power.exponent => "failed"
    (power.exponent = 1) => "failed"
    power

negativePowerOK? upxs ==
    -- checks the lower order coefficient of a Puiseux series;
    -- the coefficient may be inverted only if
    -- (a) the only function involving x is 'log', or
    -- (b) the lowest order coefficient is a product of exponentials
    -- and functions not involving x
    deg := degree upxs
    if (coef := coefficient(upxs,deg)) = 0 then
        deg := order(upxs,deg + ZEROCOUNT :: Expon)
        (coef := coefficient(upxs,deg)) = 0 =>
            error "inverse of series with many leading zero coefficients"
    xOpList := opsInvolvingX coef
    -- only function involving x is 'log'
    (null xOpList) => true
    (null rest xOpList and is?(first xOpList,"log" :: SY)) => true
    -- lowest order coefficient is a product of exponentials and
    -- functions not involving x
    productOfNonZeros? coef => true
    false

powerToXXP(fcn,n,posCheck?) ==
    -- converts an integral power to an exponential expansion
    (b := iExprToXXP(fcn,posCheck?)) case %problem => b
    xxp := b.%expansion
    n > 0 => [xxp ** n]
    -- a Puiseux series will be reciprocated only if n < 0 and
    -- numerator of 'xxp' has exactly one monomial
    numberOfMonomials(num := numer xxp) > 1 => [xxp ** n]
    negativePowerOK? leadingCoefficient num =>
        (rec := recip num) case "failed" => error "FS2EXPXP: can't happen"
        nn := (-n) :: NNI
        [[((denom xxp) ** nn) * ((rec :: UPXSSING) ** nn)) :: XXP]
        --!! we may want to create a fraction instead of trying to
        --!! reciprocate the numerator
stateProblem("inv","lowest order coefficient involves x")

carefulNthRootIfCan(ups,n,posCheck?) ==
  -- similar to 'nthRootIfCan', but it is fussy about the series
  -- it takes as an argument. If 'n' is EVEN and 'posCheck '?'
  -- is true then the leading coefficient of the series must
  -- be POSITIVE. In this case, if 'rightOnly?' is false, the
  -- order of the series must be zero. The idea is that the
  -- series represents a real function of a real variable, and
  -- we want a unique real nth root defined on a neighborhood
  -- of zero.
  n < 1 => error "nthRoot: n must be positive"
  deg := degree ups
  if (coef := coefficient(ups,deg)) = 0 then
    deg := order(ups,deg + ZEROCOUNT :: Expon)
    (coef := coefficient(ups,deg)) = 0 =>
      error "log of series with many leading zero coefficients"
  -- if 'posCheck?' is true, we do not allow nth roots of negative
  -- numbers when n in even
  if even?(n :: I) then
    if posCheck? and ((signum := sign(coef)$SIGNEF) case I) then
      (signum :: I) = -1 =>
        return stateSeriesProblem("nth root","root of negative number")
    (ans := nthRootIfCan(ups,n)) case "failed" =>
      stateSeriesProblem("nth root","no nth root")
  [ans :: UPXS]

nthRootXXPIfCan(xxp,n,posCheck?) ==
  num := numer xxp; den := denom xxp
  not zero?(reductum num) or not zero?(reductum den) =>
    stateProblem("nth root","several monomials in numerator or denominator")
  nInv : RN := 1/n
  newNum :=
    coef : UPXS :=
      root := carefulNthRootIfCan(leadingCoefficient num,n,posCheck?)
      root case %problem => return [root.%problem]
      root.%series
      deg := (nInv :: FE) * (degree num)
      monomial(coef,deg)
  newDen :=
    coef : UPXS :=
      root := carefulNthRootIfCan(leadingCoefficient den,n,posCheck?)
      root case %problem => return [root.%problem]
      root.%series
      deg := (nInv :: FE) * (degree den)
      monomial(coef,deg)
  [newNum/newDen]

nthRootToXXP(arg,n,posCheck?) ==
  -- converts an nth root to a power series
-- this is not used in the limit package, so the series may
-- have non-zero order, in which case nth roots may not be unique
(result := iExprToXXP(arg,posCheck?)) case %problem => [result,%problem]
ans := nthRootXXPIfCan(result,%expansion,n,posCheck?)
ans case %problem => [ans,%problem]
[ans,%expansion]

--% general powers f(x) ** g(x)

genPowerToXXP(args,posCheck?) ==
-- converts a power f(x) ** g(x) to an exponential expansion
(logBase := logToXXP(first args,posCheck?)) case %problem =>
  logBase
(expon := iExprToXXP(second args,posCheck?)) case %problem =>
  expon
xxp := (expon,%expansion) * (logBase,%expansion)
(f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
  stateProblem("exp","multiply nested exponential")
exponential(f,posCheck?)

--% kernels

kernelToXXP(ker,posCheck?) ==
-- converts a kernel to a power series
(sym := symbolIfCan(ker)) case Symbol =>
  (sym :: Symbol) = x => [monomial(1,1)$UPXS :: XXP]
  [monomial(ker :: FE,0)$UPXS :: XXP]
empty?(args := argument ker) => [monomial(ker :: FE,0)$UPXS :: XXP]
empty? rest args =>
  arg := first args
  is?(ker,"%paren" :: Symbol) => iExprToXXP(arg,posCheck?)
  is?(ker,"log" :: Symbol) => logToXXP(arg,posCheck?)
  is?(ker,"exp" :: Symbol) => expToXXP(arg,posCheck?)
  tranToXXP(ker,arg,posCheck?)
  is?(ker,"%power" :: Symbol) => genPowerToXXP(args,posCheck?)
  is?(ker,"nthRoot" :: Symbol) =>
    n := retract(second args)@I
    nthRootToXXP(first args,n :: NNI,posCheck?)
    stateProblem(string name ker,"unknown kernel")

--% exponentials and logarithms

genExp(ups,posCheck?) ==
-- If the series has order zero and the constant term a0 of the
-- series involves x, the function tries to expand exp(a0) as
-- a power series.
(deg := order(ups,1)) < 0 =>
  -- this "can’t happen"
  error "exp of function with singularity"
deg > 0 => [exp(ups)]
lc := coefficient(ups,0); varOpList := opsInvolvingX lc
not opInOpList?("log" :: Symbol,varOpList) => [exp(ups)]
-- try to fix exp(lc) if necessary
expCoef := normalize(exp lc,x)$ElementaryFunctionStructurePackage(R,FE)
result := exprToGenUPS(expCoef,posCheck?,"real:right side")$FS2UPS
---! will deal with problems in limitPlus in EXPEXPAN
--result case %problem => result
result case %problem => [exp(ups)]
[(result.%series) * exp(ups - monomial(lc,0))]

exponential(f,posCheck?) ==
singPart := truncate(f,0) - (coefficient(f,0) :: UPXS)
taylorPart := f - singPart
expon := exponential(singPart)$EXPUPXS
(coef := genExp(taylorPart,posCheck?)) case %problem => [coef.%problem]
[monomial(coef.%series,expon)$UPXSSING :: XXP]
expoToXXP(arg,posCheck?) ==
(result := iExprToXXP(arg,posCheck?)) case %problem => result
xsp := result.%expansion
(f := retractIfCan(xsp)@Union(UPXS,"failed")) case "failed" =>
stateProblem("exp","multiply nested exponential")
exponential(f,posCheck?)
genLog(ups,posCheck?) ==
deg := degree ups
if (coef := coefficient(ups,deg)) = 0 then
deg := order(ups,deg + ZEROCOUNT)
(coef := coefficient(ups,deg)) = 0 =>
error "log of series with many leading zero coefficients"
-- if 'posCheck?' is true, we do not allow logs of negative numbers
if posCheck? then
if ((signum := sign(coef)$SIGNEF) case I) then
(signum :: I) = -1 =>
return stateSeriesProblem("log","negative leading coefficient")
lit := monomial(coef,deg)$UPXS
-- check to see if lowest order coefficient is a negative rational
negRat? : Boolean :=
((rat := ratIfCan coef case RN) case RN) =>
(rat :: RN) < 0 => true
false
false
logTerm : FE :=
mon : FE := (x :: FE) - (cen :: FE)
pow : FE := mon ** (deg :: FE)
negRat? => log(coef * pow)
term1 : FE := (deg :: FE) * log(mon)
log(coef) + term1
[monomial(logTerm,0)$UPXS + log(ups/lit)]
logToXXP(arg,posCheck?) ==
  (result := iExprToXXP(arg,posCheck?)) case %problem => result
  xxp := result.%expansion
  num := numer xxp; den := denom xxp
  not zero?(reductum num) or not zero?(reductum den) =>
    stateProblem("log","several monomials in numerator or denominator")
  numCoefLog : UPXS :=
    (res := genLog(leadingCoefficient num,posCheck?)) case %problem =>
      return [res.%problem]
    res.%series
  denCoefLog : UPXS :=
    (res := genLog(leadingCoefficient den,posCheck?)) case %problem =>
      return [res.%problem]
    res.%series
  numLog := (exponent degree num) + numCoefLog
  denLog := (exponent degree den) + denCoefLog --?? num?
  [(numLog - denLog) :: XXP]

applyIfCan(fcn,arg,fcnName,posCheck?) ==
  -- converts fcn(arg) to an exponential expansion
  (xxpArg := iExprToXXP(arg,posCheck?)) case %problem => xxpArg
  xxp := xxpArg.%expansion
  (f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
    stateProblem(fcnName,"multiply nested exponential")
  upxs := f :: UPXS
  (deg := order(upxs,1)) < 0 =>
    stateProblem(fcnName,"essential singularity")
  deg > 0 => [fcn(upxs) :: UPXS :: XXP]
  lc := coefficient(upxs,0); xOpList := opsInvolvingX lc
  null xOpList => [fcn(upxs) :: UPXS :: XXP]
  opInOpList?("log" :: SY,xOpList) =>
    stateProblem(fcnName,"logs in constant coefficient")
  contOnReals? fcnName => [fcn(upxs) :: UPXS :: XXP]
  stateProblem(fcnName,"x in constant coefficient")

applyBddIfCan(fe,fcn,arg,fcnName,posCheck?) ==
  -- converts fcn(arg) to a generalized power series, where the
  -- function fcn is bounded for real values
  -- if fcn(arg) has an essential singularity as a complex
  -- function, we return fcn(arg) as a monomial of degree 0
  (xxpArg := iExprToXXP(arg,posCheck?)) case %problem =>
    trouble := xxpArg.%problem
    trouble.prob = "essential singularity" => [monomial(fe,0)$UPXS :: XXP]
  xxpArg
  xxp := xxpArg.%expansion
  (f := retractIfCan(xxp)@Union(UPXS,"failed")) case "failed" =>
    stateProblem("exp","multiply nested exponential")
  (ans := fcn(f :: UPXS)) case "failed" => [monomial(fe,0)$UPXS :: XXP]
[ans :: UPXS :: XXP]

CONTFCNS : L S := ["sin","cos","atan","acot","exp","asinh"]
-- functions which are defined and continuous at all real numbers

BDDFCNS : L S := ["sin","cos","atan","acot"]
-- functions which are bounded on the reals

contOnReals? fcn == member?(fcn,CONTFCNS)
bddOnReals? fcn == member?(fcn,BDDFCNS)

opsInvolvingX fcn ==
  opList := [op for k in tower fcn | unary?(op := operator k) _
  and member?(x,variables first argument k)]
  removeDuplicates opList

opInOpList?(name,opList) ==
  for op in opList repeat
    is?(op,name) => return true
    false

exponential? fcn ==
  -- is 'fcn' of the form exp(f)?
  (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
    is?(ker :: K,"exp" :: Symbol)
    false

productOfNonZeroes? fcn ==
  -- is 'fcn' a product of non-zero terms, where 'non-zero'
  -- means an exponential or a function not involving x
  exponential? fcn => true
  (prod := isTimes fcn) case "failed" => false
  for term in (prod :: L(FE)) repeat
    (not exponential? term) and member?(x,variables term) =>
      return false
  true

tranToXXP(ker,arg,posCheck?) ==
  -- converts op(arg) to a power series for certain functions
  -- op in trig or hyperbolic trig categories
  -- N.B. when this function is called, 'k2elem' will have been
  -- applied, so the following functions cannot appear:
  -- tan, cot, sec, csc, sinh, cosh, tanh, coth, sech, csch
  -- acosh, atanh, acoth, asech, acsch
  is?(ker,"sin" :: SY) =>
    applyBddIfCan(ker :: FE,sinIfCan,arg,"sin",posCheck?)
  is?(ker,"cos" :: SY) =>
    applyBddIfCan(ker :: FE,cosIfCan,arg,"cos",posCheck?)
  is?(ker,"asin" :: SY) =>
    applyIfCan(asinIfCan,arg,"asin",posCheck?)
is?(ker,"acos" :: SY) =>
  applyIfCan(acosIfCan, arg, "acos", posCheck?)
is?(ker,"atan" :: SY) =>
  atancotToXXP(ker :: FE, arg, posCheck?, 1)
is?(ker,"acot" :: SY) =>
  atancotToXXP(ker :: FE, arg, posCheck?, -1)
is?(ker,"asec" :: SY) =>
  applyIfCan(asecIfCan, arg, "asec", posCheck?)
is?(ker,"acsc" :: SY) =>
  applyIfCan(acscIfCan, arg, "acsc", posCheck?)
is?(ker,"asinh" :: SY) =>
  applyIfCan(asinhIfCan, arg, "asinh", posCheck?)
stateProblem(string name ker,"unknown kernel")

if FE has abs: FE -> FE then
  localAbs fcn == abs fcn
else
  localAbs fcn == sqrt(fcn * fcn)

sign0fExpression: FE -> FE
sign0fExpression arg == localAbs(arg)/arg

atancotToXXP(fe, arg, posCheck?, plusMinus) ==
  -- converts atan(f(x)) to a generalized power series
  atanFlag : String := "real: right side"; posCheck? : Boolean := true
  (result := exprToGenUPS(arg, posCheck?, atanFlag)$FS2UPS) case %problem =>
  trouble := result.%problem
  trouble.prob = "essential singularity" => [monomial(fe, 0)$UPXS :: XXP]
  ups := result.%series; coef := coefficient(ups, 0)
  -- series involves complex numbers
  (ord := order(ups, 0)) = 0 and coef * coef = -1 =>
  y := differentiate(ups)/(1 + ups*ups)
  yCoef := coefficient(y, -1)
  [(monomial(log yCoef, 0) + integrate(y - monomial(yCoef, -1)$UPXS)) :: XXP]
  cc : FE :=
  ord < 0 =>
    (rn := ratIfCan(ord :: FE)) case "failed" =>
      -- this condition usually won't occur because exponents will
      -- be integers or rational numbers
      return stateProblem("atan","branch problem")
  lc := coefficient(ups, ord)
  (signum := sign(lc)$SIGNEF) case "failed" =>
    -- can't determine sign
    posNegPi2 := sign0fExpression(lc) * pi()/(2 :: FE)
    plusMinus = 1 => posNegPi2
    pi()/(2 :: FE) - posNegPi2
  (n := signum :: Integer) = -1 =>
    plusMinus = 1 => -pi()/(2 :: FE)
    pi()
plusMinus = 1 => pi()/(2 :: FE)  
0  
atan coef  
[((cc :: UPXS) + integrate(differentiate(ups)/(1 + ups*ups))) :: XXP]

-- FS2EXXPX.dotabb --

"FS2EXXP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FS2EXXP"]  
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]  
"FS2EXXP" -> "ULSCCAT"

package FS2UPS FunctionSpaceToUnivariatePowerSeries

--- FunctionSpaceToUnivariatePowerSeries.input ---

)set break resume  
)sys rm -f FunctionSpaceToUnivariatePowerSeries.output  
)spool FunctionSpaceToUnivariatePowerSeries.output  
)set message test on  
)set message auto off  
)clear all  
--S 1 of 1  
)show FunctionSpaceToUnivariatePowerSeries  
--R  
--R FunctionSpaceToUnivariatePowerSeries(R: Join(GcdDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),FE,Expon: OrderedRing,UPS,TRAN: PartialTranscendentalFunctions(UPS),x: Symbol)where  
--R FE: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(R))with  
--R coerce : Expon -> %  
--R UPS: Join(UnivariatePowerSeriesCategory(FE,Expon),Field,TranscendentalFunctionCategory)with  
--R differentiate : % -> %  
--R integrate : % -> % is a package constructor  
--R Abbreviation for FunctionSpaceToUnivariatePowerSeries is FS2UPS  
--R This constructor is not exposed in this frame.  
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FS2UPS  
--R  
--R------------------------------- Operations ------------------------------  
--R localAbs : FE -> FE  
--R exprToGenUPS : (FE,Boolean,String) -> Union(%series: UPS,%problem: Record(func: String,prob: String))  
--R exprToUPS : (FE,Boolean,String) -> Union(%series: UPS,%problem: Record(func: String,prob: String))  
--R
FunctionSpaceToUnivariatePowerSeries (FS2UPS)

Exports:
   localAbs  exprToGenUPS  exprToUPS

--- package FS2UPS FunctionSpaceToUnivariatePowerSeries ---

)abbrev package FS2UPS FunctionSpaceToUnivariatePowerSeries
This package converts expressions in some function space to power
series in a variable x with coefficients in that function space.
The function `exprToUPS` converts expressions to power series
whose coefficients do not contain the variable x. The function
`exprToGenUPS` converts functional expressions to power series
whose coefficients may involve functions of \( \log(x) \).

```spad
definition FunctionSpaceToUnivariatePowerSeries(R, FE, Expon, UPS, TRAN, x):
   Exports == Implementation where
        R : Join(GcdDomain, OrderedSet, RetractableTo Integer, _
            LinearlyExplicitRingOver Integer)
        FE : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory, _
            FunctionSpace R)
            with
                coerce: Expon -> %
                    ++ coerce(e) converts an 'exponent' e to an 'expression'
        Expon : OrderedRing
        UPS : Join(UnivariatePowerSeriesCategory(FE, Expon), Field, _
            TranscendentalFunctionCategory)
            with
                differentiate: % -> %
                    ++ differentiate(x) returns the derivative of x since we
                    ++ need to be able to differentiate a power series
                integrate: % -> %
                    ++ integrate(x) returns the integral of x since
                    ++ we need to be able to integrate a power series
        TRAN : PartialTranscendentalFunctions UPS
        x : Symbol
        B == Boolean
        BOP == BasicOperator
        I == Integer
        NNI == NonNegativeInteger
        K == Kernel FE
        L == List
        RN == Fraction Integer
        S == String
        SY == Symbol
        PCL == PolynomialCategoryLifting(IndexedExponents K, K, R, SMP, FE)
        POL == Polynomial R
        SMP == SparseMultivariatePolynomial(R, K)
        SUP == SparseUnivariatePolynomial Polynomial R
        Problem == Record(func: String, prob: String)
        Result == Union(%series: UPS, %problem: Problem)
        SIGNEF == ElementaryFunctionSign(R, FE)

    Exports == with
```
exprToUPS : (FE,B,S) -> Result
  ++ exprToUPS(fcn,posCheck?,atanFlag) converts the expression
  ++ \spad{fcn} to a power series. If \spad{posCheck?} is true,
  ++ log’s of negative numbers are not allowed nor are nth roots of
  ++ negative numbers with n even. If \spad{posCheck?} is false,
  ++ these are allowed. \spad{atanFlag} determines how the case
  ++ \spad{atan(f(x))}, where \spad{f(x)} has a pole, will be treated.
  ++ The possible values of \spad{atanFlag} are \spad{"complex"},
  ++ \spad{"real: two sides"}, \spad{"real: left side"},
  ++ \spad{"real: right side"}, and \spad{"just do it"}.
  ++ If \spad{atanFlag} is \spad{"complex"}, then no series expansion
  ++ will be computed because, viewed as a function of a complex
  ++ variable, \spad{atan(f(x))} has an essential singularity.
  ++ Otherwise, the sign of the leading coefficient of the series
  ++ expansion of \spad{f(x)} determines the constant coefficient
  ++ in the series expansion of \spad{atan(f(x))}. If this sign cannot
  ++ be determined, a series expansion is computed only when
  ++ \spad{atanFlag} is \spad{"just do it"}. When the leading term
  ++ in the series expansion of \spad{f(x)} is of odd degree (or is a
  ++ rational degree with odd numerator), then the constant coefficient
  ++ in the series expansion of \spad{atan(f(x))} for values to the
  ++ left differs from that for values to the right. If \spad{atanFlag}
  ++ is \spad{"real: two sides"}, no series expansion will be computed.
  ++ If \spad{atanFlag} is \spad{"real: left side"} the constant
  ++ coefficient for values to the left will be used and if \spad{atanFlag}
  ++ \spad{"real: right side"} the constant coefficient for values to the
  ++ right will be used.
  ++ If there is a problem in converting the function to a power series,
  ++ a record containing the name of the function that caused the problem
  ++ and a brief description of the problem is returned.
  ++ When expanding the expression into a series it is assumed that
  ++ the series is centered at 0. For a series centered at a, the
  ++ user should perform the substitution \spad{x -> x + a} before calling
  ++ this function.

exprToGenUPS : (FE,B,S) -> Result
  ++ exprToGenUPS(fcn,posCheck?,atanFlag) converts the expression
  ++ \spad{fcn} to a generalized power series. If \spad{posCheck?}
  ++ is true, log’s of negative numbers are not allowed nor are nth roots
  ++ of negative numbers with n even. If \spad{posCheck?} is false,
  ++ these are allowed. \spad{atanFlag} determines how the case
  ++ \spad{atan(f(x))}, where \spad{f(x)} has a pole, will be treated.
  ++ The possible values of \spad{atanFlag} are \spad{"complex"},
  ++ \spad{"real: two sides"}, \spad{"real: left side"},
  ++ \spad{"real: right side"}, and \spad{"just do it"}.
  ++ If \spad{atanFlag} is \spad{"complex"}, then no series expansion
  ++ will be computed because, viewed as a function of a complex
  ++ variable, \spad{atan(f(x))} has an essential singularity.
  ++ Otherwise, the sign of the leading coefficient of the series
  ++ expansion of \spad{f(x)} determines the constant coefficient
++ in the series expansion of \spad{atan(f(x))}. If this sign cannot
++ be determined, a series expansion is computed only when
++ \spad{atanFlag} is \spad{"just do it"}. When the leading term
++ in the series expansion of \spad{f(x)} is of odd degree (or is a
++ rational degree with odd numerator), then the constant coefficient
++ in the series expansion of \spad{atan(f(x))} for values to the
++ left differs from that for values to the right. If \spad{atanFlag}
++ is \spad{"real: two sides"}, no series expansion will be computed.
++ If \spad{atanFlag} is \spad{"real: left side"} the constant
++ coefficient for values to the left will be used and if \spad{atanFlag}
++ \spad{"real: right side"} the constant coefficient for values to the
++ right will be used.
++ If there is a problem in converting the function to a power
++ series, we return a record containing the name of the function
++ that caused the problem and a brief description of the problem.
++ When expanding the expression into a series it is assumed that
++ the series is centered at 0. For a series centered at a, the
++ user should perform the substitution \spad{x -> x + a} before calling
++ this function.

localAbs: FE -> FE
++ localAbs(fcn) = \spad{abs(fcn)} or \spad{sqrt(fcn**2)} depending
++ on whether or not FE has a function \spad{abs}. This should be
++ a local function, but the compiler won't allow it.

Implementation ==> add

ratIfCan : FE -> Union(RN,"failed")
carefulNthRootIfCan : (UPS,NNI,B,B) -> Result
stateProblem : (S,S) -> Result
polyToUPS : SUP -> UPS
listToUPS : (L FE,(FE,B,S) -> Result,B,S,UPS,(UPS,UPS) -> UPS)_
            -> Result
isNonTrivPower : FE -> Union(Record(val:FE,exponent:I),"failed")
powerToUPS : (FE,I,B,S) -> Result
kernelToUPS : (K,B,S) -> Result
nthRootToUPS : (FE,NNI,B,S) -> Result
logToUPS : (FE,B,S) -> Result
atancotToUPS : (FE,B,S,I) -> Result
applyIfCan : (UPS -> Union(UPS,"failed"),FE,S,B) -> Result
tranToUPS : (K,FE,B,S) -> Result
powToUPS : (L FE,B,S) -> Result
newElem : FE -> FE
smpElem : SMP -> FE
k2Elem : K -> FE
contOnReals? : S -> B
bddOnReals? : S -> B
iExprToGenUPS : (FE,B,S) -> Result
opsInvolvingX : FE -> L BOP
opInOpList? : (SY,L BOP) -> B
exponential? : FE -> B
productOfNonZeroes? : FE -> B
powerToGenUPS : (FE,I,B,S) -> Result
kernelToGenUPS : (K,B,S) -> Result
nthRootToGenUPS : (FE,NNI,B,S) -> Result
logToGenUPS : (FE,B,S) -> Result
expToGenUPS : (FE,B,S) -> Result
expGenUPS : (UPS,B,S) -> Result
atancotToGenUPS : (FE,FE,B,S,I) -> Result
genUPSApplyIfCan : (UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
applyBddIfCan : (FE,UPS -> Union(UPS,"failed"),FE,S,B,S) -> Result
tranToGenUPS : (K,FE,B,S) -> Result
powToGenUPS : (L FE,B,S) -> Result

ZEROCOUNT : I := 1000
-- number of zeroes to be removed when taking logs or nth roots

ratIfCan fcn == retractIfCan(fcn)@Union(RN,"failed")

carefulNthRootIfCan(ups,n,posCheck?,rightOnly?) ==
-- similar to 'nthRootIfCan', but it is fussy about the series
-- it takes as an argument. If 'n' is EVEN and 'posCheck?'
-- is true then the leading coefficient of the series must
-- be POSITIVE. In this case, if 'rightOnly?' is false, the
-- order of the series must be zero. The idea is that the
-- series represents a real function of a real variable, and
-- we want a unique real nth root defined on a neighborhood
-- of zero.

n < 1 => error "nthRoot: n must be positive"
deg := degree ups
if (coef := coefficient(ups,deg)) = 0 then
  deg := order(ups,deg + ZEROCOUNT :: Expon)
  (coef := coefficient(ups,deg)) = 0 =>
  error "log of series with many leading zero coefficients"
-- if 'posCheck?' is true, we do not allow nth roots of negative
-- numbers when n in even
if even?(n :: I) then
  if posCheck? and ((signum := sign(coef)$SIGNEF) case I) then
    (signum :: I) = -1 =>
    return stateProblem("nth root","negative leading coefficient")
    not rightOnly? and not zero? deg => -- nth root not unique
    return stateProblem("nth root","series of non-zero order")
  end if
  (ans := nthRootIfCan(ups,n)) case "failed" =>
  stateProblem("nth root","no nth root")
[ans :: UPS]

stateProblem(function,problem) ==
-- records the problem which occurred in converting an expression
-- to a power series
[[[function,problem]]]
exprToUPS(fcn,posCheck?,atanFlag) ==
  -- converts a functional expression to a power series
  -- The following line is commented out so that expressions of
  -- the form a**b will be normalized to exp(b * log(a)) even if
  -- 'a' and 'b' do not involve the limiting variable 'x'.
  --!
  -- not member?(x,variables fcn) => [monomial(fcn,0)]
  (poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
    [polyToUPS univariate(poly :: POL,x)]
  (sum := isPlus fcn) case L(FE) =>
    listToUPS(sum :: L(FE),exprToUPS,posCheck?,atanFlag,0,
      (y1,y2) +-> y1 + y2)
  (prod := isTimes fcn) case L(FE) =>
    listToUPS(prod :: L(FE),exprToUPS,posCheck?,atanFlag,1,
      (y1,y2) +-> y1 * y2)
  (expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
    power := expt :: Record(val:FE,exponent:I)
    powerToUPS(power.val,power.exponent,posCheck?,atanFlag)
  (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
    kernelToUPS(ker :: K,posCheck?,atanFlag)
  error "exprToUPS: neither a sum, product, power, nor kernel"

polyToUPS poly ==
  -- converts a polynomial to a power series
  zero? poly => 0
  -- we don't start with 'ans := 0' as this may lead to an
  -- enormous number of leading zeroes in the power series
  deg := degree poly
  coef := leadingCoefficient(poly) :: FE
  ans := monomial(coef,deg :: Expon)$UPS
  poly := reductum poly
  while not zero? poly repeat
    deg := degree poly
    coef := leadingCoefficient(poly) :: FE
    ans := ans + monomial(coef,deg :: Expon)$UPS
    poly := reductum poly
  ans

listToUPS(list,feToUPS,posCheck?,atanFlag,ans,op) ==
  -- converts each element of a list of expressions to a power
  -- series and returns the sum of these series, when 'op' is +
  -- and 'ans' is 0, or the product of these series, when 'op' is *
  -- and 'ans' is 1
  while not null list repeat
    (term := feToUPS(first list,posCheck?,atanFlag)) case %problem =>
      return term
    ans := op(ans,term,%series)
    list := rest list
  [ans]
isNonTrivPower fcn ==
  -- is the function a power with exponent other than 0 or 1?
  (expt := isPower fcn) case "failed" => "failed"
  power := expt :: Record(val:FE,exponent:II)
  -- one? power.exponent => "failed"
  (power.exponent = 1) => "failed"
  power

powerToUPS(fcn,n,posCheck?,atanFlag) ==
  -- converts an integral power to a power series
  (b := exprToUPS(fcn,posCheck?,atanFlag)) case %problem => b
  n > 0 => [(b.%series) ** n]
  -- check lowest order coefficient when n < 0
  ups := b.%series; deg := degree ups
  if (coef := coefficient(ups,deg)) = 0 then
    error "inverse of series with many leading zero coefficients"
  [ups ** n]

kernelToUPS(ker,posCheck?,atanFlag) ==
  -- converts a kernel to a power series
  (sym := symbolIfCan(ker)) case Symbol =>
    (sym :: Symbol) = x => [monomial(1,1)]
    [monomial(ker :: FE,0)]
  empty?(args := argument ker) => [monomial(ker :: FE,0)]
  not member?(x, variables(ker :: FE)) => [monomial(ker :: FE,0)]
  empty? rest args =>
    arg := first args
    is?(ker,"abs" :: Symbol) =>
      nthRootToUPS(arg*arg,2,posCheck?,atanFlag)
    is?(ker,"%paren" :: Symbol) => exprToUPS(arg,posCheck?,atanFlag)
    is?(ker,"log" :: Symbol) => logToUPS(arg,posCheck?,atanFlag)
    is?(ker,"exp" :: Symbol) =>
      applyIfCan(expIfCan,arg,"exp",posCheck?,atanFlag)
    tranToUPS(ker,arg,posCheck?,atanFlag)
    is?(ker,"%power" :: Symbol) => powToUPS(args,posCheck?,atanFlag)
    is?(ker,"nthRoot" :: Symbol) =>
      n := retract(second args)@II
      nthRootToUPS(first args,n :: NNI,posCheck?,atanFlag)
    stateProblem(string name ker,"unknown kernel")

nthRootToUPS(arg,n,posCheck?,atanFlag) ==
  -- converts an nth root to a power series
  -- this is not used in the limit package, so the series may
  -- have non-zero order, in which case nth roots may not be unique
  (result := exprToUPS(arg,posCheck?,atanFlag)) case %problem => result
  ans := carefulNthRootIfCan(result.%series,n,posCheck?,false)
  ans case %problem => ans
  [ans.%series]
logToUPS(arg, posCheck?, atanFlag) ==
  -- converts a logarithm log(f(x)) to a power series
  -- f(x) must have order 0 and if 'posCheck?' is true,
  -- then f(x) must have a non-negative leading coefficient
  (result := exprToUPS(arg, posCheck?, atanFlag)) case %problem => result
  ups := result.%series
  not zero? order(ups,1) =>
    stateProblem("log","series of non-zero order")
  coef := coefficient(ups,0)
  -- if 'posCheck?' is true, we do not allow logs of negative numbers
  if posCheck? then
    if (signum := sign(coef)$SIGNEF) case I then
      (signum : I) = -1 =>
        return stateProblem("log","negative leading coefficient")
    logIfCan(ups) :: UPS
  if FE has abs: FE -> FE then
    localAbs fcn == abs fcn
  else
    localAbs fcn == sqrt(fcn * fcn)
  signOfExpression: FE -> FE
  signOfExpression arg == localAbs(arg)/arg

atanToUPS(arg, posCheck?, atanFlag, plusMinus) ==
  -- converts atan(f(x)) to a power series
  (result := exprToUPS(arg, posCheck?, atanFlag)) case %problem => result
  ups := result.%series; coef := coefficient(ups,0)
  (ord := order(ups,0)) = 0 and coef * coef = -1 =>
    -- series involves complex numbers
    return stateProblem("atan","logarithmic singularity")
  cc : FE :=
    ord < 0 =>
      atanFlag = "complex" =>
        return stateProblem("atan","essential singularity")
      (rn := ratIfCan(ord :: FE)) case "failed" =>
        -- this condition usually won't occur because exponents will
        -- be integers or rational numbers
        return stateProblem("atan","branch problem")
    if (atanFlag = "real: two sides") and (odd? numer(rn :: RN)) then
      -- expansions to the left and right of zero have different
      -- constant coefficients
      return stateProblem("atan","branch problem")
    lc := coefficient(ups,ord)
    (signum := sign(lc)$SIGNEF) case "failed" =>
      -- can't determine sign
      atanFlag = "just do it" =>
        plusMinus = 1 => pi()/(2 :: FE)
    0
posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
plusMinus = 1 => posNegPi2
pi()/(2 :: FE) - posNegPi2
--return stateProblem("atan", "branch problem")
left? : B := atanFlag = "real: left side"; n := signum :: Integer
(left? and n = 1) or (not left? and n = -1) =>
plusMinus = 1 => -pi()/(2 :: FE)
pi()
plusMinus = 1 => pi()/(2 :: FE)
0
atan coef
[(cc :: UPS) + integrate(plusMinus * differentiate(ups)/(1 + ups*ups))]

applyIfCan(fcn, arg, fcnName, posCheck?, atanFlag) ==
-- converts fcn(arg) to a power series
(ups := exprToUPS(arg, posCheck?, atanFlag)) case %problem => ups
ans := fcn(ups.%series)
ans case "failed" => stateProblem(fcnName, "essential singularity")
[ans :: UPS]

tranToUPS(ker, arg, posCheck?, atanFlag) ==
-- converts ker to a power series for certain functions
-- in trig or hyperbolic trig categories
is?(ker, "sin" :: SY) =>
applyIfCan(sinIfCan, arg, "sin", posCheck?, atanFlag)
is?(ker, "cos" :: SY) =>
applyIfCan(cosIfCan, arg, "cos", posCheck?, atanFlag)
is?(ker, "tan" :: SY) =>
applyIfCan(tanIfCan, arg, "tan", posCheck?, atanFlag)
is?(ker, "cot" :: SY) =>
applyIfCan(cotIfCan, arg, "cot", posCheck?, atanFlag)
is?(ker, "sec" :: SY) =>
applyIfCan(secIfCan, arg, "sec", posCheck?, atanFlag)
is?(ker, "csc" :: SY) =>
applyIfCan(cscIfCan, arg, "csc", posCheck?, atanFlag)
is?(ker, "asin" :: SY) =>
applyIfCan(asinIfCan, arg, "asin", posCheck?, atanFlag)
is?(ker, "acos" :: SY) =>
applyIfCan(acosIfCan, arg, "acos", posCheck?, atanFlag)
is?(ker, "atan" :: SY) => atancotToUPS(arg, posCheck?, atanFlag, 1)
is?(ker, "acot" :: SY) => atancotToUPS(arg, posCheck?, atanFlag, -1)
is?(ker, "asec" :: SY) =>
applyIfCan(asecIfCan, arg, "asec", posCheck?, atanFlag)
is?(ker, "acsc" :: SY) =>
applyIfCan(acscIfCan, arg, "acsc", posCheck?, atanFlag)
is?(ker, "sinh" :: SY) =>
applyIfCan(sinhIfCan, arg, "sinh", posCheck?, atanFlag)
is?(ker, "cosh" :: SY) =>
applyIfCan(coshIfCan, arg, "cosh", posCheck?, atanFlag)
is?(ker, "tanh" :: SY) =>
```

applyIfCan(tanhIfCan, arg, "tanh", posCheck?, atanFlag)
is?(ker, "coth" :: SY) =>
applyIfCan(cothIfCan, arg, "coth", posCheck?, atanFlag)
is?(ker, "sech" :: SY) =>
applyIfCan(sechIfCan, arg, "sech", posCheck?, atanFlag)
is?(ker, "csch" :: SY) =>
applyIfCan(cschIfCan, arg, "csch", posCheck?, atanFlag)
is?(ker, "asinh" :: SY) =>
applyIfCan(asinhIfCan, arg, "asinh", posCheck?, atanFlag)
is?(ker, "acosh" :: SY) =>
applyIfCan(acoshIfCan, arg, "acosh", posCheck?, atanFlag)
is?(ker, "atanh" :: SY) =>
applyIfCan(atanhIfCan, arg, "atanh", posCheck?, atanFlag)
is?(ker, "acoth" :: SY) =>
applyIfCan(acothIfCan, arg, "acoth", posCheck?, atanFlag)
is?(ker, "asech" :: SY) =>
applyIfCan(asechIfCan, arg, "asech", posCheck?, atanFlag)
is?(ker, "acsch" :: SY) =>
applyIfCan(acschIfCan, arg, "acsch", posCheck?, atanFlag)
stateProblem(string name ker, "unknown kernel")

powToUPS(args, posCheck?, atanFlag) ==
-- converts a power f(x) ** g(x) to a power series
(logBase := logToUPS(first args, posCheck?, atanFlag)) case %problem =>
logBase
(expon := exprToUPS(second args, posCheck?, atanFlag)) case %problem =>
expon
ans := expIfCan((expon.%series) * (logBase.%series))
ans case "failed" => stateProblem("exp", "essential singularity")
[ans :: UPS]

-- Generalized power series: power series in x, where log(x) and
-- bounded functions of x are allowed to appear in the coefficients
-- of the series. Used for evaluating REAL limits at x = 0.

newElem f ==
-- rewrites a functional expression; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
smpElem(numer f) / smpElem(denom f)

smpElem p == map(k2Elem, (x1:R):FE +-> x1::FE, p)$PCL

k2Elem k ==
-- rewrites a kernel; all trig functions are
-- expressed in terms of sin and cos; all hyperbolic trig
-- functions are expressed in terms of exp
null(args := [newElem a for a in argument k]) => k::FE
iez := inv(ez := exp(z := first args))
sinz := sin z; cosz := cos z
```
is?(k,"tan" :: Symbol) => sinz / cosz
is?(k,"cot" :: Symbol) => cosz / sinz
is?(k,"sec" :: Symbol) => inv cosz
is?(k,"csc" :: Symbol) => inv sinz
is?(k,"sinh" :: Symbol) => (ez - iez) / (2 :: FE)
is?(k,"cosh" :: Symbol) => (ez + iez) / (2 :: FE)
is?(k,"tanh" :: Symbol) => (ez - iez) / (ez + iez)
is?(k,"coth" :: Symbol) => (ez + iez) / (ez - iez)
is?(k,"sech" :: Symbol) => 2 * inv(ez + iez)
is?(k,"csch" :: Symbol) => 2 * inv(ez - iez)

(operator k) args

CONTFCNS : L S := \["sin","cos","atan","acot","exp","asinh"]
-- functions which are defined and continuous at all real numbers

BDDFCNS : L S := \["sin","cos","atan","acot"]
-- functions which are bounded on the reals

contOnReals? fcn == member?(fcn,CONTFCNS)
bddOnReals? fcn == member?(fcn,BDDFCNS)

exprToGenUPS(fcn,posCheck?,atanFlag) ==
-- converts a functional expression to a generalized power series; "generalized" means that log(x) and bounded functions -- of x are allowed to appear in the coefficients of the series

iExprToGenUPS(fcn,posCheck?,atanFlag) ==
-- converts a functional expression to a generalized power series without first normalizing the expression
--!! The following line is commented out so that expressions of the form a**b will be normalized to exp(b * log(a)) even if a and b do not involve the limiting variable 'x'.
--!! - cjw 1 Dec 94
--not member?(x,variables fcn) => [monomial(fcn,0)]
(poly := retractIfCan(fcn)@Union(POL,"failed")) case POL =>
[polyToUPS univariate(poly :: POL,x)]
(sum := isPlus fcn) case L(FE) =>
listToUPS(sum :: L(FE),iExprToGenUPS,posCheck?,atanFlag,0,
(y1,y2) +-> y1 + y2)
(prod := isTimes fcn) case L(FE) =>
listToUPS(prod :: L(FE),iExprToGenUPS,posCheck?,atanFlag,1,
(y1,y2) +-> y1 * y2)
(expt := isNonTrivPower fcn) case Record(val:FE,exponent:I) =>
power := expt :: Record(val:FE,exponent:I)
powerToGenUPS(power.val,power.exponent,posCheck?,atanFlag)
(ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
kernelToGenUPS(ker :: K,posCheck?,atanFlag)
error "exprToGenUPS: neither a sum, product, power, nor kernel"
opsInvolvingX fcn ==
    opList := [op for k in tower fcn | unary?(op := operator k) _
                and member?(x,variables first argument k)]
    removeDuplicates opList

opInOpList?(name,opList) ==
    for op in opList repeat
        is?(op,name) => return true
    false

exponential? fcn ==
    -- is 'fcn' of the form exp(f)?
    (ker := retractIfCan(fcn)@Union(K,"failed")) case K =>
        is?(ker :: K,"exp" :: Symbol)
        false

productOfNonZeroes? fcn ==
    -- is 'fcn' a product of non-zero terms, where 'non-zero'
    -- means an exponential or a function not involving x
    exponential? fcn => true
    (prod := isTimes fcn) case "failed" => false
    for term in (prod :: L(FE)) repeat
        (not exponential? term) and member?(x,variables term) =>
            return false
    true

powerToGenUPS(fcn,n,posCheck?,atanFlag) ==
    -- converts an integral power to a generalized power series
    -- if n < 0 and the lowest order coefficient of the series
    -- involves x, we are careful about inverting this coefficient
    -- the coefficient is inverted only if
    -- (a) the only function involving x is 'log', or
    -- (b) the lowest order coefficient is a product of exponentials
    -- and functions not involving x
    (b := exprToGenUPS(fcn,posCheck?,atanFlag)) case %problem => b
    n > 0 => [(b.%series) ** n]
    -- check lowest order coefficient when n < 0
    ups := b.%series; deg := degree ups
    if (coef := coefficient(ups,deg)) = 0 then
        deg := order(ups,deg + ZEROCOUNT :: Expon)
        (coef := coefficient(ups,deg)) = 0 =>
            error "inverse of series with many leading zero coefficients"
    xOpList := opsInvolvingX coef
    -- only function involving x is 'log'
    (null xOpList) => [ups ** n]
    (null rest xOpList and is?(first xOpList,"log" :: SY)) =>
        [ups ** n]
    -- lowest order coefficient is a product of exponentials and
    -- functions not involving x
    productOfNonZeroes? coef => [ups ** n]
stateProblem("inv","lowest order coefficient involves x")

kernelToGenUPS(ker,posCheck?,atanFlag) ==
  -- converts a kernel to a generalized power series
  (sym := symbolIfCan(ker)) case Symbol =>
    (sym :: Symbol) = x => [monomial(1,1)]
    [monomial(ker :: FE,0)]
  empty?(args := argument ker) => [monomial(ker :: FE,0)]
  empty? rest args =>
    arg := first args
    is?(ker,"abs" :: Symbol) =>
      nthRootToGenUPS(arg*arg,2,posCheck?,atanFlag)
    is?(ker,"%paren" :: Symbol) => iExprToGenUPS(arg,posCheck?,atanFlag)
    is?(ker,"log" :: Symbol) => logToGenUPS(arg,posCheck?,atanFlag)
    is?(ker,"exp" :: Symbol) => expToGenUPS(arg,posCheck?,atanFlag)
    tranToGenUPS(ker,arg,posCheck?,atanFlag)
    is?(ker,"%power" :: Symbol) => powToGenUPS(args,posCheck?,atanFlag)
    is?(ker,"nthRoot" :: Symbol) =>
      n := retract(second args)@I
      nthRootToGenUPS(first args,n :: NNI,posCheck?,atanFlag)
    stateProblem(string name ker,"unknown kernel")

nthRootToGenUPS(arg,n,posCheck?,atanFlag) ==
  -- convert an nth root to a power series
  -- used for computing right hand limits, so the series may have
  -- non-zero order, but may not have a negative leading coefficient
  -- when n is even
  (result := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem => result
  ans := carefulNthRootIfCan(result.%series,n,posCheck?,true)
  ans case %problem => ans [ans.%series]

logToGenUPS(arg,posCheck?,atanFlag) ==
  -- converts a logarithm \(\log(f(x))\) to a generalized power series
  (result := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem => result
  ups := result.%series; deg := degree ups
  if (coef := coefficient(ups,deg)) = 0 then
    deg := order(ups,deg + ZEROCOUNT :: Expon)
    (coef := coefficient(ups,deg)) = 0 =>
      error "log of series with many leading zero coefficients"
  -- if 'posCheck?' is true, we do not allow logs of negative numbers
  if posCheck? then
    if ((signum := sign(coef)$SIGNEF) case I) then
      (signum :: I) = -1 =>
        return stateProblem("log","negative leading coefficient")
  -- create logarithmic term, avoiding log's of negative rationals
  lt := monomial(coef,deg)$UPS; cen := center lt
  -- check to see if lowest order coefficient is a negative rational
negRat? : Boolean :=
   ((rat := ratIfCan coef) case RN) =>
      (rat :: RN) < 0 => true
   false
false
logTerm : FE :=
   mon : FE := (x :: FE) - (cen :: FE)
pow : FE := mon ** (deg :: FE)
negRat? => log(coef * pow)
term1 : FE := (deg :: FE) * log(mon)
log(coef) + term1
[monomial(logTerm,0) + log(ups/lt)]

expToGenUPS(arg,posCheck?,atanFlag) ==
   expGenUPS(ups,%series,posCheck?,atanFlag)
expGenUPS(ups,0) => [exp ups]
exGenUPS(ups,%series,posCheck?,atanFlag) ==
   -- computes the exponential of a generalized power series.
   -- If the series has order zero and the constant term a0 of the
   -- series involves x, the function tries to expand exp(a0) as
   -- a power series.
   (deg := order(ups,1)) < 0 =>
      stateProblem("exp","essential singularity")
   deg > 0 => [exp ups]
lc := coefficient(ups,0); xOpList := opsInvolvingX lc
not opInOpList?("log" :: SY,xOpList) => [exp ups]
   -- try to fix exp(lc) if necessary
   expCoef :=
      normalize(exp lc,x)$ElementaryFunctionStructurePackage(R,FE)
opInOpList?("log" :: SY,opsInvolvingX expCoef) =>
      stateProblem("exp","logs in constant coefficient")
result := exprToGenUPS(expCoef,posCheck?,atanFlag)
result case %problem => result
[(result.%series) * exp(ups - monomial(lc,0))]
atancotToGenUPS(fe,arg,posCheck?,atanFlag,plusMinus) ==
   -- converts atan(f(x)) to a generalized power series
   (result := exprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
      trouble := result.%problem
      trouble.prob = "essential singularity" => [monomial(fe,0)$UPS]
      result
ups := result.%series; coef := coefficient(ups,0)
   -- series involves complex numbers
   (ord := order(ups,0)) = 0 and coef * coef = -1 =>
      y := differentiate(ups)/(1 + ups*ups)
yCoef := coefficient(y,-1)
      [monomial(log yCoef,0) + integrate(y - monomial(yCoef,-1)$UPS)]
cc : FE :=
ord < 0 =>
atanFlag = "complex" =>
    return stateProblem("atan","essential singularity")
(rn := ratIfCan(ord :: FE)) case "failed" =>
    -- this condition usually won't occur because exponents will
    -- be integers or rational numbers
    return stateProblem("atan","branch problem")
if (atanFlag = "real: two sides") and (odd? numer(rn :: RN)) then
    -- expansions to the left and right of zero have different
    -- constant coefficients
    return stateProblem("atan","branch problem")
lc := coefficient(ups,ord)
(signum := sign(lc)$SIGNEF) case "failed" =>
    -- can't determine sign
atanFlag = "just do it" =>
    plusMinus = 1 => pi()/(2 :: FE)
    0
posNegPi2 := signOfExpression(lc) * pi()/(2 :: FE)
plusMinus = 1 => posNegPi2
pi()/(2 :: FE) - posNegPi2
--return stateProblem("atan","branch problem")
left? : B := atanFlag = "real: left side"; n := signum :: Integer
(left? and n = 1) or (not left? and n = -1) =>
    plusMinus = 1 => -pi()/(2 :: FE)
    pi()
plusMinus = 1 => pi()/(2 :: FE)
    0
atan coef
[(cc :: UPS) + integrate(differentiate(ups)/(1 + ups*ups))]

genUPSApplyIfCan(fcn,arg,fcnName,posCheck?,atanFlag) ==
    -- converts fcn(arg) to a generalized power series
    (series := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
        series
    ups := series.%series
    (deg := order(ups,1)) < 0 =>
        stateProblem(fcnName,"essential singularity")
    deg > 0 => [fcn(ups) :: UPS]
lc := coefficient(ups,0); xOpList := opsInvolvingX lc
    null xOpList => [fcn(ups) :: UPS]
opInOpList?("log" :: SY,xOpList) =>
        stateProblem(fcnName,"logs in constant coefficient")
    contOnReals? fcnName => [fcn(ups) :: UPS]
    stateProblem(fcnName,"x in constant coefficient")
applyBddIfCan(fe,fcn,arg,fcnName,posCheck?,atanFlag) ==
    -- converts fcn(arg) to a generalized power series, where the
    -- function fcn is bounded for real values
    -- if fcn(arg) has an essential singularity as a complex
CHAPTER 7. CHAPTER F

-- function, we return fcn(arg) as a monomial of degree 0
(ups := iExprToGenUPS(arg,posCheck?,atanFlag)) case %problem =>
  trouble := ups.%problem
  trouble.prob = "essential singularity" => [monomial(fe,0)$UPS]
  ups
(ans := fcn(ups.%series)) case "failed" => [monomial(fe,0)$UPS]
[ans :: UPS]

tranToGenUPS(ker,arg,posCheck?,atanFlag) ==
  -- converts op(arg) to a power series for certain functions
  -- op in trig or hyperbolic trig categories
  -- N.B. when this function is called, 'k2elem' will have been
  -- applied, so the following functions cannot appear:
  -- tan, cot, sec, csc, sinh, cosh, tanh, coth, sech, csch
  is?(ker,"sin" :: SY) =>
    applyBddIfCan(ker :: FE,sinIfCan,arg,"sin",posCheck?,atanFlag)
  is?(ker,"cos" :: SY) =>
    applyBddIfCan(ker :: FE,cosIfCan,arg,"cos",posCheck?,atanFlag)
  is?(ker,"asin" :: SY) =>
    genUPSApplyIfCan(asinIfCan,arg,"asin",posCheck?,atanFlag)
  is?(ker,"acos" :: SY) =>
    genUPSApplyIfCan(acosIfCan,arg,"acos",posCheck?,atanFlag)
  is?(ker,"atan" :: SY) =>
    atancotToGenUPS(ker :: FE,arg,posCheck?,atanFlag)
  is?(ker,"acot" :: SY) =>
    atancotToGenUPS(ker :: FE,arg,posCheck?,atanFlag)
  is?(ker,"asec" :: SY) =>
    genUPSApplyIfCan(asecIfCan,arg,"asec",posCheck?,atanFlag)
  is?(ker,"acsc" :: SY) =>
    genUPSApplyIfCan(acscIfCan,arg,"acsc",posCheck?,atanFlag)
  is?(ker,"asinh" :: SY) =>
    genUPSApplyIfCan(asinhIfCan,arg,"asinh",posCheck?,atanFlag)
  is?(ker,"acosh" :: SY) =>
    genUPSApplyIfCan(acoshIfCan,arg,"acosh",posCheck?,atanFlag)
  is?(ker,"atanh" :: SY) =>
    genUPSApplyIfCan(atanhIfCan,arg,"atanh",posCheck?,atanFlag)
  is?(ker,"acoth" :: SY) =>
    genUPSApplyIfCan(acothIfCan,arg,"acoth",posCheck?,atanFlag)
  is?(ker,"asech" :: SY) =>
    genUPSApplyIfCan(asechIfCan,arg,"asech",posCheck?,atanFlag)
  is?(ker,"acsch" :: SY) =>
    genUPSApplyIfCan(acschIfCan,arg,"acsch",posCheck?,atanFlag)
  stateProblem(string name ker,"unknown kernel")

powToGenUPS(args,posCheck?,atanFlag) ==
  -- converts a power \( f(x)^g(x) \) to a generalized power series
  (logBase := logToGenUPS(first args,posCheck?,atanFlag)) case %problem =>
    logBase
  expon := iExprToGenUPS(second args,posCheck?,atanFlag)
  expon case %problem => expon
expGenUPS((expon.%series) * (logBase.%series),posCheck?,atanFlag)

--

FS2UPS.dotabb

"FS2UPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FS2UPS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"FS2UPS" -> "ACF"
"FS2UPS" -> "FS"

package FSUPFACT FunctionSpaceUnivariatePolynomialFactor

FunctionSpaceUnivariatePolynomialFactor.input

)set break resume
)sys rm -f FunctionSpaceUnivariatePolynomialFactor.output
)spool FunctionSpaceUnivariatePolynomialFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show FunctionSpaceUnivariatePolynomialFactor
--R
--R FunctionSpaceUnivariatePolynomialFactor(R: Join(IntegralDomain,OrderedSet,RetractableTo(Integer)),F: FunctionSpace(R),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for FunctionSpaceUnivariatePolynomialFactor is FSUPFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FSUPFACT
--R
--R------------------------------- Operations --------------------------------
--R ffactor : UP -> Factored(UP)
--R UP2ifCan : UP -> Union(overq: SparseUnivariatePolynomial(Fraction(Integer)),overan: SparseUnivariatePolynomial(AlgebraicNumber),failed: Boolean)
--R anfactor : UP -> Union(Factored(SparseUnivariatePolynomial(AlgebraicNumber)),"failed") if F has RETRACT(AN)
--R qfactor : UP -> Union(Factored(SparseUnivariatePolynomial(Fraction(Integer)),"failed")
--R
--E 1

)spool
)lisp (bye)
CHAPTER 7. CHAPTER F

---

FunctionSpaceUnivariatePolynomialFactor.help —

====================================================================
FunctionSpaceUnivariatePolynomialFactor examples
====================================================================

This package is used internally by IR2F

See Also:
  o )show FunctionSpaceUnivariatePolynomialFactor

---

FunctionSpaceUnivariatePolynomialFactor (FSUPFACT)

Exports:
  anfactor ffactor qfactor UP2ifCan

---

)abbrev package FSUPFACT FunctionSpaceUnivariatePolynomialFactor
++ Author: Manuel Bronstein
++ Date Created: 12 May 1988
++ Date Last Updated: 22 September 1993
++ Description:
  ++ This package is used internally by IR2F

FunctionSpaceUnivariatePolynomialFactor(R, F, UP):
  Exports == Implementation where
R : Join(IntegralDomain, OrderedSet, RetractableTo Integer)
F : FunctionSpace R
UP: UnivariatePolynomialCategory F

Q ==> Fraction Integer
K ==> Kernel F
AN ==> AlgebraicNumber
PQ ==> SparseUnivariatePolynomial(Q, K)
PR ==> SparseMultivariatePolynomial(R, K)
UPQ ==> SparseUnivariatePolynomial Q
UPA ==> SparseUnivariatePolynomial AN
FR ==> Factored UP
FRQ ==> Factored UPQ
FRA ==> Factored UPA

Exports ==> with
  ffactor: UP -> FR
    ++ ffactor(p) tries to factor a univariate polynomial p over F
qfactor: UP -> Union(FRQ, "failed")
    ++ qfactor(p) tries to factor p over fractions of integers,
    ++ returning "failed" if it cannot
UP2ifCan: UP -> Union(overq: UPQ, overan: UPA, failed: Boolean)
    ++ UP2ifCan(x) should be local but conditional.
if F has RetractableTo AN then
  anfactor: UP -> Union(FRA, "failed")
    ++ anfactor(p) tries to factor p over algebraic numbers,
    ++ returning "failed" if it cannot

Implementation ==> add
import AlgFactor(UPA)
import RationalFactorize(UPQ)

P2QifCan : PR -> Union(PQ, "failed")
UPQ2UP : (SparseUnivariatePolynomial PQ, F) -> UP
PQ2F : (PQ, F) -> F
ffactor0 : UP -> FR

dummy := kernel(new()$Symbol)$K

if F has RetractableTo AN then
  UPAN2F: UPA -> UP
  UPQ2AN: UPQ -> UPA

UPAN2F p ==
  map(x+->x::F, p)$UnivariatePolynomialCategoryFunctions2(AN,UPA,F,UP)

UPQ2AN p ==
  map(x+->x::AN, p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,AN,UPA)

ffactor p ==
(pq := anfactor p) case FRA =>
    map(UPAN2F, pq::FRA)$FactoredFunctions2(UPA, UP)
ffactor0 p

anfactor p ==
(q := UP2ifCan p) case overq =>
    map(UPQ2AN, factor(q.overq))$FactoredFunctions2(UPQ, UPA)
q case overan => factor(q.overan)
"failed"

UP2ifCan p ==
ansq := 0$UPQ ; ansa := 0$UPA
goforq? := true
while p ^= 0 repeat
    if goforq? then
        rq := retractIfCan(leadingCoefficient p)@Union(Q, "failed")
        if rq case Q then
            ansq := ansq + monomial(rq::Q, degree p)
            ansa := ansa + monomial(rq::Q::AN, degree p)
        else
            goforq? := false
        ra := retractIfCan(leadingCoefficient p)@Union(AN, "failed")
        if ra case AN then ansa := ansa + monomial(ra::AN, degree p)
            else return [true]
    else
        ra := retractIfCan(leadingCoefficient p)@Union(AN, "failed")
        if ra case AN then ansa := ansa + monomial(ra::AN, degree p)
            else return [true]
p := reductum p
goforq? => [ansq]
[ansa]

else
UPQ2F: UPQ -> UP

UPQ2F p ==
    map(x+->x::F, p)$UnivariatePolynomialCategoryFunctions2(Q,UPQ,F,UP)

ffactor p ==
(pq := qfactor p) case FRQ =>
    map(UPQ2F, pq::FRQ)$FactoredFunctions2(UPQ, UP)
ffactor0 p

UP2ifCan p ==
ansq := 0$UPQ
while p ^= 0 repeat
    rq := retractIfCan(leadingCoefficient p)@Union(Q, "failed")
    if rq case Q then ansq := ansq + monomial(rq::Q, degree p)
        else return [true]
p := reductum p
ffactor0 p ==
smp := numer(ep := p(dummy::F))
(q := P2QifCan smp) case "failed" => p::FR
map(x+->UPQ2UP(univariate(x, dummy), denom(ep)::F), factor(q::PQ)
)\$MRationalFactorize(IndexedExponents K, K, Integer, PQ)\$FactoredFunctions2(PQ, UP)

UPQ2UP(p, d) ==
map(x+->PQ2F(x, d), p)\$UnivariatePolynomialCategoryFunctions2(PQ, SparseUnivariatePolynomial PQ, F, UP)

PQ2F(p, d) ==
map((x:K):F+->x::F, (y:Q):F+->y::F, p)_
\$PolynomialCategoryLifting(IndexedExponents K, K, Q, PQ, F) / d

qfactor p ==
(q := UP2ifCan p) case overq => factor(q.overq)
"failed"

P2QifCan p ==
and/\[retractIfCan(c::F)@Union(Q, "failed") case Q
for c in coefficients p] =>
map(x+->x::PQ, y+->retract(y::F)@Q :: PQ, p)_
\$PolynomialCategoryLifting(IndexedExponents K,K,R,PR,PQ)
"failed"
package GALFACTU GaloisGroupFactorizationUtilities

— GaloisGroupFactorizationUtilities.input —

)set break resume
)sys rm -f GaloisGroupFactorizationUtilities.output
)spool GaloisGroupFactorizationUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupFactorizationUtilities
--R
--R GaloisGroupFactorizationUtilities(R: Ring,UP: UnivariatePolynomialCategory(R),F: Join(FloatingPointSystem,RetractableTo(R),Field,TranscendentalFunctionCategory,ElementaryFunctionCategory)) is a package constructor
--R Abbreviation for GaloisGroupFactorizationUtilities is GALFACTU
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GALFACTU
--R
--R------------------------------- Operations --------------------------------
--R beauzamyBound : UP -> Integer bombieriNorm : UP -> F
--R height : UP -> F infinityNorm : UP -> F
--R length : UP -> F norm : (UP,PositiveInteger) -> F
--R quadraticNorm : UP -> F rootBound : UP -> Integer
--R singleFactorBound : UP -> Integer
--R bombieriNorm : (UP,PositiveInteger) -> F
--R singleFactorBound : (UP,NonNegativeInteger) -> Integer
--R
--E 1

)spool

947
GaloisGroupFactorizationUtilities provides functions that will be used by the factorizer.

See Also:
- )show GaloisGroupFactorizationUtilities

---

GaloisGroupFactorizationUtilities (GALFACTU)

Exports:
beauzamyBound  bombieriNorm  height  infinityNorm  length
norm  quadraticNorm  rootBound  singleFactorBound

---

)abbrev package GALFACTU GaloisGroupFactorizationUtilities
++ Author: Frederic Lehobey
++ Date Created: 30 June 1994
++ Date Last Updated: 19 October 1995
++ References:
++ [1] Bernard Beauzamy, Products of polynomials and a priori estimates for
++ coefficients in polynomial decompositions: a sharp result,
++ J. Symbolic Computation (1992) 13, 463-472
++ [2] David W. Boyd, Bounds for the Height of a Factor of a Polynomial in
++ Terms of Bombieri's Norms: I. The Largest Factor,
++ J. Symbolic Computation (1993) 16, 115-130
++ [3] David W. Boyd, Bounds for the Height of a Factor of a Polynomial in
++ Terms of Bombieri's Norms: II. The Smallest Factor,
++ J. Symbolic Computation (1993) 16, 131-145
++ [4] Maurice Mignotte, Some Useful Bounds,
++ Computing, Suppl. 4, 259-263 (1982), Springer-Verlag
++ Algorithms) 1st edition, 2nd printing, Addison-Wesley 1971, p. 397-398
++ Factorization: Sharp Bounds, Efficient Algorithms,
++ J. Symbolic Computation (1993) 15, 393-413
++ De Bure Frères, Paris 1829 (reprinted Œuvres, II Série, Tome IX,
++ Description:
++ \spadtype{GaloisGroupFactorizationUtilities} provides functions
++ that will be used by the factorizer.

GaloisGroupFactorizationUtilities(R,UP,F): Exports == Implementation where
R : Ring
UP : UnivariatePolynomialCategory R
F : Join(FloatingPointSystem,RetractableTo(R),Field,
      TranscendentalFunctionCategory,ElementaryFunctionCategory)
N ==> NonNegativeInteger
P ==> PositiveInteger
Z ==> Integer
Exports == with
  beauzamyBound: UP -> Z -- See [1]
  ++ beauzamyBound(p) returns a bound on the larger coefficient of any
  ++ factor of p.
  bombieriNorm: UP -> F -- See [1]
  ++ bombieriNorm(p) returns quadratic Bombieri's norm of p.
  bombieriNorm: (UP,F) -> F -- See [2] and [3]
  ++ bombieriNorm(p,n) returns the nth Bombieri's norm of p.
  ++ rootBound(p) returns a bound on the largest norm of the complex roots
  ++ of p.
  singleFactorBound: (UP,N) -> Z -- See [6]
  ++ singleFactorBound(p,r) returns a bound on the infinite norm of
  ++ the factor of p with smallest Bombieri's norm. r is a lower bound
  ++ for the number of factors of p. p shall be of degree higher or equal
  ++ to 2.
  singleFactorBound: UP -> Z -- See [6]
  ++ singleFactorBound(p,r) returns a bound on the infinite norm of
  ++ the factor of p with smallest Bombieri's norm. p shall be of degree
  ++ higher or equal to 2.
norm: (UP,P) -> F
++ norm(f,p) returns the lp norm of the polynomial f.
quadraticNorm: UP -> F
++ quadraticNorm(f) returns the l2 norm of the polynomial f.
infinityNorm: UP -> F
++ infinityNorm(f) returns the maximal absolute value of the coefficients
++ of the polynomial f.
height: UP -> F
++ height(p) returns the maximal absolute value of the coefficients of
++ the polynomial p.
length: UP -> F
++ length(p) returns the sum of the absolute values of the coefficients
++ of the polynomial p.

Implementation ==> add

import GaloisGroupUtilities(F)

height(p:UP):F == infinityNorm(p)

length(p:UP):F == norm(p,1)

norm(f:UP,p:P):F ==
  n : F := 0
  for c in coefficients f repeat
    n := n+abs(c::F)**p
  nthRoot(n,p::N)

quadraticNorm(f:UP):F == norm(f,2)

infinityNorm(f:UP):F ==
  n : F := 0
  for c in coefficients f repeat
    n := max(n,c::F)
  n

  n : N := degree p
  r := max(2,r)
  n < r => error "singleFactorBound: Bad arguments."
  nf : F := n :: F
  num : F := nthRoot(bombieriNorm(p),r)
  if F has Gamma: F -> F then
    num := num*nthRoot(Gamma(nf+1$F),2*r)
    den : F := Gamma(nf/((2*r)::F)+1$F)
  else
    num := num*(2::F)**(5/8+n/2)*exp(1$F/(4*nf))
    den : F := (pi()$F*nf)**(3/8)
  safeFloor( num/den )
singleFactorBound(p:UP):Z == singleFactorBound(p,2) -- See [6]

  n := degree p
  zero? n => 0
  lc := abs(leadingCoefficient(p)::F)
  b1 : F := 0 -- Mignotte
  b2 : F := 0 -- Knuth
  b3 : F := 0 -- Zassenhaus in [5]
  b4 : F := 0 -- Cauchy in [7]
  c : F := 0
  cl : F := 0
  for i in 1..n repeat
    c := abs(coefficient(p,(n-i)::N)::F)
    b1 := max(b1,c)
    cl := c/lc
    b2 := max(b2,nthRoot(cl,i))
    b3 := max(b3,nthRoot(cl/pascalTriangle(n,i),i))
    b4 := max(b4,nthRoot(n*cl,i))
  min(1+safeCeiling(b1/lc),min(safeCeiling(2*b2),min(safeCeiling(b3/(nthRoot(2::F,n)-1)),safeCeiling(b4)))))

beauzamyBound(f:UP):Z == -- See [1]
  d := degree f
  zero? d => safeFloor bombieriNorm f
  safeFloor( (bombieriNorm(f)*(3::F)**(3/4+d/2))/
            (2*sqrt(pi()$F*(d::F))))

  d := degree f
  b := abs(coefficient(f,0)::F)
  if zero? d then return b
  else b := b**p
  b := b+abs(leadingCoefficient(f)::F)**p
  dd := (d-1) quo 2
  for i in 1..dd repeat
    b := b+(abs(coefficient(f,i)::F)**p+abs(coefficient(f,(d-i)::N)::F)**p)/pascalTriangle(d,i)
  if even? d then
    dd := dd+1
    b := b+(abs(coefficient(f, dd::N)::F)**p/pascalTriangle(d,dd)
nthRoot(b,p::N)

bombieriNorm(f:UP):F == bombieriNorm(f,2) -- See [1]
package GALFACT GaloisGroupFactorizer

-- GaloisGroupFactorizer.input --

)set break resume
)sys rm -f GaloisGroupFactorizer.output
)spool GaloisGroupFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupFactorizer

--R GaloisGroupFactorizer(UP: UnivariatePolynomialCategory(Integer)) is a package constructor
--R Abbreviation for GaloisGroupFactorizer is GALFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GALFACT
--R
--R------------------------------- Operations --------------------------------
--R factor : UP -> Factored(UP)  musserTrials : () -> PositiveInteger
--R btwFact : (UP,Boolean,Set(NonNegativeInteger),NonNegativeInteger) -> Record(contp: Integer,factors: List(Record(irr: UP,pow: Integer)))
--R degreePartition : List(Record(factor: UP,degree: Integer)) -> Multiset(NonNegativeInteger)
--R eisensteinIrreducible? : UP -> Boolean
--R factor : (UP,NonNegativeInteger) -> Factored(UP)
--R factor : (UP,List(NonNegativeInteger)) -> Factored(UP)
--R factor : (UP,List(NonNegativeInteger),NonNegativeInteger) -> Factored(UP)
--R factor : (UP,NonNegativeInteger,NonNegativeInteger) -> Factored(UP)
--R factorOfDegree : (PositiveInteger,UP) -> Union(UP,"failed")
--R factorOfDegree : (PositiveInteger,UP,NonNegativeInteger) -> Union(UP,"failed")
--R factorOfDegree : (PositiveInteger,UP,List(NonNegativeInteger)) -> Union(UP,"failed")
--R factorOfDegree : (PositiveInteger,UP,List(NonNegativeInteger),NonNegativeInteger) -> Union(UP,"failed")
--R factorOfDegree : (PositiveInteger,UP,List(NonNegativeInteger),NonNegativeInteger,Boolean) -> Union(UP,"failed")
--R factorSquareFree : UP -> Factored(UP)
--R factorSquareFree : (UP,NonNegativeInteger) -> Factored(UP)
--R factorSquareFree : (UP,List(NonNegativeInteger)) -> Factored(UP)
--R factorSquareFree : (UP,List(NonNegativeInteger),NonNegativeInteger) -> Factored(UP)
--R factorSquareFree : (UP,NonNegativeInteger,NonNegativeInteger) -> Factored(UP)
--R henselFact : (UP,Boolean) -> Record(contp: Integer,factors: List(Record(irr: UP,pow: Integer)))
--R makeFR : Record(contp: Integer,factors: List(Record(irr: UP,pow: Integer))) -> Factored(UP)
---R modularFactor : UP -> Record(prime: Integer, factors: List(UP))
---R musserTrials : PositiveInteger -> PositiveInteger
---R numberOfFactors : List(Record(factor: UP, degree: Integer)) -> NonNegativeInteger
---R stopMusserTrials : () -> PositiveInteger
---R tryFunctionalDecomposition : Boolean -> Boolean
---R tryFunctionalDecomposition? : () -> Boolean
---R useEisensteinCriterion : Boolean -> Boolean
---R useEisensteinCriterion? : () -> Boolean
---R useSingleFactorBound : Boolean -> Boolean
---R useSingleFactorBound? : () -> Boolean
---R
---E 1

)spool
)lisp (bye)

---

---

--- GaloisGroupFactorizer.help ---

====================================================================
GaloisGroupFactorizer examples
====================================================================

GaloisGroupFactorizer provides functions to factor resolvents.

See Also:
o )show GaloisGroupFactorizer

---

GaloisGroupFactorizer (GALFACT)
Exports:
btwFact degreePartition eisensteinIrreducible? factor
factorSquareFree henselFact makeFR modularFactor
numberOfFactors stopMusserTrials tryFunctionalDecomposition
useEisensteinCriterion? useSingleFactorBound useSingleFactorBound?

— package GALFACT GaloisGroupFactorizer —

)abbrev package GALFACT GaloisGroupFactorizer
++ Author: Frederic Lehobey
++ Date Created: 28 June 1994
++ Date Last Updated: 11 July 1997
++ References:
++ Factorization: Sharp Bounds, Efficient Algorithms,
++ J. Symbolic Computation (1993) 15, 393-413
++ [2] John Brillhart, Note on Irreducibility Testing,
++ [3] David R. Musser, On the Efficiency of a Polynomial Irreducibility Test,
++ Description:
++ \spadtype{GaloisGroupFactorizer} provides functions to factor resolvents.
-- improvements to do :
-- + reformulate the lifting problem in completeFactor -- See [1] (hard)
-- + implement algorithm RC -- See [1] (easy)
-- + use Dedekind’s criterion to prove sometimes irreducibility (easy)
-- or even to improve early detection of true factors (hard)
-- + replace Sets by Bits

GaloisGroupFactorizer(UP):Exports == Implementation where
Z ==> Integer
UP: UnivariatePolynomialCategory Z
N ==> NonNegativeInteger
P ==> PositiveInteger
CYC ==> CyclotomicPolynomialPackage()
SUPZ ==> SparseUnivariatePolynomial Z
ParFact ==> Record(irr: UP, pow: Z)
FinalFact == Record(contp: Z, factors: List ParFact)
DDRecord == Record(factor: UP, degree: Z) -- a Distinct-Degree factor
DDList == List DDRecord
MFact == Record(prime: Z, factors: List UP) -- Modular Factors
LR == Record(left: UP, right: UP) -- Functional decomposition
Exports == with
makeFR: FinalFact -> Factored UP
++ makeFR(flist) turns the final factorization of henselFact into a
++ \spadtype{Factored} object.
degreePartition: DDList -> Multiset N
++ degreePartition(ddfactorization) returns the degree partition of
++ the polynomial f modulo p where ddfactorization is the distinct
++ degree factorization of f computed by ddFact for some prime p.

musserTrials: () -> P
++ musserTrials() returns the number of primes that are tried in
++ \spadfun{modularFactor}.

musserTrials: P -> P
++ musserTrials(n) sets to n the number of primes to be tried in
++ \spadfun{modularFactor} and returns the previous value.

stopMusserTrials: () -> P
++ stopMusserTrials() returns the bound on the number of factors for
++ which \spadfun{modularFactor} stops to look for an other prime. You
++ will have to remember that the step of recombining the extraneous
++ factors may take up to \spad{2**stopMusserTrials()} trials.

stopMusserTrials: P -> P
++ stopMusserTrials(n) sets to n the bound on the number of factors for
++ which \spadfun{modularFactor} stops to look for an other prime. You
++ will have to remember that the step of recombining the extraneous
++ factors may take up to \spad{2**n} trials. Returns the previous
++ value.

numberOfFactors: DDList -> N
++ numberOfFactors(ddfactorization) returns the number of factors of
++ the polynomial f modulo p where ddfactorization is the distinct
++ degree factorization of f computed by ddFact for some prime p.

modularFactor: UP -> MFact
++ modularFactor(f) chooses a "good" prime and returns the factorization
++ of f modulo this prime in a form that may be used by
++ \spadfun{completeHensel}. If prime is zero
++ it means that f has been proved to be irreducible over the integers
++ or that f is a unit (i.e. 1 or -1).
++ f shall be primitive (i.e. content(p)=1) and square free (i.e.
++ without repeated factors).

useSingleFactorBound?: () -> Boolean
++ useSingleFactorBound?() returns \spad{true} if algorithm with single
++ factor bound is used for factorization, \spad{false} for algorithm
++ with overall bound.

useSingleFactorBound: Boolean -> Boolean
++ useSingleFactorBound(b) chooses the algorithm to be used by the
++ factorizers: \spad{true} for algorithm with single
++ factor bound, \spad{false} for algorithm with overall bound.
++ Returns the previous value.

useEisensteinCriterion?: () -> Boolean
++ useEisensteinCriterion?() returns \spad{true} if factorizers
++ check Eisenstein’s criterion before factoring.

useEisensteinCriterion: Boolean -> Boolean
++ useEisensteinCriterion(b) chooses whether factorizers check
++ Eisenstein’s criterion before factoring: \spad{true} for
++ using it, \spad{false} else. Returns the previous value.

eisensteinIrreducible?: UP -> Boolean
++ eisensteinIrreducible?(p) returns \spad{true} if p can be
++ shown to be irreducible by Eisenstein’s criterion,
++ \spad{false} is inconclusive.
tryFunctionalDecomposition?: () -> Boolean
++ tryFunctionalDecomposition?() returns \spad{true} if
++ factorizers try functional decomposition of polynomials before
++ factoring them.
tryFunctionalDecomposition: Boolean -> Boolean
++ tryFunctionalDecomposition(b) chooses whether factorizers have
++ to look for functional decomposition of polynomials
++ (\spad{true}) or not (\spad{false}). Returns the previous value.
factor: UP -> Factored UP
++ factor(p) returns the factorization of p over the integers.
factor: (UP,N) -> Factored UP
++ factor(p,r) factorizes the polynomial p using the single factor bound
++ algorithm and knowing that p has at least r factors.
factor: (UP,List N) -> Factored UP
++ factor(p,listOfDegrees) factorizes the polynomial p using the single
++ factor bound algorithm and knowing that p has for possible
++ splitting of its degree listOfDegrees.
factor: (UP,List N,N) -> Factored UP
++ factor(p,listOfDegrees,r) factorizes the polynomial p using the single
++ factor bound algorithm, knowing that p has for possible
++ splitting of its degree listOfDegrees and that p has at least r
++ factors.
factor: (UP,N,N) -> Factored UP
++ factor(p,d,r) factorizes the polynomial p using the single
++ factor bound algorithm, knowing that d divides the degree of all
++ factors of p and that p has at least r factors.
factorSquareFree: UP -> Factored UP
++ factorSquareFree(p) returns the factorization of p which is supposed
++ not having any repeated factor (this is not checked).
factorSquareFree: (UP,N) -> Factored UP
++ factorSquareFree(p,r) factorizes the polynomial p using the single
++ factor bound algorithm and knowing that p has at least r factors.
++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,List N) -> Factored UP
++ factorSquareFree(p,listOfDegrees) factorizes the polynomial p using
++ the single factor bound algorithm and knowing that p has for possible
++ splitting of its degree listOfDegrees.
++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,List N,N) -> Factored UP
++ factorSquareFree(p,listOfDegrees,r) factorizes the polynomial p using
++ the single factor bound algorithm, knowing that p has for possible
++ splitting of its degree listOfDegrees and that p has at least r
++ factors.
++ f is supposed not having any repeated factor (this is not checked).
factorSquareFree: (UP,N,N) -> Factored UP
++ factorSquareFree(p,d,r) factorizes the polynomial p using the single
++ factor bound algorithm, knowing that d divides the degree of all
++ factors of p and that p has at least r factors.
++ f is supposed not having any repeated factor (this is not checked).
factorOfDegree: (P,UP) -> Union(UP,"failed")
++ factorOfDegree(d,p) returns a factor of p of degree d.
factorOfDegree: (P,UP,N) -> Union(UP,"failed")
++ factorOfDegree(d,p,r) returns a factor of p of degree
++ d knowing that p has at least r factors.
factorOfDegree: (P,UP,List N) -> Union(UP,"failed")
++ factorOfDegree(d,p,listOfDegrees) returns a factor
++ of p of degree d knowing that p has for possible splitting of its
++ degree listOfDegrees.
factorOfDegree: (P,UP,List N,N) -> Union(UP,"failed")
++ factorOfDegree(d,p,listOfDegrees,r) returns a factor
++ of p of degree d knowing that p has for possible splitting of its
++ degree listOfDegrees, and that p has at least r factors.
factorOfDegree: (P,UP,List N,N,Boolean) -> Union(UP,"failed")
++ factorOfDegree(d,p,listOfDegrees,r,sqf) returns a factor
++ of p of degree d knowing that p has for possible splitting of its
++ degree listOfDegrees, and that p has at least r factors.
++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
++ without repeated factors).
henselFact: (UP,Boolean) -> FinalFact
++ henselFact(p,sqf) returns the factorization of p, the result
++ is a Record such that \spad{contp=}content p,
++ \spad{factors=}List of irreducible factors of p with exponent.
++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
++ without repeated factors).
btwFact: (UP,Boolean,Set N,N) -> FinalFact
++ btwFact(p,sqf,pd,r) returns the factorization of p, the result
++ is a Record such that \spad{contp=}content p,
++ \spad{factors=}List of irreducible factors of p with exponent.
++ If \spad{sqf=true} the polynomial is assumed to be square free (i.e.
++ without repeated factors).
++ pd is the \spad{type} of possible degrees. r is a lower bound for
++ the number of factors of p. Please do not use this function in your
++ code because its design may change.

Implementation ==> add

fUnion ==> Union("nil", "sqfr", "irred", "prime")  
FFE ==> Record(flg:fUnion, fctr:UP, xpnt:Z) -- Flag-Factor-Exponent  
DDFact ==> Record(prime:Z, ddfactors:DDList) -- Distinct Degree Factors  
HLR ==> Record(list:List UP, modulo:Z) -- HenselLift Record

mussertrials: P := 5
stopmussertrials: P := 8
usesinglefactorbound: Boolean := true
tryfunctionaldecomposition: Boolean := true
useeisensteinCriterion: Boolean := true

useEisensteinCriterion?():Boolean == useeisensteinCriterion
useEisensteinCriterion(b:Boolean):Boolean ==
  (useeisensteinCriterion,b) := (b,useeisensteinCriterion)
  b

tryFunctionalDecomposition?():Boolean == tryfunctionaldecomposition

tryFunctionalDecomposition(b:Boolean):Boolean ==
  (tryfunctionaldecomposition,b) := (b,tryfunctionaldecomposition)
  b

useSingleFactorBound?():Boolean == usesinglefactorbound

useSingleFactorBound(b:Boolean):Boolean ==
  (usesinglefactorbound,b) := (b,usesinglefactorbound)
  b

stopMusserTrials():P == stopmussertrials

stopMusserTrials(n:P):P ==
  (stopmussertrials,n) := (n,stopmussertrials)
  n

musserTrials():P == mussertrials

musserTrials(n:P):P ==
  (mussertrials,n) := (n,mussertrials)
  n

import GaloisGroupFactorizationUtilities(Z,UP,Float)

import GaloisGroupPolynomialUtilities(Z,UP)

import IntegerPrimesPackage(Z)
import IntegerFactorizationPackage(Z)
import ModularDistinctDegreeFactorizer(UP)

eisensteinIrreducible?(f:UP):Boolean ==
  rf := reductum f
  c: Z := content rf
  zero? c => false
  unit? c => false
  lc := leadingCoefficient f
  tc := lc
  while not zero? rf repeat
    tc := leadingCoefficient rf
    rf := reductum rf
    for p in factors(factor c)$Factored(Z) repeat
      -- if (one? p.exponent) and (not zero? (lc rem p.factor)) and
      if (p.exponent = 1) and (not zero? (lc rem p.factor)) and
(not zero? (tc rem ((p.factor)**2))) then return true
false

numberOfFactors(ddlist:DDList):N ==
n: N := 0
d: Z := 0
for dd in ddlist repeat
   n := n +
   zero? (d := degree(dd.factor)::Z) => 1
   (d quo dd.degree)::N
   n

-- local function, returns the a Set of shifted elements
shiftSet(s:Set N,shift:N):Set N == set [ e+shift for e in parts s ]

-- local function, returns the "reductum" of an Integer (as chain of bits)
reductum(n:Z):Z == n-shift(1,length(n)-1)

-- local function, returns an integer with level lowest bits set to 1
seed(level:Z):Z == shift(1,level)-1

-- local function, returns the next number (as a chain of bit) for
-- factor reconciliation of a given level (which is the number of
-- extraneous factors involved) or "End of level" if not any
nextRecNum(levels:N,level:Z,n:Z):Union("End of level",Z) ==
   if (l := length n)<levels then return(n+shift(1,l-1))
   (n=shift(seed(level),levels-level)) => "End of level"
   b: Z := 1
   while (((1-b) = (lr := length(n := reductum n))))@Boolean repeat b := b+1
   reductum(n)+shift(seed(b+1),lr)

-- local function, return the set of N, 0..n
fullSet(n:N):Set N == set [ i for i in 0..n ]

modularFactor(p:UP):MFact ==
   -- not one? abs(content(p)) =>
   not (abs(content(p)) = 1) =>
      error "modularFactor: the polynomial is not primitive."
   zero? (n := degree p) => [0,[p]]

-- declarations --
cprime: Z := 2
trials: List DDFact := empty()
d: Set N := fullSet(n)
dirred: Set N := set [0,n]
s: Set N := empty()
ddlist: DDFList := empty()
degfact: N := 0
nf: N := stopmussertrials+1
i: Z
-- Musser, see [3] --

diffp := differentiate p
for i in 1..mussertials | nf>stopmussertials repeat
  -- test 1: cprime divides leading coefficient
  -- test 2: "bad" primes: (in future: use Dedekind's Criterion)
  while (zero? ((leadingCoefficient p) rem cprime)) or
    (not zero? degree gcd(p, diffp, cprime)) repeat
    cprime := nextPrime(cprime)
  ddlist := ddFact(p, cprime)
  -- degree compatibility: See [3] --
  s := set [0]
  for f in ddlist repeat
    degfact := f.degree::N
    if not zero? degfact then
      for j in 1..(degree(f.factor) quo degfact) repeat
        s := union(s, shiftSet(s, degfact))
  trials := cons([cprime, ddlist] $DDFact, trials)
  d := intersect(d, s)
  d = dirred => return [0, [p]] -- p is irreducible
  cprime := nextPrime(cprime)
  nf := numberOfFactors ddlist
  -- choose the one with the smallest number of factors
  choice := first trials
  nfc := numberOfFactors(choice.ddfactors)
  for t in rest trials repeat
    nf := numberOfFactors(t.ddfactors)
    if nf<nfc or ((nf=nfc) and (t.prime>choice.prime)) then
      nfc := nf
      choice := t
  cprime := choice.prime
  -- HenselLift$GHENSEL expects the degree 0 factor first
  [cprime, separateFactors(choice.ddfactors, cprime)]

degreePartition(ddlist: DDList): Multiset N ==
  dp: Multiset N := empty()
  d: N := 0
  dd: N := 0
  for f in ddlist repeat
    zero? (d := degree(f.factor)) => dp := insert!(0, dp)
    dd := f.degree::N
    dp := insert!(dd, dp, d quo dd)
  dp
package GalFact \ GaloisGroupFactorizer

-- positive leading coefficient and non zero trailing coefficient,
-- using the overall bound technique. If pdecomp is true then look
-- for a functional decomposition of f.

henselfact(f: UP, pdecomp: Boolean): List UP ==
  if brillhartIrreducible? f or
    (useEisensteinCriterion => eisensteinIrreducible? f ; false)
  then return [f]
  cf: Union(LR, "failed")
  if pdecomp and tryFunctionalDecomposition then
    cf := monicDecomposeIfCan f
  else
    cf := "failed"
  cf case "failed" =>
    m := modularFactor f
    zero? (cprime := m.prime) => m.factors
    b: P := (2*leadingCoefficient(f)*beauzamyBound(f)) :: P
    completeHensel(f, m.factors, cprime, b)
    lrf := cf::LR
    "append"/[ henselfact(g(lrf.right),false) for g in
      henselfact(lrf.left,true) ]

-- local function, returns the complete factorization of its arguments,
-- using the single-factor bound technique

  lc := leadingCoefficient f
  f0 := coefficient(f, 0)
  ltrue: List UP := empty()
  found? := true
  degf: N := 0
  degg: N := 0
  g0: Z := 0
  g: UP := 0
  rg: N := 0
  nb: Z := 0
  lg: List UP := empty()
  b: P := 1
  dg: Set N := empty()
  llg: HLR := [empty(), 0]
  levels: N := #lf
  level: Z := 1
  ic: Union(Z, "End of level") := 0
  i: Z := 0
  while level<levels repeat
    -- try all possible factors with degree in d
    ic := seed(level)
    while ((not found?) and (ic case Z)) repeat
      i := ic::Z
      degg := 0
      g0 := 1 -- LC algorithm
      for j in 1..levels repeat
if bit?(i,j-1) then  
    degg := degg+degree lf.j
    g0 := g0*coefficient(lf.j,0) -- LC algorithm
    g0 := symmetricRemainder(lc*g0,pk) -- LC algorithm
if member?(degg,d) and (((lc*f0) exquo g0) case Z) then  
    -- LC algorithm
    g := lc::UP -- build the possible factor -- LC algorithm
for j in 1..levels repeat if bit?(i,j-1) then g := g*lf.j
    g := primitivePart reduction(g,pk)
f1 := f exquo g
if f1 case UP then -- g is a true factor
    found? := true
    -- remove the factors of g from lf
    nb := 1
for j in 1..levels repeat
    if bit?(i,j-1) then
        swap!(lf,j,nb)
        nb := nb+1
lg := lf
lf := rest(lf,level::N)
setrest!(rest(lg,(level-1)::N),empty()$List(UP))
f := f1::UP
lc := leadingCoefficient f
f0 := coefficient(f,0)
-- is g irreducible?
dg := select(x+->x <= degg,d)
if not(dg=set [0,degg]) then -- implies degg >= 2
    rg := max(2,(r+level-2)::N)
b := (2*leadingCoefficient(g)*singleFactorBound(g,rg)) :: P
if b>pk and (not brillhartIrreducible?(g)) and
    (useeisensteinCriterion => not eisensteinIrreducible?(g) ; true)
then  
    -- g may be reducible
    llg := HenselLift(g,lg,cprime,b)
gpk: P := (llg.modulo)::P
    -- In case exact factorisation has been reached by
    -- HenselLift before coefficient bound.
    if gpk<b then
        lg := llg.plist
    else
        lg := completeFactor(g,llg.plist,cprime,gpk,rg,dg)
else
    lg := [ g ] -- g irreducible
else lg := [ g ] -- g irreducible
ltrue := append(ltrue,lg)
r := max(2,(r-#ltrue)::N)
degf := degree f
d := select(x+->x <= deffd,d)
if deff<=1 then -- if exhausted
    -- if one? deff then
if (degf = 1) then
    ltrue := cons(f,ltrue)
return ltrue -- 1st exit, all factors found
else -- can we go on with the same pk?
    b := (2*lc*singleFactorBound(f,r)) :: P
if b>pk then -- unlucky: no we can’t
    llg := HenselLift(f,lf,cprime,b) -- I should reformulate
        -- the lifting probleme, but hadn’t time for that.
        -- In any case, such case should be quite exceptional.
    lf := llg.plist
    pk := (llg.modulo)::P
        -- In case exact factorisation has been reached by
        -- HenselLift before coefficient bound.
if pk<b then return append(lf,ltrue) -- 2nd exit
level := 1
    ic := nextRecNum(levels,level,i)
if found? then
    levels := #lf
    found? := false
if not (ic case Z) then level := level+1
cons(f,ltrue) -- 3rd exit, the last factor was irreducible but not "true"

-- local function, returns the set of elements "divided" by an integer
divideSet(s:Set N, n:N):Set N ==
    l: List N := [ 0 ]
for e in parts s repeat
    if (ee := (e exquo n)$N) case N then l := cons(ee::N,l)
set(l)

-- Beauzamy-Trevisan-Wang FACTOR, see [1] with some refinements
-- and some differences. f is assumed to be primitive, square-free
-- and with positive leading coefficient. If pdecomp is true then
-- look for a functional decomposition of f.
btwFactor(f:UP,d:Set N,r:N,pdecomp:Boolean):List UP ==
    df := degree f
    not (max(d) = df) => error "btwFact: Bad arguments"
reverse? := Boolean := false
negativelc?: Boolean := false

    (d = set [0,df]) => [ f ]
if abs(coefficient(f,0))<abs(leadingCoefficient(f)) then
    f := reverse f
 reverse?: := true
brillhartIrreducible? f or
    (useeisensteincriterion => eisensteinIrreducible?(f) ; false) =>
        if reverse? then [ reverse f ] else [ f ]
if leadingCoefficient(f)<0 then
    f := -f
 negativelc?: := true
cf: Union(LR,"failed")
if pdecomp and tryfunctionaldecomposition then
    cf := monicDecomposeIfCan f
else
    cf := "failed"
if cf case "failed" then
    m := modularFactor f
zero? (cprime := m.prime) =>
    if reverse? then
        if negativelc? then return [ -reverse f ]
        else return [ reverse f ]
    else if negativelc? then return [ -f ]
    else return [ f ]
if noLinearFactor? f then d := remove(1,d)
lc := leadingCoefficient f
f0 := coefficient(f,0)
b: P := (2*lc*singleFactorBound(f,r)) :: P -- LC algorithm
lm := HenselLift(f,m.factors,cprime,b)
lf := lm.plist
pk: P := (lm.modulo)::P
if ground? first lf then lf := rest lf
-- in case exact factorisation has been reached by HenselLift
-- before coefficient bound
if not pk < b then lf := completeFactor(f,lf,cprime,pk,r,d)
else
    lrf := cf::LR
    dh := degree lrf.right
    lg := btwFactor(lrf.left,divideSet(d,dh),2,true)
lf: List UP := empty()
for i in 1..#lg repeat
    g := lg.i
dgh := (degree g)*dh
    df := subtractIfCan(df,dgh)::N
    lfg := btwFactor(g(lrf.right),
        select(x+->x <= dgh,d),max(2,r-df)::N,false)
    lf := append(lf,lfg)
r := max(2,r-#lfg)::N
if reverse? then lf := [ reverse(fact) for fact in lf ]
for i in 1..#lf repeat
    if leadingCoefficient(lf.i)<0 then lf.i := -lf.i
-- because we assume f with positive leading coefficient
lf
makeFR(flist:FinalFact):Factored UP ==
    ctp := factor flist.contp
    fflist: List FFE := empty()
    for ff in flist.factors repeat
        fflist := cons(["prime", ff.irr, ff.pow]$FFE, fflist)
    for fc in factorList ctp repeat
        fflist := cons([fc.flg, fc.fctr::UP, fc.xpnt]$FFE, fflist)
    makeFR(unit(ctp)::UP, fflist)
import IntegerRoots(Z)

-- local function, factorizes a quadratic polynomial
quadratic(p:UP):List UP ==
a := leadingCoefficient p
b := coefficient(p,1)
d := b**2-4*a*coefficient(p,0)
r := perfectSqrt(d)
r case "failed" => [p]
b := b+(r::Z)
a := 2*a
d := gcd(a,b)
-- if not one? d then
if not (d = 1) then
  a := a quo d
  b := b quo d
f: UP := monomial(a,1)+monomial(b,0)
  cons(f,[(p exquo f)::UP])

isPowerOf2(n:Z): Boolean ==
n = 1 => true
qr: Record(quotient: Z, remainder: Z) := divide(n,2)
qr.remainder = 1 => false
isPowerOf2 qr.quotient

subMinusX(supPol: SUPZ): UP ==
  minusX: SUPZ := monomial(-1,1)$SUPZ
  unmakeSUP(elt(supPol,minusX)$SUPZ)

henselFact(f:UP, sqf:Boolean):FinalFact ==
  factorlist: List(ParFact) := empty()

  -- make m primitive
  c: Z := content f
  f := (f exquo c)::UP

  -- make the leading coefficient positive
  if leadingCoefficient f < 0 then
    c := -c
    f := -f

  -- is x**d factor of f
  if (d := minimumDegree f) > 0 then
    f := monicDivide(f,monomial(1,d)).quotient
    factorlist := [[monomial(1,1),d]$ParFact]
  d := degree f

  -- is f constant?
zero? d => [c,factorlist] \text{FinalFact}

-- is f linear?
one? d => [c,cons([f,1]\text{ParFact},factorlist)] \text{FinalFact}
(d = 1) => [c,cons([f,1]\text{ParFact},factorlist)] \text{FinalFact}

lcPol: UP := leadingCoefficient(f) :: UP

-- is f cyclotomic \((x^n - 1)\)?
lcPol = reductum(f) => -- if true, both will = 1
for fac in map(z+->unmakeSUP(z)\text{SUP},
cyclotomicDecomposition(d)\text{CYC}$ListFunctions2(SUPZ,UP) repeat
factorlist := cons([fac,1]\text{ParFact},factorlist)
[c,factorlist] \text{FinalFact}

-- is f odd cyclotomic \((x^{2n+1} + 1)\)?
odd?(d) and (lcPol = reductum(f)) =>
for sfac in cyclotomicDecomposition(d)\text{CYC} repeat
fac := subMinusX sfac
if leadingCoefficient fac < 0 then fac := -fac
factorlist := cons([fac,1]\text{ParFact},factorlist)
[c,factorlist] \text{FinalFact}

-- is the poly of the form \((x^n + 1)\) with \(n\) a power of 2?
-- if so, then irreducible
isPowerOf2(d) and (lcPol = reductum(f)) =>
factorlist := cons([f,1]\text{ParFact},factorlist)
[c,factorlist] \text{FinalFact}

-- other special cases to implement...

-- f is square-free :
sqf => [c, append([pf,1]\text{ParFact} for pf in henselfact(f,true)],
factorlist)] \text{FinalFact}

-- f is not square-free :
sqfflist := factors squareFree f
for sqfr in sqfflist repeat
mult := sqfr.exponent
sqff := sqfr.factor
d := degree sqff
-- one? d => factorlist := cons([sqff,mult]\text{ParFact},factorlist)
(d = 1) => factorlist := cons([sqff,mult]\text{ParFact},factorlist)
d=2 =>
factorlist := append([pf,mult]\text{ParFact} for pf in quadratic(sqff)],
factorlist)
factorlist := append([pf,mult]\text{ParFact} for pf in henselfact(sqff,\text{true}]),factorlist)
[c,factorlist] \text{FinalFact}
btwFact(f:UP, sqf:Boolean, fd:Set N, r:N):FinalFact ==
d := degree f
not(max(fd)=d) => error "btwFact: Bad arguments"
factorlist: List(ParFact) := empty()

-- make m primitive
c := content f
f := (f exquo c)::UP

-- make the leading coefficient positive
if leadingCoefficient f < 0 then
c := -c
f := -f

-- is x**d factor of f
if (maxd := minimumDegree f) > 0 then
f := monicDivide(f,monomial(1,maxd)).quotient
factorlist := [[monomial(1,1),maxd]$ParFact]
r := max(2,r-maxd)::N
d := subtractIfCan(d,maxd)::N
fd := select(x+->x <= d,fd)

-- is f constant?
zero? d => [c,factorlist]$FinalFact

-- is f linear?
-- one? d => [c,cons([f,1]$ParFact,factorlist)$FinalFact
(d = 1) => [c,cons([f,1]$ParFact,factorlist)$FinalFact

lcPol: UP := leadingCoefficient(f) :: UP

-- is f cyclotomic (x**n - 1)?
-lcPol = reductum(f) => -- if true, both will = 1
for fac in map(z+->unmakeSUP(z)$UP,
cyclotomicDecomposition(d)$CYC)$ListFunctions2(SUPZ,UP) repeat
factorlist := cons([fac,1]$ParFact,factorlist)
[c,factorlist]$FinalFact

-- is f odd cyclotomic (x**(2*n+1) + 1)?
odd?(d) and (lcPol = reductum(f)) =>
for sfac in cyclotomicDecomposition(d)$CYC repeat
fac := subMinusX sfac
if leadingCoefficient fac < 0 then fac := -fac
factorlist := cons([fac,1]$ParFact,factorlist)
[c,factorlist]$FinalFact

-- is the poly of the form x**n + 1 with n a power of 2?
if so, then irreducible
isPowerOf2(d) and (lcPol = reductum(f)) =>
factorlist := cons([f,1]$ParFact,factorlist)
[c,factorlist]$FinalFact

-- other special cases to implement...

-- f is square-free:
\texttt{sqf} => [c, append([[pf, 1]$ParFact for \texttt{pf} in \texttt{btwFactor(f,fd,r,true)}],
factorlist)]$FinalFact

-- f is not square-free:
\texttt{sqfflist} := factors \texttt{squareFree(f)}

-- if one?(#(\texttt{sqfflist})) then -- indeed \texttt{f} was a power of a square-free
if ((#(\texttt{sqfflist})) = 1) then -- indeed \texttt{f} was a power of a square-free
\texttt{r} := max(r \texttt{quo} ((\texttt{first sqfflist}).\texttt{exponent}),2)::\texttt{N}
else
\texttt{r} := 2
for \texttt{sqfr} in \texttt{sqfflist} repeat
\texttt{mult} := \texttt{sqfr}.\texttt{exponent}
\texttt{sqff} := \texttt{sqfr}.\texttt{factor}
\texttt{d} := degree \texttt{sqff}

--
(d = 1) =>
\texttt{factorlist} := cons([\texttt{sqff}, \texttt{mult}]$ParFact, \texttt{factorlist})
\texttt{maxd} := (max(fd)-\texttt{mult})::\texttt{N}
\texttt{fd} := select(x+\rightarrow x <= \texttt{maxd}, \texttt{fd})
d=2 =>
\texttt{factorlist} := append([[\texttt{pf}, \texttt{mult}]$ParFact for \texttt{pf} in \texttt{quadratic(sqff)}],
factorlist)
\texttt{maxd} := (max(fd)-2*\texttt{mult})::\texttt{N}
\texttt{fd} := select(x+\rightarrow x <= \texttt{maxd}, \texttt{fd})
\texttt{factorlist} := append([[\texttt{pf}, \texttt{mult}]$ParFact for \texttt{pf} in \texttt{btwFactor(sqff,select(x+\rightarrow x <= d,fd),r,true)}],factorlist)
\texttt{maxd} := (max(fd)-d*\texttt{mult})::\texttt{N}
\texttt{fd} := select(x+\rightarrow x <= \texttt{maxd}, \texttt{fd})
[c,factorlist]$FinalFact

factor(f:UP):Factored UP ==
makeFR
  usesinglefactorbound => btwFact(f,false,fullSet(degree f),2)
henselFact(f,false)

-- local function, returns true if the sum of the elements of the list
-- is not the degree.
errorsum?(d: \texttt{N}, ld: \texttt{List N}): \texttt{Boolean} == not (d = +/ld)

-- local function, turns list of degrees into a Set
makeSet(ld: \texttt{List N}): \texttt{Set N} ==
s := set [0]
for \texttt{d} in ld repeat s := union(s, shiftSet(s,d))
s
factor(f:UP,ld:List N,r:N):Factored UP ==
  errorsum?(degree f,ld) => error "factor: Bad arguments"
  makeFR btwFact(f,false,makeSet(ld),r)

factor(f:UP,r:N):Factored UP ==
  makeFR btwFact(f,false,fullSet(degree f),r)

factor(f:UP,ld:List N):Factored UP ==
  factor(f,ld,2)

factorSquareFree(f:UP):Factored UP ==
  makeFR
    usesinglefactorbound => btwFact(f,true,fullSet(degree f),2)
    henselFact(f,true)

factorSquareFree(f:UP,ld:List(N),r:N):Factored UP ==
  errorsum?(degree f,ld) => error "factorSquareFree: Bad arguments"
  makeFR btwFact(f,true,makeSet(ld),r)

factorSquareFree(f:UP,r:N):Factored UP ==
  makeFR btwFact(f,true,fullSet(degree f),r)

factorSquareFree(f:UP,ld:List N):Factored UP ==
  factorSquareFree(f,ld,2)

factorSquareFree(f:UP,d:N,r:N):Factored UP ==
  n := (degree f) exquo d
  n case "failed" => error "factorSquareFree: Bad arguments"
  factorSquareFree(f,new(n::N,d)$List(N),r)

  dp := degree p
  errorsum?(dp,ld) => error "factorOfDegree: Bad arguments"
  --
  (one? (d::N)) and noLinearFactor?(p) => "failed"
  (if :btwFact(p,sqf,makeSet(ld),r).factors
    for f in if repeat
    degree(f.irr)=d => return f.irr
  "failed"

factorOfDegree(d:P,p:UP,ld:List N, r:N):Union(UP, "failed") ==
  factorOfDegree(d,p,ld,r, false)

factorOfDegree(d:P,p:UP,r:N):Union(UP, "failed") ==
  factorOfDegree(d,p,new(degree p,1)$List(N),r,false)

factorOfDegree(d:P,p:UP,ld:List N):Union(UP, "failed") ==
  factorOfDegree(d,p,ld,2, false)
factorOfDegree(d: P, p: UP) : Union(UP, "failed") ==
    factorOfDegree(d, p, new(degree p, 1)$List(N), 2, false)

---

--- GALFACT.dotabb ---

"GALFACT" [color="#FF4488", href="bookvol10.4.pdf#nameddest=GALFACT"]
"FSAGG" [color="#4488FF", href="bookvol10.2.pdf#nameddest=FSAGG"]
"GALFACT" \rightarrow "FSAGG"

---

package GALPOLYU GaloisGroupPolynomialUtilities

--- GaloisGroupPolynomialUtilities.input ---

)set break resume
)sys rm -f GaloisGroupPolynomialUtilities.output
)spool GaloisGroupPolynomialUtilities.output
)set message GaloisGroupPolynomialUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupPolynomialUtilities

--R GaloisGroupPolynomialUtilities(R: Ring, UP: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for GaloisGroupPolynomialUtilities is GALPOLYU
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GALPOLYU
--R
--R------------------------------ Operations ------------------------------
--R monic?: UP -> Boolean  reverse : UP -> UP
--R scaleRoots : (UP, R) -> UP  shiftRoots : (UP, R) -> UP
--R unvectorise : Vector(R) -> UP
--R degreePartition : Factored(UP) -> Multiset(NonNegativeInteger)
--R factorOfDegree : (PositiveInteger, Factored(UP)) -> UP
--R factorsOfDegree : (PositiveInteger, Factored(UP)) -> List(UP)

--E 1

)spool
)lisp (bye)
GaloisGroupPolynomialUtilities (GALPOLYU)

Exports:

degreePartition factorOfDegree factorsOfDegree monic? reverse
scaleRoots shiftRoots unvectorise

---

package GALPOLYU GaloisGroupPolynomialUtilities ---

)abbrev package GALPOLYU GaloisGroupPolynomialUtilities
++ Author: Frederic Lehobey
++ Date Created: 30 June 1994
++ Date Last Updated: 15 July 1994
++ Description:
++ \spadtype{GaloisGroupPolynomialUtilities} provides useful
++ functions for univariate polynomials which should be added to
++ \spadtype{UnivariatePolynomialCategory} or to \spadtype{Factored}
GaloisGroupPolynomialUtilities(R,UP): Exports == Implementation where
  R : Ring
  UP : UnivariatePolynomialCategory R
  N ==> NonNegativeInteger
  P ==> PositiveInteger

Exports ==> with
  monic?: UP -> Boolean
    ++ monic?(p) tests if p is monic (i.e. leading coefficient equal to 1).
  unvectorise: Vector R -> UP
    ++ unvectorise(v) returns the polynomial which has for coefficients the
    ++ entries of v in the increasing order.
  reverse: UP -> UP
    ++ reverse(p) returns the reverse polynomial of p.
  scaleRoots: (UP,R) -> UP
    ++ scaleRoots(p,c) returns the polynomial which has c times the roots
    ++ of p.
  shiftRoots: (UP,R) -> UP
    ++ shiftRoots(p,c) returns the polynomial which has for roots c added
    ++ to the roots of p.
  degreePartition: Factored UP -> Multiset N
    ++ degreePartition(f) returns the degree partition (i.e. the multiset
    ++ of the degrees of the irreducible factors) of
    ++ the polynomial f.
  factorOfDegree: (P, Factored UP) -> UP
    ++ factorOfDegree(d,f) returns a factor of degree d of the factored
    ++ polynomial f. Such a factor shall exist.
  factorsOfDegree: (P, Factored UP) -> List UP
    ++ factorsOfDegree(d,f) returns the factors of degree d of the factored
    ++ polynomial f.

Implementation ==> add

import Factored UP

factorsOfDegree(d:P,r:Factored UP):List UP ==
  lfact : List UP := empty()
  for fr in factors r | degree(fr.factor)=(d::N) repeat
    for i in 1..fr.exponent repeat
      lfact := cons(fr.factor,lfact)
  lfact

factorOfDegree(d:P,r:Factored UP):UP ==
  factor : UP := 0
  for i in 1..numberOfFactors r repeat
    factor := nthFactor(r,i)
    if degree(factor)=(d::N) then return factor
  error "factorOfDegree: Bad arguments"
degreePartition(r: Factored UP): Multiset N ==
multiset([degree(nthFactor(r,i)) for i in 1..numberOfFactors r])

-- monic?(p:UP):Boolean == one? leadingCoefficient p
monic?(p:UP):Boolean == (leadingCoefficient p) = 1

unvectorise(v: Vector R): UP ==
p : UP := 0
for i in 1..#v repeat p := p + monomial(v(i),(i-1)::N)
p

reverse(p:UP):UP ==
r : UP := 0
n := degree(p)
for i in 0..n repeat r := r + monomial(coefficient(p,(n-i)::N),i)
r

scaleRoots(p:UP,c:R):UP ==
-- one? c => p
(c = 1) => p
n := degree p
zero? c => monomial(leadingCoefficient p,n)
r : UP := 0
mc : R := 1
for i in n..0 by -1 repeat
r := r + monomial(mc*coefficient(p,i),i)
mc := mc*c
r

import UnivariatePolynomialCategoryFunctions2(R,UP,UP,
SparseUnivariatePolynomial UP)

shiftRoots(p:UP,c:R):UP == elt(map(coerce,p),monomial(1,1)$UP-c::UP)::UP
package GALUTIL GaloisGroupUtilities

--- GaloisGroupUtilities.input ---

)set break resume
)sys rm -f GaloisGroupUtilities.output
)spool GaloisGroupUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaloisGroupUtilities
--R
--R GaloisGroupUtilities(R: Ring) is a package constructor
--R Abbreviation for GaloisGroupUtilities is GALUTIL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GALUTIL
--R
--R------------------------------- Operations --------------------------------
--R fillPascalTriangle : () -> Void
--R pascalTriangle : (NonNegativeInteger,Integer) -> R
--R rangePascalTriangle : NonNegativeInteger -> NonNegativeInteger
--R rangePascalTriangle : () -> NonNegativeInteger
--R safeCeiling : R -> Integer if R has FPS
--R safeFloor : R -> Integer if R has FPS
--R safetyMargin : NonNegativeInteger -> NonNegativeInteger if R has FPS
--R safetyMargin : () -> NonNegativeInteger if R has FPS
--R sizePascalTriangle : () -> NonNegativeInteger
--R
--E 1

)spool
)lisp (bye)

--- GaloisGroupUtilities.help ---

====================================================================
GaloisGroupUtilities examples
====================================================================

GaloisGroupUtilities provides several useful functions.

See Also:
  o )show GaloisGroupUtilities
GaloisGroupUtilities (GALUTIL)

Exports:
fillPascalTriangle pascalTriangle rangePascalTriangle safeCeiling safeFloor
safetyMargin sizePascalTriangle

— package GALUTIL GaloisGroupUtilities —

)abbrev package GALUTIL GaloisGroupUtilities
++ Author: Frederic Lehobey
++ Date Created: 29 June 1994
++ Date Last Updated: 30 June 1994
++ Description:
++ \spadtype{GaloisGroupUtilities} provides several useful functions.

GaloisGroupUtilities(R): Exports == Implementation where
   N ==> NonNegativeInteger
   Z ==> Integer
   R : Ring

Exports ==> with
   pascalTriangle: (N,Z) -> R
   rangePascalTriangle: N -> N
   rangePascalTriangle: () -> N
   sizePascalTriangle: () -> N
   ++ pascalTriangle(n,r) returns the binomial coefficient
   ++ \spad{C(n,r)=n!/(r! (n-r)!)}
   ++ and stores it in a table to prevent recomputation.
   ++ rangePascalTriangle(n) sets the maximal number of lines which
   ++ are stored and returns the previous value.
   ++ rangePascalTriangle() returns the maximal number of lines stored.
   ++ sizePascalTriangle() returns the number of entries currently stored
   ++ in the table.
fillPascalTriangle: () -> Void
++ fillPascalTriangle() fills the stored table.

if R has FloatingPointSystem then
safeCeiling: R -> Z
++ safeCeiling(x) returns the integer which is greater than any integer
++ with the same floating point number representation.
safeFloor: R -> Z
++ safeFloor(x) returns the integer which is lower or equal to the
++ largest integer which has the same floating point number
++ representation.
safetyMargin: N -> N
++ safetyMargin(n) sets to n the number of low weight digits we do not
++ trust in the floating point representation and returns the previous
++ value (for use by \spadfun{safeCeiling}).
safetyMargin: () -> N
++ safetyMargin() returns the number of low weight digits we do not
++ trust in the floating point representation (used by
++ \spadfun{safeCeiling}).

Implementation ==> add

if R has FloatingPointSystem then
safetymargin : N := 6

safeFloor(x:R):Z ==
if (shift := order(x)-precision(R)+safetymargin) >= 0 then
  x := x+float(1,shift)
retract(floor(x))@Z

safeCeiling(x:R):Z ==
if (shift := order(x)-precision(R)+safetymargin) >= 0 then
  x := x+float(1,shift)
retract(ceiling(x))@Z

safetyMargin(n:N):N ==
  (safetymargin,n) := (n,safetymargin)
n
safetyMargin():N == safetymargin

pascaltriangle : FlexibleArray(R) := empty()
ncomputed : N := 3
rangepascaltriangle : N := 216

pascalTriangle(n:N, r:Z):R ==
negative? r => 0
d := n-r < r => pascalTriangle(n,d)
zero? r => 1$R
one? r => n :: R
(r = 1) => n :: R
n > rangePascalTriangle =>
binomial(n,r)$IntegerCombinatoricFunctions(Z) :: R
n <= ncomputed =>
m := divide(n-4,2)
mq := m.quotient
pascaltriangle((mq+1)*(mq+m.remainder)+r-1)
-- compute the missing lines
for i in (ncomputed+1)..n repeat
  for j in 2..(i quo 2) repeat
    pascaltriangle := concat!(pascaltriangle,pascalTriangle((i-1) :: N, j-1)+pascalTriangle((i-1) :: N,j))
ncomputed := i
pascalTriangle(n,r)

rangePascalTriangle(n:N):N ==
  if n<ncomputed then
    if n<3 then
      pascaltriangle := delete!(pascaltriangle,1..#pascaltriangle)
ncomputed := 3
    else
      d := divide(n-3,2)
dq := d.quotient
      pascaltriangle := delete!(pascaltriangle,((dq+1)*(dq+d.remainder) +1)..#pascaltriangle)
ncomputed := n
    (rangePascalTriangle,n) := (n,rangePascalTriangle)
  n

rangePascalTriangle():N == rangePascalTriangle

sizePascalTriangle():N == #pascaltriangle

fillPascalTriangle():Void == pascalTriangle(rangepascaltriangle,2)

---

— GALUTIL.dotabb —

"GALUTIL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GALUTIL"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"GALUTIL" -> "A1AGG"

---
package GAUSSFAC GaussianFactorizationPackage

— GaussianFactorizationPackage.input —

)set break resume
)sys rm -f GaussianFactorizationPackage.output
)spool GaussianFactorizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GaussianFactorizationPackage
--R
--R GaussianFactorizationPackage is a package constructor
--R Abbreviation for GaussianFactorizationPackage is GAUSSFAC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GAUSSFAC
--R
--R------------------------------- Operations --------------------------------
--R prime? : Complex(Integer) -> Boolean
--R factor : Complex(Integer) -> Factored(Complex(Integer))
--R sumSquares : Integer -> List(Integer)
--R
--E 1

)spool
)lisp (bye)

— GaussianFactorizationPackage.help —

======================================== GaussianFactorizationPackage examples
========================================

Package for the factorization of complex or gaussian integers.

See Also:
 o )show GaussianFactorizationPackage
GaussianFactorizationPackage (GAUSSFAC)

Exports:
factor  prime?  sumSquares

— package GAUSSFAC GaussianFactorizationPackage —

)abbrev package GAUSSFAC GaussianFactorizationPackage
++ Author: Patrizia Gianni
++ Date Created: Summer 1986
++ Description:
++ Package for the factorization of complex or gaussian integers.

GaussianFactorizationPackage() : C == T
where
  NNI == NonNegativeInteger
  Z  ==> Integer
  ZI ==> Complex Z
  FRZ ==> Factored ZI
  fUnion =>> Union("nil", "sqfr", "irred", "prime")
  FFE =>> Record(flg:fUnion, fctr:ZI, xpnt:Integer)

C == with
  factor  :  ZI  ->  FRZ
    ++ factor(zi) produces the complete factorization of the complex
    ++ integer zi.
  sumSquares  :  Z  ->  List Z
    ++ sumSquares(p) construct \spad{a} and \spad{b} such that \spad{a**2+b**2}
    ++ is equal to
    ++ the integer prime p, and otherwise returns an error.
    ++ It will succeed if the prime number p is 2 or congruent to 1
    ++ mod 4.
  prime?  :  ZI  ->  Boolean
    ++ prime?(zi) tests if the complex integer zi is prime.

T == add
  import IntegerFactorizationPackage Z
reduction(u:Z,p:Z):Z ==
  p=0 => u
  positiveRemainder(u,p)

merge(p:Z,q:Z):Union(Z,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

  p=0 => u exquo v
  v rem p = 0 => "failed"
  positiveRemainder((extendedEuclidean(v,p,u)::Record(coef1:Z,coef2:Z)).coef1,p)

FMod := ModularRing(Z,Z,reduction,merge,exactquo)

fact2:ZI:= complex(1,1)

findelt(q:Z) : Z ==
  q1:=q-1
  r:=q1
  r1:=r exquo 4
  while ^(r1 case "failed") repeat
    r:=r1::Z
    r1:=r exquo 2
  s : FMod := reduce(1,q)
  qq1:FMod :=reduce(q1,q)
  for i in 2.. while (s=1 or s=qq1) repeat
    s:=reduce(i,q)**(r::NNI)
  t:=s
  while t^=qq1 repeat
    s:=t
    t:=t**2
  s::Z

sumsq1(p:Z) : List Z ==
  s:= findelt(p)
  u:=p
  while u**2>p repeat
    w:=u rem s
    u:=s
    s:=w
  [u,s]

factorization of an integer
intfactor(n:Z) : Factored ZI ==
    lfn:= factor n
    r : List FFE :=[]
    unity:ZI:=complex(unit lfn,0)
    for term in (factorList lfn) repeat
        n:=term.fctr
        exp:=term.xpnt
        n=2 =>
            r :=concat(["prime",fact2,2*exp]$FFE,r)
            unity:=unity*complex(0,-1)**(exp rem 4)::NNI
        (n rem 4) = 3 => r:=concat(["prime",complex(n,0),exp]$FFE,r)
    sz:=sumsq1(n)
    z:=complex(sz.1,sz.2)
    r:=concat(["prime",z,exp]$FFE,
                concat(["prime",conjugate(z),exp]$FFE,r))
    makeFR(unity,r)

    ---- factorization of a gaussian number ----
factor(m:ZI) : FRZ ==
    m=0 => primeFactor(0,1)
    a:= real m
    (b:= imag m)=0 => intfactor(a) :: FRZ
    a=0 =>
        ris:=intfactor(b)
        unity:= unit(ris)*complex(0,1)
        makeFR(unity,factorList ris)
    d:=gcd(a,b)
    result : List FFE :=[]
    unity:ZI:=1$ZI
    if d^=1 then
        a:=(a exquo d)::Z
        b:=(b exquo d)::Z
        r:= intfactor(d)
        result:=factorList r
        unity:=unit r
        m:=complex(a,b)
    n:=a**2+b**2
    factn:= factorList(factor n)
    part:FFE:=["prime",0$ZI,0]
    for term in factn repeat
        n:=term.fctr
        exp:=term.xpnt
        n=2 =>
part := ["prime", fact2, exp] $FFE
m := m quo (fact2 ** exp: NNI)
result := concat(part, result)

(n rem 4) = 3 =>
g0 := complex(n, 0)
part := ["prime", g0, exp quo 2] $FFE
m := m quo g0
result := concat(part, result)

z := gcd(m, complex(n, 0))
part := ["prime", z, exp] $FFE
z := z ** (exp: NNI)
m := m quo z
result := concat(part, result)

if m = 1 then unity := unity * m
makeFR(unity, result)

----- write p prime like sum of two squares -----

sumSquares(p: Z) : List Z ==
p = 2 => [1, 1]
p rem 4 = 1 => error "no solutions"
sumsq1(p)

prime?(a: ZI) : Boolean ==
n := Z := norm a
n = 0 => false -- zero
n = 1 => false -- units
prime?(n)$IntegerPrimesPackage(Z) => true
re := Z := real a
im := Z := imag a
re = 0 and im = 0 => false
p := Z := abs(re + im) -- a is of the form p, -p, %i*p or -%i*p
p rem 4 = 3 => false
-- return-value true, if p is a rational prime,
-- and false, otherwise
prime?(p)$IntegerPrimesPackage(Z)

— GAUSSFAC.dotabb —
package GHENSEL GeneralHenselPackage

-- GeneralHenselPackage.input --

)set break resume
)sys rm -f GeneralHenselPackage.output
)spool GeneralHenselPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralHenselPackage
--R
--R GeneralHenselPackage(RP: EuclideanDomain,TP: UnivariatePolynomialCategory(RP)) is a package constructor
--R Abbreviation for GeneralHenselPackage is GHENSEL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GHENSEL
--R
--R----------------------------------- Operations -----------------------------------
--R reduction : (TP,RP) -> TP
--R HenselLift : (TP,List(TP),RP,PositiveInteger) -> Record(plist: List(TP),modulo: RP)
--R completeHensel : (TP,List(TP),RP,PositiveInteger) -> List(TP)
--R
--E 1

)spool
)lisp (bye)

---

-- GeneralHenselPackage.help --

====================================================================
GeneralHenselPackage examples
====================================================================

Used for Factorization of bivariate polynomials over a finite field.

See Also:
  o )show GeneralHenselPackage
**GeneralHenselPackage (GHENSEL)**

Exports:
- `completeHensel`
- `HenselLift`
- `reduction`

--- package GHENSEL GeneralHenselPackage ---

```lisp
)abbrev package GHENSEL GeneralHenselPackage
++ Author : P.Gianni
++ Description:
++ General Hensel Lifting
++ Used for Factorization of bivariate polynomials over a finite field.

GeneralHenselPackage(RP,TP):C == T where
  RP : EuclideanDomain
  TP : UnivariatePolynomialCategory RP

  PI ==> PositiveInteger

  C == with
    HenselLift: (TP,List(TP),RP,PI) -> Record(plist:List(TP), modulo:RP)
    ++ HenselLift(pol,lfacts,prime,bound) lifts lfacts,
    ++ that are the factors of pol mod prime,
    ++ to factors of pol mod prime**k > bound. No recombining is done .

    completeHensel: (TP,List(TP),RP,PI) -> List TP
    ++ completeHensel(pol,lfact,prime,bound) lifts lfact,
    ++ the factorization mod prime of pol,
    ++ to the factorization mod prime**k>bound.
    ++ Factors are recombined on the way.

    reduction : (TP,RP) -> TP
    ++ reduction(u,pol) computes the symmetric reduction of u mod pol

  T == add
    GenExEuclid: (List(FP),List(FP),FP) -> List(FP)
    HenselLift1: (TP,List(TP),List(FP),List(FP),RP,RP,F) -> List(TP)
```
mQuo: (TP, RP) -> TP

reduceCoef(c: RP, p: RP): RP ==
    zero? p => c
    RP is Integer => symmetricRemainder(c, p)
    c rem p

reduction(u: TP, p: RP): TP ==
    zero? p => u
    RP is Integer => map(x +-> symmetricRemainder(x, p), u)
    map(x +-> x rem p, u)

merge(p: RP, q: RP): Union(RP, "failed") ==
    p = q => p
    p = 0 => q
    q = 0 => p
    "failed"

modInverse(c: RP, p: RP): RP ==
    (extendedEuclidean(c, p, 1):: Record(coef1: RP, coef2: RP)).coef1

exactquo(u: TP, v: TP, p: RP): Union(TP, "failed") ==
    invlcv := modInverse(leadingCoefficient v, p)
    r := monicDivide(u, reduction(invlcv*v, p))
    reduction(r.remainder, p) ^= 0 => "failed"
    reduction(invlcv*r.quotient, p)

FP := EuclideanModularRing(RP, TP, RP, reduction, merge, exactquo)

mQuo(poly: TP, n: RP) : TP == map(x +-> x quo n, poly)

GenExEuclid(fl: List FP, cl: List FP, rhs: FP) : List FP ==
    [clp*rhs rem flp for clp in cl for flp in fl]

-- generate the possible factors
genFact(fln: List TP, factlist: List List TP) : List List TP ==
    factlist=[] => [[pol] for pol in fln]
    maxd := +/[degree f for f in fln] quo 2
    auxfl: List List TP := []
    for poly in fln while factlist=[] repeat
        factlist := [term for term in factlist | "member?(poly, term)]
        dp := degree poly
        for term in factlist repeat
            (+/[degree f for f in term]) + dp > maxd => "next term"
            auxfl := cons(cons(poly, term), auxfl)
        auxfl

HenselLift1(poly: TP, fln: List TP, f1l: List FP, cl1: List FP,
    lcp := leadingCoefficient poly
rhs := reduce(mQuo(poly - lcp */fln,Modulus),prime)
zero? rhs => fln
lcinv:=reduce(cinv::TP,prime)
vl := GenExEuclid(fl1,c1l,lcinv*rhs)
[flp + Modulus*(vlp::TP) for flp in fln for vlp in vl]

HenselLift(poly:TP,tll:List TP,prime:RP,bound:PI) ==
  -- convert tll
  constp:TP:=0
  if degree first tll = 0 then
    constp:=tll.first
  tll := rest tll
  fl1:=[reduce(ttl,prime) for ttl in tll]
  cl1 := multiEuclidean(fl1,1)::List FP
  Modulus:=prime
  fln :List TP := [ffl1::TP for ffl1 in fl1]
  lcinv:RP:=retract((inv
  (reduce((leadingCoefficient poly)::TP,prime)))::TP)
  while euclideanSize(Modulus)<bound repeat
    nfln:=HenselLift1(poly,fln,fl1,cl1,prime,Modulus,lcinv)
    fln = nfln and zero?(err:=poly-*/fln) => leave "finished"
    fln := nfln
    Modulus := prime*Modulus
  if constp^=0 then fln:=cons(constp,fln)
  [fln,Modulus]

completeHensel(m:TP,tll:List TP,prime:RP,bound:PI) ==
  hlift:=HenselLift(m,tll,prime,bound)
  Modulus:RP:=hlift.modulo
  fln:List TP:=hlift.plist
  nm := degree m
  u:Union(TP,"failed")
  aux,auxl,finallist:List TP
  auxfl,factlist:List List TP
  factlist := []
  dfn :NonNegativeInteger := nm
  lcm1 := leadingCoefficient m
  mm := lcm1*m
  while dfn>0 and (factlist := genFact(fln,factlist))^=[] repeat
    auxfl := []
    while factlist"=[] repeat
      auxl := factlist.first
      factlist := factlist.rest
      tc := reduceCoef((lcm1 * *[coefficient(poly,0)
        for poly in auxl]), Modulus)
      coefficient(mm,0) exquo tc case "failed" =>
        auxfl := cons(auxl,auxfl)
      pol := */[poly for poly in auxl]
      poly :=reduction(lcm1*pol,Modulus)
      u := mm exquo poly
u case "failed" => auxfl := cons(auxl,auxfl)
poly1: TP := primitivePart poly
m := mQuo((u::TP),leadingCoefficient poly1)
lcm1 := leadingCoefficient(m)
mm := lcm1 *m
finallist := cons(poly1,finallist)
dfn := degree m
aux := []
for poly in fln repeat
  ^member?(poly,auxl) => aux := cons(poly,aux)
  auxfl := [term for term in auxfl | ^member?(poly,term)]
  factlist := [term for term in factlist | ^member?(poly,term)]
  fln := aux
  factlist := auxfl
if dfn > 0 then finallist := cons(m,finallist)
finallist

package GENMFACT GeneralizedMultivariateFactorize

)set break resume
)sys rm -f GeneralizedMultivariateFactorize.output
)spool GeneralizedMultivariateFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralizedMultivariateFactorize
--R
--R GeneralizedMultivariateFactorize(OV,E: OrderedAbelianMonoidSup,S: IntegralDomain,R: IntegralDomain,P: PolynomialCategory(R,E,OV))where
--R OV: OrderedSet with
--R convert : % -> Symbol
--R variable : Symbol -> Union(%,"failed") is a package constructor
--R Abbreviation for GeneralizedMultivariateFactorize is GENMFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GENMFACT
--R
--R----------------------------------- Operations -----------------------------------
--R factor : P -> Factored(P)
--R

;spool
;lisp (bye)

— GeneralizedMultivariateFactorize.help —

====================================================================
GeneralizedMultivariateFactorize examples
====================================================================

This is the top level package for doing multivariate factorization
over basic domains like Integer or Fraction Integer.

See Also:
o )show GeneralizedMultivariateFactorize

GeneralizedMultivariateFactorize (GENMFACT)

Exports:
factor
package GENMFACT GeneralizedMultivariateFactorize

)abbrev package GENMFACT GeneralizedMultivariateFactorize
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Description:
++ This is the top level package for doing multivariate factorization
++ over basic domains like \texttt{Integer} or \texttt{Fraction Integer}.

GeneralizedMultivariateFactorize(OV,E,S,R,P) : C == T
where
  R : IntegralDomain
    -- with factor on \texttt{R}[x]
  S : IntegralDomain
  OV : OrderedSet with
      convert : % -> Symbol
          ++ convert(x) converts x to a symbol
      variable: Symbol -> Union(%, "failed")
          ++ variable(s) makes an element from symbol s or fails.
  E : OrderedAbelianMonoidSup
  P : PolynomialCategory(R,E,OV)

C == with
  factor : P -> Factored P
    ++ factor(p) factors the multivariate polynomial p over its coefficient
    ++ domain

T == add
  factor(p:P) : Factored P ==
    R has FiniteFieldCategory => factor(p)$MultFiniteFactorize(OV,E,R,P)
    R is Polynomial(S) and S has EuclideanDomain =>
      factor(p)$MPolyCatPolynomialFactorizer(E,OV,S,P)
    R is Fraction(S) and S has CharacteristicZero and
      S has EuclideanDomain =>
      factor(p)$MRationalFactorize(E,OV,S,P)
    R is Fraction Polynomial S =>
      factor(p)$MPolyCatRationalFunctionFactorizer(E,OV,S,P)
    R has CharacteristicZero and R has EuclideanDomain =>
      factor(p)$MultivariateFactorize(OV,E,R,P)
  squareFree p

---

GENMFACT.dotabb

"GENMFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENMFACT"]
package GPAFF GeneralPackageForAlgebraicFunctionField

— GeneralPackageForAlgebraicFunctionField.input —

)set break resume
)sys rm -f GeneralPackageForAlgebraicFunctionField.output
)spool GeneralPackageForAlgebraicFunctionField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GeneralPackageForAlgebraicFunctionField

--R
--R GeneralPackageForAlgebraicFunctionField(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory) is a package constructor
--R Abbreviation for GeneralPackageForAlgebraicFunctionField is GPAFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GPAFF
--R
--R------------------------------- Operations --------------------------------
--R adjunctionDivisor : () -> DIVISOR  desingTree : () -> List(DesTree)
--R eval : (PolyRing,Plc) -> K  eval : (PolyRing,PolyRing,Plc) -> K
--R eval : (Fraction(PolyRing),Plc) -> K  genus : () -> NonNegativeInteger
--R genusNeg : () -> Integer  parametrize : (PolyRing,Plc) -> PCS
--R placesAbove : ProjPt -> List(Plc)  pointDominates : Plc -> ProjPt
--R printInfo : List(Boolean) -> Void  reset : () -> Void
--R setCurve : PolyRing -> PolyRing  singularPoints : () -> List(ProjPt)
--R theCurve : () -> PolyRing
--R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if K has Finite
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if K has Finite
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if K has Finite
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if K has Finite
--R classNumber : () -> Integer if K has Finite
--R desingTreeWithFullParam : () -> List(DesTree)
--R evalIfCan : (PolyRing,Plc) -> Union(K,"failed")
--R evalIfCan : (Fraction(PolyRing),Plc) -> Union(K,"failed")
--R findOrderOfDivisor : (DIVISOR,Integer,Integer) -> Record(ord: Integer,num: PolyRing,den: PolyRing)
--R homogenize : (PolyRing,Integer) -> PolyRing
--R interpolateForms : (DIVISOR,NonNegativeInteger) -> List(PolyRing)
--R interpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(PolyRing)
--R intersectionDivisor : PolyRing -> DIVISOR
--R lBasis : DIVISOR -> Record(num: List(PolyRing),den: PolyRing)
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if K has FINITE
--R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if K has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if K has FINITE
--R placesOfDegree : PositiveInteger -> List(Plc) if K has FINITE
--R rationalPlaces : () -> List(Plc) if K has FINITE
--R rationalPoints : () -> List(ProjPt) if K has FINITE
--R setSingularPoints : List(ProjPt) -> List(ProjPt)
--R
--E 1

)spool
)lisp (bye)

---

<table>
<thead>
<tr>
<th>GeneralPackageForAlgebraicFunctionField.help</th>
</tr>
</thead>
</table>

GeneralPackageForAlgebraicFunctionField examples

A package that implements the Brill-Noether algorithm.
Part of the PAFF package.

See Also:
- )show GeneralPackageForAlgebraicFunctionField

---

GeneralPackageForAlgebraicFunctionField (GPAFF)
Exports:
adjunctionDivisor  classNumber  desingTree
desingTreeWoFullParam  eval  evalIfCan
findOrderOfDivisor  genus  genusNeg
homogenize  interpolateForms  interpolateFormsForFact
intersectionDivisor  lBasis  LPolynomial
numberOfPlacesOfDegree  numberPlacesDegExtDeg  numberRatPlacesExtDeg
parametrize  placesAbove  placesOfDegree
pointDominatesBy  printInfo  rationalPlaces
rationalPoints  reset  setCurve
setSingularPoints  singularPoints  theCurve
ZetaFunction

— package GPAFF GeneralPackageForAlgebraicFunctionField —

)abbrev package GPAFF GeneralPackageForAlgebraicFunctionField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that implements the Brill-Noether algorithm. Part of the
++ PAFF package.
GeneralPackageForAlgebraicFunctionField( K, symb, PolyRing, E, ProjPt, PCS, Plc, DIVISOR, InfClsPoint, DesTree, BLMET ) :Exports == Implementation where

K:Field
symb: List(Symbol)
OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)
InfClsPoint : InfinitlyClosePointCategory(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,DesTree,BLMET)
DesTree : DesingTreeCategory(InfClsPoint)
BLMET : BlowUpMethodCategory

FRACPOLY ==> Fraction PolyRing
PACKAGE GPAFF GENERALPACKAGEFORALGEBRAICFUNCTIONFIELD

OF ==> OutputForm
INT ==> Integer
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
UP ==> UnivariatePolynomial
UPZ ==> UP(t, Integer)
UTSZ ==> UnivariateTaylorSeriesCZero(Integer,t)
SUP ==> SparseUnivariatePolynomial
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)

ParamPackFC ==> LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,_,
E,ProjPt,PCS,Plc)
ParamPack ==> ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
RatSingPack ==> ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt)
IntDivPack ==> IntersectionDivisorPackage(K,symb,PolyRing,E,ProjPt,PCS,_,
Plc,DIVISOR,InfClsPoint,DesTree,BLMET)
IntFrmPack ==> InterpolateFormsPackage(K,symb,PolyRing,E,ProjPt,PCS,_,
Plc,DIVISOR)
DesTrPack ==> DesingTreePackage(K,symb,PolyRing,E,ProjPt,PCS,_,
Plc,DIVISOR,InfClsPoint,DesTree,BLMET)
PackPoly ==> PackageForPoly(K,PolyRing,E,#symb)

Exports ==> with
reset: () -> Void
setCurve: PolyRing -> PolyRing
homogenize: (PolyRing,Integer) -> PolyRing
printInfo: List Boolean -> Void
  ++ printInfo(lbool) prints some information comming from various
  ++ package and domain used by this package.
theCurve: () -> PolyRing
  ++ theCurve returns the specified polynomial for the package.
genus: () -> NNI
  ++ genus returns the genus of the curve defined by the polynomial
  ++ given to the package.
genusNeg: () -> INT
desingTree: () -> List DesTree
  ++ desingTree returns the desingularisation trees at all singular
  ++ points of the curve defined by the polynomial given to the package.
desingTreeWoFullParam : () -> List DesTree
  ++ desingTreeWoFullParam returns the desingularisation trees at all

++ singular points of the curve defined by the polynomial given to +
++ the package. The local parametrizations are not computed.

setSingularPoints: List ProjPt -> List ProjPt
++ setSingularPoints(lpt) sets the singular points to be used.
++ Beware: no attempt is made to check if the points are singular
++ or not, nor if all of the singular points are presents. Hence,
++ results of some computation maybe false. It is intend to be use
++ when one want to compute the singular points are computed by other
++ means than to use the function singularPoints.

singularPoints: () -> List(ProjPt)
++ rationalPoints() returns the singular points of the
++ curve defined by the polynomial given to the package.
++ If the singular points lie in an extension of the specified ground
++ field an error message is issued specifying the extension degree
++ needed to find all singular points.

parametrize: (PolyRing,Plc) -> PCS
++ parametrize(f,pl) returns a local parametrization of f at the place
++ pl.

lBasis: DIVISOR -> Record(num:List PolyRing, den: PolyRing)
++ lBasis computes a basis associated to the specified divisor

findOrderOfDivisor: (DIVISOR,Integer,Integer) -> _
++ findOrderOfDivisor(d,n) returns the order of the given divisor d.

interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
++ interpolateForms(d,n) returns a basis of the interpolate forms of
++ degree n of the divisor d.

interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

eval: (PolyRing,Plc) -> K
++ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing,PolyRing,Plc) -> K
++ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FRACPOLY,Plc) -> K
++ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,pl) evaluate f at the place pl
++ (returns "failed" if it is a pole).

evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
++ (returns "failed" if it is a pole).
evalIfCan: (FRACPOLY,Plc) -> Union(K,"failed")
++ evalIfCan(u,pl) evaluate the function u at the place pl
++ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
++ intersectionDivisor(pol) compute the intersection divisor
++ (the Cartier divisor) of the form pol with the curve. If some
++ intersection points lie in an extension of the ground field,
++ an error message is issued specifying the extension degree
++ needed to find all the intersection points.
++ (If pol is not homogeneous an error message is issued).

adjunctionDivisor: () -> DIVISOR
++ adjunctionDivisor computes the adjunction divisor of the plane
++ curve given by the polynomial crv.

placesAbove: ProjPt -> List Plc

pointDominateBy : Plc -> ProjPt
++ pointDominateBy(pl) returns the projective point dominated
++ by the place pl.

if K has Finite then --should we say LocallyAlgebraicallyClosedField??

rationalPlaces: () -> List Plc
++ rationalPlaces returns all the rational places of the
++ curve defined by the polynomial given to the package.

rationalPoints: () -> List(ProjPt)

LPolynomial: () -> SparseUnivariatePolynomial Integer
++ Returns the L-Polynomial of the curve.

LPolynomial: PI -> SparseUnivariatePolynomial Integer
++ LPolynomial(d) returns the L-Polynomial of the curve in
++ constant field extension of degree d.

classNumber: () -> Integer
++ Returns the class number of the curve.

placesOfDegree: PI -> List Plc
++ placesOfDegree(d) returns all places of degree d of the
++ curve.

numberOfPlacesOfDegree: PI -> Integer
++ returns the number of places of the given degree

numberRatPlacesExtDeg: PI -> Integer
++ numberRatPlacesExtDeg(n) returns the number of rational
numberPlacesDegExtDeg: (PI, PI) -> Integer
  ++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
  ++ places of degree d in the constant field extension of
  ++ degree n

ZetaFunction: () -> UTSZ
  ++ Returns the Zeta function of the curve. Calculated by
  ++ using the L-Polynomial

ZetaFunction: PI -> UTSZ
  ++ Returns the Zeta function of the curve in constant field
  ++ extension. Calculated by using the L-Polynomial

Implementation ==> add
import PPFC1
import PPFC2
import DesTrPack
import IntFrmPack
import IntDivPack
import RatSingPack
import ParamPack
import ParamPackFC
import PackPoly

crvLocal:PolyRing:=1$PolyRing

-- flags telling such and such is already computed.
genusCalc?:Boolean:= false()$Boolean
theGenus:INT:=0
desingTreeCalc?:Boolean:=false()$Boolean
theTree:List DesTree := empty()
desingTreeWoFullParamCalc?:Boolean:=false()$Boolean
adjDivCalc?:Boolean:=false()$Boolean
theAdjDiv:DIVISOR:=0
singularPointsCalc?:Boolean:=false()$Boolean
lesPtsSing:List(ProjPt):=empty()
rationalPointsCalc?:Boolean:=false()$Boolean
lesRatPts:List(ProjPt):=empty()
rationalPlacesCalc?:Boolean:=false()$Boolean
lesRatPlcs:List(Plc):=empty()
zf:UTSZ:=1$UTSZ
zfCalc : Boolean := false()$Boolean

DegOfPlacesFound: List Integer := empty()

-- see package IntersectionDivisorPackage
intersectionDivisor(pol)==
  if ~ (pol =$PolyRing homogenize(pol,1)) then _
    error _
    "From intersectionDivisor: the input is NOT a homogeneous polynomial"
  intersectionDivisor(pol,theCurve(),desingTree(),singularPoints())

lBasis(divis)==
d:=degree divis
d < 0 => [[0$PolyRing],1$PolyRing]
A:=adjunctionDivisor()
-- modifie le 08/05/97: avant c’était formToInterp:=divOfZero(divis) + A
formToInterp:= divOfZero(divis + A)
degDpA:=degree formToInterp
degCrb:=totalDegree(theCurve())$PackPoly
dd:=divide(degDpA,degCrb pretend Integer)
dmin:NNI:=
  if ~ zero?(dd.remainder) then (dd.quotient+1) pretend NNI
  else dd.quotient pretend NNI
print("Trying to interpolate with forms of degree:"::OF)
print(dmin::OF)
lg0:List PolyRing:=interpolateForms(formToInterp,dmin)
while zero?(first lg0) repeat
  dmin:=dmin+1
  print("Trying to interpolate with forms of degree:"::OF)
  print(dmin::OF)
lg0:=interpolateForms(formToInterp,dmin)
print("Denominator found":OF)
g0:PolyRing:=first lg0
dg0:=intersectionDivisor(g0)
print("Intersection Divisor of Denominator found":OF)
lnumer:List PolyRing:=interpolateForms(dg0-divis,dmin)
[lnumer,g0]

genus==
  if ~(genusCalc?) then
    degCrb:=totalDegree(theCurve())$PackPoly
    theGenus:=genusTreeNeg(degCrb,desingTreeWoFullParam())
    genusCalc?:=true()$Boolean
  theGenus < 0 =>
    print("Too many infinitly near points":OF)
    print("The curve may not be absolutely irreducible":OF)
    error "Have a nice day"
  theGenus pretend NNI
CHAPTER 8. CHAPTER G

\begin{verbatim}

genusNeg==
  if ^(genusCalc?) then
    degCrb:=totalDegree(theCurve())$PackPoly
    theGenus:=genusTreeNeg(degCrb,desingTreeWoFullParam())
    genusCalc?:=true()$Boolean
    theGenus

homogenize(pol,n)== homogenize(pol,n)$PackPoly

fPl(pt:ProjPt,desTr:DesTree):Boolean ==
  nd:=value desTr
  lpt:=pointV nd
  pt = lpt

placesAbove(pt)==
  -- verifie si le point est simple, s\'i s\'est le cas,
  -- on retourne la place correspondante
  -- avec pointToPlace qui cre\'e la place si necessaire.
  "member?(pt,singularPoints()) => _
    [pointToPlace(pt,theCurve())$ParamPackFC]
  -- les quatre lignes suivantes trouvent les feuilles qui
  -- sont au-dessus du point.
  theTree:= desingTree()
  cTree:= find(fPl(pt,#1),theTree)
  cTree case "failed" => error "Big error in placesAbove"
  -- G. Hache, gaetan.hache@inria.fr"
  lvs:=leaves cTree
  -- retourne les places correspondant aux feuilles en "consultant"
  -- les diviseurs exceptionnels.
  concat [supp excpDivV(l) for l in lvs]

pointDominateBy(pl)== pointDominateBy(pl)$ParamPackFC

reduceForm(p1:PolyRing,p2:PolyRing):PolyRing==
  normalForm(p1,[p2])$GroebnerPackage(K,E,OV,PolyRing)

evalIfCan(f:PolyRing,pl:P1c)==
  u:=reduceForm(f, theCurve() )
  zero?(u) => 0
  pf:= parametrize(f,pl)
  ord:INT:=order pf
  ord < 0 => "failed"
  ord > 0 => 0
  coefOfFirstNonZeroTerm pf

eval(f:PolyRing,pl:P1c)==
  eic:=evalIfCan(f,pl)
  eic case "failed" => _
    error "From eval (function at place): its a pole !!!"
\end{verbatim}


eic

setCurve(pol)==
crvLocal:=pol
"(crvLocal =$PolyRing homogenize(crvLocal,1)) =>
  print("the defining polynomial is not homogeneous")::OF)
  error "Have a nice day"
reset()
theCurve()

reset ==
setFoundPlacesToEmpty()$Plc
genusCalc?:Boolean:= false()$Boolean
theGenus:INT:=0
desingTreeCalc?:Boolean:=false()$Boolean
desingTreeWoFullParamCalc?:Boolean:=false()$Boolean
theTree:List DesTree := empty()
adjDivCalc?:Boolean:=false()$Boolean
theAdjDiv:DIVISOR:=0
singularPointsCalc?:Boolean:=false()$Boolean
lesPtsSing:List(ProjPt):=empty()
rationalPointsCalc?:Boolean:=false()$Boolean
lesRatPts:List(ProjPt):=empty()
rationalPlacesCalc?:Boolean:=false()$Boolean
lesRatPlcs:List(Plc):=empty()
DegOfPlacesFound: List Integer := empty()
zf:UTSZ:=1$UTSZ
zfCalc:Boolean := false$Boolean

foundPlacesOfDeg?(i:PositiveInteger):Boolean ==
  ld: List Boolean := [zero?(a rem i) for a in DegOfPlacesFound]
  entry?(true$Boolean,ld)

findOrderOfDivisor(divis,lb,hb) ==
  "zero?(degree divis) => error("The divisor is NOT of degre zero !!!!")
A:=adjunctionDivisor()
formToInterp:=divOfZero ( hb*divis + A )
degDpA:=degree formToInterp
degCrb:=totalDegree( theCurve())$PackPoly
dd:=divide(degDpA,degCrb pretend Integer)
dmin:NNI:=
  if "zero?(dd.remainder) then (dd.quotient+1) pretend NNI
  else dd.quotient pretend NNI
lg0:List PolyRing:=interpolateForms(formToInterp,dmin)
while zero?(first lg0) repeat
  dmin:=dmin+1
  lg0:=interpolateForms(formToInterp,dmin)
g0:PolyRing:=first lg0
dg0:=intersectionDivisor(g0)
nhb:=hb
while effective?(dg0 - nhb*divis - A) repeat
  nhb:=nhb+1
  nhb:=nhb-1
  ftry:=1b
  lnumer:List PolyRing:=interpolateForms(dg0-ftry*divis,dmin)
  while zero?(first lnumer) and ftry < nhb repeat
    ftry:=ftry + 1
    lnumer:List PolyRing:=interpolateForms(dg0-ftry*divis,dmin)
  [ftry,first lnumer,g0,nhb]

theCurve==
  one?(crvLocal) => error "The defining polynomial has not been set yet!"
  crvLocal

printInfo(lbool)==
  printInfo(lbool.2)$ParamPackFC
  printInfo(lbool.3)$PCS
  void()

desingTree==
  theTree:= desingTreeWoFullParam()
  if ~(desingTreeCalc?) then
    for arb in theTree repeat
      fullParamInit(arb)
      desingTreeCalc?:=true()$Boolean
      theTree

  desingTreeWoFullParam==
  if ~(desingTreeWoFullParamCalc?) then
    theTree:=[desingTreeAtPoint(pt,theCurve()) for pt in singularPoints()]
    desingTreeWoFullParamCalc?:=true()$Boolean
    theTree

-- compute the adjunction divisor of the curve using adjunctionDivisor
-- from DesingTreePackage
adjunctionDivisor()==
  if ~(adjDivCalc?) then
    theAdjDiv:=_
    reduce("+",[adjunctionDivisor(tr) for tr in desingTree()],0$DIVISOR)
    adjDivCalc?:=true()$Boolean
    theAdjDiv

-- returns the singular points using the function singularPoints
-- from ProjectiveAlgebraicSetPackage
singularPoints==
  if ~(singularPointsCalc?) then
    lesPtsSing:=singularPoints(theCurve())
    singularPointsCalc?:=true()$Boolean
    lesPtsSing
setSingularPoints(lspt)==
singularPointsCalc?:=true()$Boolean
lesPtsSing:= lspt

-- returns the rational points using the function rationalPoints
-- from ProjectiveAlgebraicSetPackage

-- compute the local parametrization of f at the place pl
-- (from package ParametrizationPackage)
parametrize(f,pl)==parametrize(f,pl)$ParamPack

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==
lm:List PolyRing:=listAllMono(n)$PackPoly
interpolateForms(d,n,theCurve(),lm)

interpolateFormsForFact(d,lm)==
interpolateFormsForFact(d,lm)$IntFrmPack

evalIfCan(f:PolyRing,g:PolyRing,pl:Plc)==
fu:=reduceForm(f,theCurve())
gu:=reduceForm(g,theCurve())
zero?(fu) and ~zero?(gu) => 0
~zero?(fu) and zero?(gu) => "failed"

pf:= parametrize(fu,pl)
pg:= parametrize(gu,pl)
ordf:INT:=order pf
ordg:INT:=order pg
cf:=coefOfFirstNonZeroTerm pf
cg:=coefOfFirstNonZeroTerm pg
(ordf - ordg) < 0 => "failed"
(ordf - ordg) > 0 => 0

\text{cf} \ast \text{inv cg}

eval(f:PolyRing,g:PolyRing,pl:Plc)==
eic:=evalIfCan(f,g,pl)
eic case "failed" => error "From eval (function at place): its a pole"
eic

evalIfCan(u:FRACPOLY,pl:Plc)==
f:PolyRing := numer u
g:PolyRing := denom u
evalIfCan(f,g,pl)

eval(u:FRACPOLY,pl:Plc)==
f:PolyRing := numer u
g:PolyRing := denom u
eval(f,g,pl)

thedeg:PI := 1
crap(p:Plc):Boolean ==
  degree(p)$Plc = thedeg

if K has Finite then
  rationalPlaces ==
    K has PseudoAlgebraicClosureOfFiniteFieldCategory => _
    placesOfDegree(1$PI)
    -- non good pour LACF !!!!
    rationalPlacesCalc? => lesRatPlcs
    ltr:List(DesTree):=desingTree()
    ratP:List(ProjPt):=rationalPoints()
    singP:List(ProjPt):=singularPoints()
    simRatP:List(ProjPt):=setDifference(ratP,singP)
    for pt in simRatP repeat
      pointToPlace(pt,theCurve())$ParamPackFC
    rationalPlacesCalc? := true()$Boolean
    lesRatPlcs:=foundPlaces()$Plc
    lesRatPlcs

rationalPoints ==
  if ^(rationalPointsCalc?) then
    if K has Finite then
      lesRatPts:= rationalPoints(theCurve(),1)$RatSingPack
      rationalPointsCalc?:=true()$Boolean
    else
      error "Can't find rationalPoints when the field is not finite"
    lesRatPts
  rationalPointsCalc? ==
    if K has Finite then
      lesRatPts:= rationalPoints(theCurve(),1)$RatSingPack
      rationalPointsCalc?:=true()$Boolean
    else
      error "Can't find rationalPoints when the field is not finite"
    lesRatPts

ZetaFunction() ==
  if not zfCalc then
    zf:= ZetaFunction(1)
    zfCalc:= true$Boolean
    zf

ZetaFunction(d) ==
  lp:= LPolynomial(d)
  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)
  q:INT := size()$K ** d
  lpt:UPZ := unmakeSUP(lp)$UPZ
  lps:UTSZ := coerce(lpt)$UTSZ
  x:= monomial(1,1)$UTSZ
  mul: UTSZ := (1-x)*(1 - q * x)
  invmul:Union(UTSZ,"failed") := recip(mul)$UTSZ
  ivm: UTSZ
  if not (invmul case "failed") then
    ivm := invmul pretend UTSZ
  else
    ivm := 1
lps * ivm

calculatedSer: List UTSZ := [1]
   -- in index i is the "almost ZetaFunction" summed to i-1.
   -- Except calculatedSer.1 which is 1

--Except calculatedSer.1 which is 1

numberOfPlacesOfDegreeUsingZeta(degree:PI): Integer ==
   -- is at most called once for each degree. Will calculate the
   -- entries in calculatedSer.
   ser:UTSZ := 1
   x := monomial(1,1)$UTSZ
   pol := (1-x**i) ** (n pretend PI)
   polser := recip(pol)$UTSZ -- coerce(pol)$UTSZ)
   serdel := (polser pretend UTSZ)
   if not (polser case "failed") then
      serdel := (polser pretend UTSZ)
   error "In numberOfPlacesOfDegreeUsingZeta. This shouldn't happen"
   ser := serdel * calculatedSer.i
   calculatedSer := concat(calculatedSer, ser)
   i := i + 1
   if degree = 1 then
      coefficient(ZetaFunction(),degree)
   else
      coefficient(ZetaFunction(),degree) - _
      coefficient(calculatedSer.degree, degree)

   calculatedNP: List Integer := empty()
   -- local variable, in index i is number of places of degree i.

degreeOfPlacesOfDegreeUsingZeta(degree:PI): Integer ==
   if zfCalc then
      if (m := maxIndex(calculatedNP)) < i then
         calculatedNP := _
         concat(calculatedNP, _
            [numberOfPlacesOfDegreeUsingZeta(j pretend PI) _
            for j in ((m+1) pretend PI)...
         calculatedNP.i
   else
      # possibly we should make an improvement in this

      placesOfDegree(i) ==
      if not foundPlacesOfDeg?(i) then
         if characteristic()$K**i > (2**16 - 1) then
print("If you are using a prime field and"":OF)
print("GB this will not work."":OF)
desingTree()
placesOfDegree(i,thecurve(),singularPoints())
DegOfPlacesFound:= concat(DegOfPlacesFound, i)
thedeg:= i
select(crap(#1), foundPlaces()$Plc)

numberRatPlacesExtDeg(extDegree:PI): Integer ==
  numberPlacesDegExtDeg(1,extDegree)

numberPlacesDegExtDeg(degree:PI, extDegree:PI): Integer ==
  res:Integer:=0
m:PI := degree * extDegree
d: PI
while m > 0 repeat
  d:= gcd(m, extDegree)
  if (m quo d) = degree then
    res:= res + (numberOfPlacesOfDegree(m) * d)
  m:= (m - 1) pretend PI
res

calculateS(extDeg:PI): List Integer ==
  g := genus()
  sizeK:NNI := size()$K ** extDeg
  i:PositiveInteger := g pretend PI
  S: List Integer := [0 for j in 1..g]
  good:Boolean := true()$Boolean
  while good repeat
    S.i := numberRatPlacesExtDeg(i*extDeg) - ((sizeK **$NNI i) + 1)
    j:Integer := i - 1
    if (not (j = 0)) then
      i:= (j pretend PI)
    else good:= false()$Boolean
    S

LPolynomial(): SparseUnivariatePolynomial Integer ==
LPolynomial(1)

LPolynomial(extDeg:PI): SparseUnivariatePolynomial Integer ==
--when translating to AxiomXL rewrite this function!
  g := genus()
  zero?(g) => 1
  coef: List Integer := [1]
  if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
    setTower!(1$K)
    sizeK:Integer := size()$K ** extDeg --need to do a setExtension before
    coef:= concat(coef,[0 for j in 1..(2*g)])
  S: List Integer := calculateS(extDeg)
i:PI := 1
tmp: Integer
   while i < g + 1 repeat
      j: PI := 1
      tmp:= 0
      while j < i + 1 repeat
         tmp:= tmp + S.j * coef((i + 1 - j) pretend PI)
         j:= j + 1
      coef.(i+1) := tmp quo i
      i:= i + 1
      i:= 1
      while i < g + 1 repeat
         ss: Integer := sizeK **$Integer ((g + 1 - i) pretend PI)
         val: Integer := ss * coef.i
         coef.((2*g+2 - i) pretend PI) := val
         i:= i + 1
      x:= monomial(1,1)$SUP(INT)
      result: SparseUnivariatePolynomial(Integer):= 
         1$SparseUnivariatePolynomial(Integer)
      coef:= rest(coef)
      i:= 1
      while i < 2 * g + 1 repeat
         pol: SUP(INT) := (first(coef) :: Integer) * (x ** i)
         result:= result + pol --(first(coef) :: Integer) * (x ** i)
         coef:= rest(coef)
         i:= i + 1
      result

   classNumber():Integer ==
      LPolynomial()(1)


```lisp
(set break resume)
(sys rm -f GeneralPolynomialGcdPackage.output)
(spool GeneralPolynomialGcdPackage.output)
(set message test on)
(set message auto off)
clear all

--S 1 of 1
)show GeneralPolynomialGcdPackage
--R
--R GeneralPolynomialGcdPackage(E: OrderedAbelianMonoidSup,OV: OrderedSet,R: PolynomialFactorizationExplicit,P: PolynomialCategory(R,E,OV)) is a package constructor
--R Abbreviation for GeneralPolynomialGcdPackage is GENPGCD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GENPGCD
--R
--R------------------------------- Operations --------------------------------
--R randomR : () -> R
--R gcdPolynomial : (SparseUnivariatePolynomial(P),SparseUnivariatePolynomial(P)) -> SparseUnivariatePolynomial(P)
--E 1

)spool
)lisp (bye)
```

---

**GeneralPolynomialGcdPackage.help**

This package provides operations for GCD computations on polynomials

See Also:
- )show GeneralPolynomialGcdPackage

---
GeneralPolynomialGcdPackage (GENPGCD)

Exports:
  gcdPolynomial randomR

--- package GENPGCD GeneralPolynomialGcdPackage ---

)abbrev package GENPGCD GeneralPolynomialGcdPackage
++ Description:

GeneralPolynomialGcdPackage(E,OV,R,P):C == T where
  R : PolynomialFactorizationExplicit
  P : PolynomialCategory(R,E,OV)
  OV : OrderedSet
  E : OrderedAbelianMonoidSup

  SUPP ==> SparseUnivariatePolynomial P
  --JHD ContPrim ==> Record(cont:P,prim:P)

  C == with
    gcdPolynomial : (SUPP,SUPP) -> SUPP
      ++ gcdPolynomial(p,q) returns the GCD of p and q
    randomR : () -> R
      ++ randomR() should be local but conditional

    gcd : (P,P) -> P
    gcd : List P -> P
    gcdprim : (P,P) -> P
    gcdprim : List P -> P
    gcdcofact : List P -> List P
    gcdcofactprim : List P -> List P
    primitate : (P,OV) -> P
    primitate : SUPP -> SUPP
    content : P -> P
    content : List P -> List P
--JHD  contprim : List P -> List ContPrim
--JHD  monomContent : (P,OV) -> P
--JHD  monomContent : SUPP -> SUPP

T == add

SUPR  ==> SparseUnivariatePolynomial R
--JHD  SUPLGcd ==> Record(locgcd:SUPP,goodint:List R)
--JHD  LGcd   ==> Record(locgcd:P,goodint:List R)
--JHD  UTerm  ==> Record(lpol:List SUPR,lint:List R,mpol:P)
--JHD--JHD  pmod:R := (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

--JHD  import MultivariateLifting(E,OV,R,P,pmod)
import UnivariatePolynomialCategoryFunctions2(R,SUPR,P,SUPP)
import UnivariatePolynomialCategoryFunctions2(P,SUPP,R,SUPR)

-------- Local Functions --------
--JHD  abs : P -> P
better : (P,P) -> Boolean
--JHD  failtest : (P,P,P) -> Boolean
--JHD  gcdMonom : (P,P,OV) -> P
--JHD  gcdTermList : (P,P) -> P
--JHD  gcdPrim : (P,P,OV) -> P
--JHD  gcdSameMainvar : (P,P,OV) -> P
--JHD  internal : (P,P,OV) -> P
--JHD  good : (P,List OV) -> Record(upol:SUPR,inval:List R)
--JHD  gcdPrs : (P,P,NNI,OV) -> Union(P,"failed")
--JHD
--JHD  chooseVal : (P,P,List OV) -> UTerm
--JHD  localgcd : (P,P,List OV) -> LGcd
--JHD  notCoprime : (P,P,List NNI,List OV) -> P
--JHD  imposelc : (List SUPR,List OV,List R,List P) -> List SUPR

--JHD  lift? :(P,P,UTerm,List NNI,List OV) -> Union("failed",P)
--JHD  lift : (P,SUPR,SUPR,P,List OV,List NNI,List R) -> P
lift : (SUPR,SUPP,SUPR,List OV,List R) -> Union(SUPP,"failed")
-- lifts first and third arguments as factors of the second
-- fourth is number of variables.
--JHD  monomContent : (P,OV) -> P

monomContentSup : SUPP -> SUPP

--
--JHD  gcdcofact : List P -> List P

gcdTrivial : (SUPP,SUPP) -> SUPP
gcdSameVariables: (SUPP,SUPP,List OV) -> SUPP
recursivelyGCDCoefficients: (SUPP,List OV,SUPP,List OV) -> SUPP
flatten : (SUPP,List OV) -> SUPP
-- evaluates out all variables in the second
gcdPolynomial(p1:SUPP,p2:SUPP) ==
  zero? p1 => p2
  zero? p2 => p1
  0=degree p1 => gcdTrivial(p1,p2)
  0=degree p2 => gcdTrivial(p2,p1)
if degree p1 < degree p2 then (p1,p2):=(p2,p1)
p1 exquo p2 case SUPP => (unitNormal p2).canonical
  c1:= monomContentSup(p1)
  c2:= monomContentSup(p2)
  p1:= (p1 exquo c1)::SUPP
  p2:= (p2 exquo c2)::SUPP
(p1 exquo p2) case SUPP => (unitNormal p2).canonical * gcd(c1,c2)
vp1:=variables p1
vp2:=variables p2
v1:=setDifference(vp1,vp2)
v2:=setDifference(vp2,vp1)
#v1 = 0 and #v2 = 0 => gcdSameVariables(p1,p2,vp1)*gcd(c1,c2)
-- all variables are in common
v:=setDifference(vp1,v1)
pp1:=flatten(p1,v1)
pp2:=flatten(p2,v2)
g:=gcdSameVariables(pp1,pp2,v)
-- one? g => gcd(c1,c2)::SUPP
  (g = 1) => gcd(c1,c2)::SUPP
  (#v1 = 0 or not (p1 exquo g) case "failed") and
  -- if #v1 = 0 then pp1 = p1, so we know g divides
  -- (#v2 = 0 or not (p2 exquo g) case "failed")
  => g*gcd(c1,c2) -- divides them both, so is the gcd
-- OK, so it's not the gcd: try again
v:=variables g -- there can be at most these variables in answer
v1:=setDifference(vp1,v)
v2:=setDifference(vp2,v)
if (#v1 = 0) then g:=gcdSameVariables(g,flatten(p2,v2),v)
else if (#v2=0) then g:=gcdSameVariables(g,flatten(p1,v1),v)
else g:=gcdSameVariables(g,flatten(p1,v1)-flatten(p2,v2),v)
-- one? g => gcd(c1,c2)::SUPP
  (g = 1) => gcd(c1,c2)::SUPP
  (#v1 = 0 or not (p1 exquo g) case "failed") and
  (#v2 = 0 or not (p2 exquo g) case "failed")
  => g*gcd(c1,c2) -- divides them both, so is the gcd
v:=variables g -- there can be at most these variables in answer
v1:=setDifference(vp1,v)
if #v1 = 0 then
  g:=recursivelyGCDCoefficients(g,v,p1,v1)
-- one? g => return gcd(c1,c2)::SUPP
(g = 1) => return gcd(c1,c2)::SUPP

v := variables g -- there can be at most these variables in answer
v2 := setDifference(vp2, v)

recursivelyGCDCoefficients(g, v, p2, v2) * gcd(c1, c2)

if R has StepThrough then
    randomCount := init()
    randomR() ==
        (v := nextItem(randomCount)) case R =>
            randomCount := v
            SAY("Taking next stepthrough range in General Polynomial GCD Package")$Lisp
            randomCount := init()
            randomCount

else
    randomR() == (random$Integer() rem 100) :: R

    ---- JHD's local functions ----

gcdSameVariables(p1:SUPP, p2:SUPP, lv: List OV) ==
    -- two non-trivial primitive (or, at least, we don't care
    -- about content)
    -- polynomials with precisely the same degree
    #lv = 0 => map((x: R): P+-> x:: P, gcdPolynomial(map(ground, p1),
                   map(ground, p2))

    degree p2 = 1 =>
        case p1 exquo p2 => p2
        1
    gcdLC := gcd(leadingCoefficient p1, leadingCoefficient p2)
    lr := [randomR() for vv in lv]
    count := NonNegativeInteger := 0
    while count < 10 repeat
        while zero? eval(gcdLC, lv, lr) and count<10 repeat
            lr := [randomR() for vv in lv]
            count := count + 1
        count = 10 => error "too many evaluations in GCD code"
        up1 := SUPR := map(y+-> ground eval(y, lv, lr), p1)
        up2 := SUPR := map(z+-> ground eval(z, lv, lr), p2)
        u := gcdPolynomial(up1, up2)
        degree u = 0 => return 1
        -- let's pick a second one, just to check
        lrr := [randomR() for vv in lv]
        while zero? eval(gcdLC, lv, lrr) and count<10 repeat
            lrr := [randomR() for vv in lv]
            count := count + 1
        count = 10 => error "too many evaluations in GCD code"
        vp1 := SUPR := map(x1+-> ground eval(x1, lv, lrr), p1)
        vp2 := SUPR := map(y1+-> ground eval(y1, lv, lrr), p2)
        v := gcdPolynomial(vp1, vp2)
        degree v = 0 => return 1
        if degree v < degree u then
            u := v
            up1 := vp1
    
up2 := up2
lr := lr
up1 := (up1 \text{ exquo } u) :: \text{SUPR}
degree \text{gcd}(u, up1) = 0 =>
    ans := \text{lift}(u, p1, up1, lv, lr)
    ans case \text{SUPP} => return ans
    "next"
up2 := (up2 \text{ exquo } u) :: \text{SUPR}
degree \text{gcd}(u, up2) = 0 =>
    ans := \text{lift}(u, p2, up2, lv, lr)
    ans case \text{SUPP} => return ans
    "next"
-- so neither cofactor is relatively prime
count := 0
while count < 10 repeat
    \text{r} := \text{randomR}()
    uu := up1 + r * up2
    degree \text{gcd}(u, uu) = 0 =>
        ans := \text{lift}(u, p1 + r \cdot P * p2, uu, lv, lr)
        ans case \text{SUPP} => return ans
        "next"
error "too many evaluations in GCD code"

count >= 10 => error "too many evaluations in GCD code"

liftR := \text{liftingCoefficient of \text{p}}
\text{lcp} := \text{leadingCoefficient of \text{p}}
\text{g} := \text{monomial}(\text{lcp}, \text{degree } \text{gR}) + \text{map}(x + -> x \cdot P, \text{reductum } \text{gR})
\text{cf} := \text{monomial}(\text{lcp}, \text{degree } \text{cfR}) + \text{map}(y + -> y \cdot P, \text{reductum } \text{cfR})
\text{p} := \text{lcp} \cdot P -- impose leading coefficient of \text{p} on each factor
while \text{lv} \neq [] repeat
    \text{v} := \text{first lv}
    \text{r} := \text{first lr}
    \text{lv} := \text{rest lv}
    \text{lr} := \text{rest lr}
    \text{thisp} := \text{map}(x1 + -> \text{eval}(x1, lv, lr), p)
    \text{d} := \text{"max"/}[\text{degree}(c, v) \text{ for } c \text{ in coefficients } \text{p}]
    \text{prime} := v \cdot P - r \cdot P
    \text{pn} := \text{prime}
    \text{origFactors} := [\text{g, cf}] :: \text{List SUPP}
for \text{n} in 1 .. \text{d} repeat
    \text{Ecart} := (\text{thisp} - \text{g*cf}) \text{ exquo } \text{pn}
    \text{Ecart} case \text{"failed"} =>
        error \text{"failed lifting in hensel in Complex Polynomial GCD"}
        \text{zero? Ecart} => \text{leave}
        \text{step} := \text{solveLinearPolynomialEquation}(\text{origFactors},
            \text{map}(x2 + -> \text{eval}(x2, v, r), \text{Ecart} :: \text{SUPP}))
    \text{step} case \text{"failed"} => return \text{"failed"}
    \text{g} := g + pn * first step
    \text{cf} := cf + pn * second step
```

1012 CHAPTER 8. CHAPTER G

... pn:=pn*prime
... thisp := g*cf => return "failed"

... recursivelyGCDcoefficients(g:SUPP,v:List OV,p:SUPP,pv:List OV) ==
... mv:=first pv -- take each coefficient w.r.t. mv
... pv:=rest pv -- and recurse on pv as necessary
... d:="max"/[degree(u,mv) for u in coefficients p]
... for i in 0..d repeat
... p1:=map(x+->coefficient(x,mv,i),p)
... oldg:=g
... if pv = [] then g:=gcdSameVariables(g,p1,v)
... else g:= recursivelyGCDcoefficients(p,v,p1,pv)
... -- one? g => return 1
... (g = 1) => return 1
... g``oldg =>
... oldv:=v
... v:=variables g
... pv:=setUnion(pv,setDifference(v,oldv))

... flatten(p1:SUPP,lv:List OV) ==
... #lv = 0 => p1
... lr:=[ randomR() for vv in lv]
... dg:=degree p1
... while dg ^= degree (ans:= map(x+->eval(x,lv,lr),p1)) repeat
... lr:=[ randomR() for vv in lv]
... ans
... -- eval(p1:SUPP,lv:List OV,lr:List R) == map(eval(#1,lv,lr),p1)

... variables(p1:SUPP) ==
... removeDuplicates ("concat"/[variables u for u in coefficients p1])

... gcdTrivial(p1:SUPP,p2:SUPP) ==
... -- p1 is non-zero, but has degree zero
... -- p2 is non-zero
... cp1:=leadingCoefficient p1
... -- one? cp1 => 1
... (cp1 = 1) => 1
... degree p2 = 0 => gcd(cp1,leadingCoefficient p2)::SUPP
... un?:=unit? cp1
... while not zero? p2 and not un? repeat
... cp1:=gcd(leadingCoefficient p2,cp1)
... un?:=unit? cp1
... p2:=reductum p2
... un? => 1
... cp1::SUPP

... Local functions ----
... --JHD -- test if something wrong happened in the gcd
... --JHD failtest(f:P,p1:P,p2:P) : Boolean ==
... --JHD (p1 exquo f) case "failed" or (p2 exquo f) case "failed"
... --JHD
... --JHD -- Choose the integers
```
--JHD    x:OV:=lvar.first
--JHD    lvr:=lvar.rest
--JHD    d1:=degree(p1,x)
--JHD    d2:=degree(p2,x)
--JHD    dd:NNI:=0$NNI
--JHD    nvr:NNI:=#lvr
--JHD    lval:List R :=[]
--JHD    range:I:=8
--JHD    for i in 1.. repeat
--JHD      range:=2*range
--JHD      lval:=[(random()$I rem (2*range) - range)::R for i in 1..nvr]
--JHD    uf1:SUPR:=univariate eval(p1,lvr,lval)
--JHD    degree uf1 ^= d1 => "new point"
--JHD    uf2:SUPR:=univariate eval(p2,lvr,lval)
--JHD    degree uf2 ^= d2 => "new point"
--JHD    u:=gcd(uf1,uf2)
--JHD    du:=degree u
--JHD    --the univariate gcd is 1
--JHD    if du=0 then return [[1$SUPR],lval,0$P]$UTerm
--JHD    ugcd:List SUPR:=[u,(uf1 exquo u)::SUPR,(uf2 exquo u)::SUPR]
--JHD    uterm:=[ugcd,lval,0$P]$UTerm
--JHD    dd:=du
--JHD    du=dd =>
--JHD    --test if one of the polynomials is the gcd
--JHD    dd=d1 =>
--JHD      if !(f:=p2 exquo p1) case "failed") then
--JHD        return [[u],lval,p1]$UTerm
--JHD    dd=d2 =>
--JHD      if !(f:=p1 exquo p2) case "failed") then
--JHD        return [[u],lval,p2]$UTerm
--JHD    dd:=(dd-1)::NNI
--JHD    return uterm
--JHD    du<dd => dd:=du
--JHD    good(f:P,lvr:List OV):Record(upol:SUPR,inval:List R) ==
--JHD    nvr:NNI:=#lvr
--JHD    range:I:=1
--JHD    ltry:List List R:=[[]
--JHD    while true repeat
--JHD      range:=2*range
--JHD      lval:=[(random()$I rem (2*range) -range)::R for i in 1..nvr]
--JHD member?(lval,ltry) => "new point"
--JHD ltry:=cons(lval,ltry)
--JHD uf:=univariate eval(f,lvr,lval)
--JHD if degree gcd(uf,differentiate uf)=0 then return [uf,lval]

--JHD -- impose the right lc
--JHD result:List SUPR :=[]
--JHD lvar:=lvar.rest
--JHD for pol in lipol for leadpol in leadc repeat
--JHD p1:= univariate eval(leadpol,lvar,lval) * pol
--JHD result:= cons((p1 exquo leadingCoefficient pol)::SUPR,result)
--JHD reverse result

--JHD --Compute the gcd between not coprime polynomials
--JHD notCoprime(g:P,p2:P,ldeg:List NNI,lvar:List OV) : P ==
--JHD x:OV:=lvar.first
--JHD lvar1:List OV:=lvar.rest
--JHD lg1:=gcdcofact([g,differentiate(g,x)])
--JHD g1:=lg1.1
--JHD lg:LGcd:=localgcd(g1,p2,lvar)
--JHD (l,lval):=(lg.locgcd,lg.goodint)
--JHD p2:=(p2 exquo 1)::P
--JHD (gd1,gd2):=(l,l)
--JHD ul:=univariate(eval(l,lvar1,lval))
--JHD dl:=degree ul
--JHD if degree gcd(ul,differentiate ul) ^=0 then
--JHD newchoice:=good(l,lvar.rest)
--JHD ul:=newchoice.upol
--JHD lval:=newchoice.inval
--JHD ugl:=univariate(eval(g1,lvar1,lval))
--JHD ulist:=[ugl,univariate eval(p2,lvar1,lval)]
--JHD lcpol:=[leadingCoefficient univariate(g1,x),
--JHD leadingCoefficient univariate(p2,x)]
--JHD while true repeat
--JHD d:SUPR:=gcd(cons(ul,ulist))
--JHD if degree d =0 then return gd1
--JHD lquo:=(ul exquo d)::SUPR
--JHD if degree lquo ^=0 then
--JHD lgcd:=gcdcofact([leadingCoefficient univariate(1,x),lcpol])
--JHD gd2:=lift(1,d,lquo,lgcd,lvar,ldeg,lval)
--JHD l:=gd2
--JHD ul:=univariate(eval(1,lvar1,lval))
--JHD dl:=degree ul
--JHD gd1:=gd1*gd2
--JHD ulist:=[(ul exquo d)::SUPR for uf in ulist]
--JHD

--JHD -- we suppose that the poly have the same mainvar, deg p1<deg p2 and the
--JHD -- polys primitive
--JHD internal(p1:P,p2:P,x:OV) : P ==
--JHD lvar:List OV:=sort(#1>#2,setUnion(variables p1,variables p2))
--JHD d1:=degree(p1,x)
--JHD d2:=degree(p2,x)
--JHD result: P:=localgcd(p1,p2,lvar).locgcd
--JHD -- special cases
--JHD result=1 => 1$P
--JHD (dr:=degree(result,x))=d1 or dr=d2 => result
--JHD while failtest(result,p1,p2) repeat
--JHD SAY$Lisp "retrying gcd"
--JHD result:=localgcd(p1,p2,lvar).locgcd
--JHD result
--JHD
--JHD --local function for the gcd : it returns the evaluation point too
--JHD localgcd(p1:P,p2:P,lvar:List(OV)) : LGcd ==
--JHD x:OV:=lvar.first
--JHD uterm:=chooseVal(p1,p2,lvar)
--JHD listpol:= uterm.lpol
--JHD ud:=listpol.first
--JHD dd:= degree ud
--JHD
--JHD --the univariate gcd is 1
--JHD dd=0 => [1$P,uterm.lint]$LGcd
--JHD
--JHD --one of the polynomials is the gcd
--JHD dd=degree(p1,x) or dd=degree(p2,x) =>
--JHD [uterm.mpol,uterm.lint]$LGcd
--JHD ldeg:List NNI:=map(min,degree(p1,lvar),degree(p2,lvar))
--JHD
--JHD -- if there is a polynomial g s.t. g/gcd and gcd are coprime ...
--JHD -- I can lift
--JHD (h:=lift?(p1,p2,uterm,ldeg,lvar)) case "failed" =>
--JHD [notCoprime(p1,p2,ldeg,lvar),uterm.lint]$LGcd
--JHD [h::P,uterm.lint]$LGcd
--JHD
--JHD
--JHD -- content, internal functions return the poly if it is a monomial
--JHD monomContent(p:P,var:OV):P ==
--JHD ground? p => 1$P
--JHD md:= minimumDegree(p,var)
--JHD ((var::P)**md)*(gcd sort(better,coefficients univariate(p,var)))

monomContentSup(u:SUPP):SUPP ==
  degree(u) = 0$NonNegativeInteger => 1$SUPP
  md:= minimumDegree u
gcd(sort(better,coefficients u)) * monomial(1$P,md)$SUPP

--JHD -- change the polynomials to have positive lc
--JHD abs(p:P): P == unitNormal(p).canonical
CHAPTER 8. CHAPTER G

-- Ordering for gcd purposes
better(p1:P,p2:P):Boolean ==
  ground? p1 => true
  ground? p2 => false
  degree(p1,mainVariable(p1)::OV) < degree(p2,mainVariable(p2)::OV)

-- PRS algorithm
--   u1:= univariate(p1,var)
--   u2:= univariate(p2,var)
--   finished:Boolean:= false
--   until finished repeat
--     dd:NNI:=(degree u1 - degree u2)::NNI
--     lc1:SUPP:=leadingCoefficient u2 * reductum u1
--     lc2:SUPP:=leadingCoefficient u1 * reductum u2
--     u3:SUPP:= primitate((lc1-lc2)*monomial(1$P,dd))$%
--     (d3:=degree(u3)) <= d => finished:= true
--     u1:= u2
--     u2:= u3
--     if d3 > degree(u1) then (u1,u2):= (u2,u1)
--     g:= (u2 exquo u3)
--     g case SUPP => abs multivariate(u3,var)
--     "failed"

-- Gcd between polynomial p1 and p2 with
-- mainVariable p1 < x=mainVariable p2
--gcdTermList(p1:P,p2:P) : P ==
--   termList:=sort(better,
--     cons(p1,coefficients univariate(p2,(mainVariable p2)::OV)))
--   q:P:=termList.first
--   for term in termList.rest until q = 1$P repeat q:= gcd(q,term)
--   q

-- Gcd between polynomials with the same mainVariable
--   if degree(p1,mvar) < degree(p2,mvar) then (p1,p2):= (p2,p1)
--   (p1 exquo p2) case P => abs p2
--   c1:= monomContent(p1,mvar)$%
--   c1 = p1 => gcdMonom(p1,p2,mvar)
--   c2:= monomContent(p2,mvar)$%
--   c2 = p2 => gcdMonom(p2,p1,mvar)
--   p1:= (p1 exquo c1):P
--   p2:= (p2 exquo c2):P
--   if degree(p1,mvar) < degree(p2,mvar) then (p1,p2):= (p2,p1)
--   (p1 exquo p2) case P => abs(p2) * gcd(c1,c2)
--   abs(gcdPrim(p1,p2,mvar)) * gcd(c1,c2)
--   gcdPrim(p1,p2,mvar) * gcd(c1,c2)

-- make the polynomial primitive with respect to var
--primitate(p:P,var:OV):P == (p exquo monomContent(p,var)):P
primitate(u:SUPP):SUPP == (u exquo monomContentSup u)::SUPP

gcd between primitive polynomials with the same mainVariable

  vars:= removeDuplicates append(variables p1,variables p2)
  #vars=1 => multivariate(gcd(univariate p1,univariate p2),mvar)
  vars:=delete(vars,position(mvar,vars))
  --d:= degModGcd(p1,p2,mvar,vars)
  --d case "failed" => internal(p2,p1,mvar)
  --deg:= d:NNI
  --deg = 0$NNI => 1$P
  --deg = degree(p1,mvar) =>
  -- (p2 exquo p1) case P => abs(p1) -- already know that
  -- ~(p1 exquo p2)
  -- internal(p2,p1,mvar)
  --cheapPrs?(p1,p2,deg,mvar) =>
  -- g:= gcdPrs(p1,p2,deg,mvar)
  -- g case P => g::P
  -- internal(p2,p1,mvar)
  internal(p2,p1,mvar)

-- gcd between a monomial and a polynomial

gcdMonom(m:P,p:P,var:OV):P ==
  ((var::P) ** min(minimumDegree(m,var),minimumDegree(p,var))) *
  gcdTermList(leadingCoefficient(univariate(m,var)),p)

-- If there is a pol s.t. pol/gcd and gcd are coprime I can lift

lift?(p1:P,p2:P,uterm:UTerm,ldeg:List NNI,
  lvar:List OV) : Union("failed",P) ==
  x:OV:=lvar.first
  leadpol:Boolean:=false
  (listpol,lval):=(uterm.lpol,uterm.lint)
  d:=listpol.first
  listpol:=listpol.rest
  nolift:Boolean:=true
  for uf in listpol repeat
  --note uf and d not necessarily primitive
  degree gcd(uf,d) =0 => nolift:=false
  nolift => "failed"
  f:P:=([p1,p2]$List(P)).(position(uf,listpol))
  lgcd:=gcd(leadingCoefficient univariate(p1,x),
    leadingCoefficient univariate(p2,x))
  lift(f,d,uf,lgcd,lvar,ldeg,lval)

-- interface with the general "lifting" function

lift(f:P,d:SUPR,uf:SUPR,lgcd:P,lvar:List OV,
  ldeg:List NNI,lval:List R):P ==
  x:OV:=lvar.first
  leadpol:Boolean:=false
  lcf:P
CHAPTER 8. CHAPTER G

--JHD lcf:=leadingCoefficient univariate(f,x)
--JHD df:=degree(f,x)
--JHD leadlist:List(P):=[ ]
--JHD
--JHD if lgcd$=$1$P$ then
--JHD leadpol:=true
--JHD f:=$(lgcd)$f
--JHD ldeg:=[n0+n1 for n0 in ldeg for n1 in degree(lgcd,lvar)]
--JHD lcd:R:=leadingCoefficient d
--JHD if ground? lgcd then d:=((retract lgcd) &* d exquo lcd)::SUPR
--JHD else d:=((retract(eval(lgcd,lvar.rest,lval)) &* d exquo lcd)::SUPR
--JHD uf:=lcd*uf
--JHD leadlist:=[lgcd,lcf]
--JHD lg:=imposeLC([d,uf],lvar,lval,leadlist)
--JHD plist:=lifting(univariate(f,x),lvar,lg,lval,leadlist,ldeg)::List P
--JHD (p0,P1,P2):=(plist.first,plist.2)
--JHD if univariate eval(p0,rest lvar,lval) $=$ lg.first then
--JHD (p0,p1):=(p1,p0)
--JHD leadpol $=>$ p0
--JHD cprim:=contprim([p0])
--JHD cprim.first.prim
--JHD
--JHD -- Gcd for two multivariate polynomials
--JHD gcd(p1,P2,P2):=P ==
--JHD (p1:=abs(p1)) = (p2:=abs(p2)) $=>$ p1
--JHD ground? p1 $=>$
--JHD p1 = 1$P$ $=>$ p1
--JHD p1 = 0$P$ $=>$ p2
--JHD ground? p2 $=>$ gcd((retract p1)@R,(retract p2)@R)::P
--JHD gcdTermList(p1,p2)
--JHD ground? p2 $=>$
--JHD p2 = 1$P$ $=>$ p2
--JHD p2 = 0$P$ $=>$ p1
--JHD gcdTermList(p2,p1)
--JHD mv1:=mainVariable(p1)::OV
--JHD mv2:=mainVariable(p2)::OV
--JHD mv1 = mv2 $=>$ gcdSameMainvar(p1,p2,rmv1)
--JHD mv1 < mv2 $=>$ gcdTermList(p1,p2)
--JHD gcdTermList(p2,p1)
--JHD
--JHD -- Gcd for a list of multivariate polynomials
--JHD gcdcofact(listp:List P):=List P ==
--JHD lf:=sort(better,listp)
--JHD f:=lf.first
--JHD for g in lf.rest repeat
--JHD f:=gcd(f,g)
--JHD if f$=$1$P$ then return f
--JHD f
--JHD -- Gcd and cofactors for a list of polynomials
--JHD gcdcofact(listp:List P):=List P ==
--JHD h:=gcd listp
--JHD cons(h,[(f exquo h) :: P for f in listp])
--JHD
--JHD -- Gcd for primitive polynomials
--JHD gcdprim(p1:P,p2:P):P ==
--JHD (p1:= abs(p1)) = (p2:= abs(p2)) => p1
--JHD ground? p1 =>
--JHD ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
--JHD p1 = 0$P => p2
--JHD 1$P
--JHD ground? p2 =>
--JHD p2 = 0$P => p1
--JHD 1$P
--JHD mv1:= mainVariable(p1)::OV
--JHD mv2:= mainVariable(p2)::OV
--JHD mv1 = mv2 =>
--JHD md:=min(minimumDegree(p1,mv1),minimumDegree(p2,mv1))
--JHD mp:=1$P
--JHD if md>1 then
--JHD mp:=(mv1::P)**md
--JHD p1:=(p1 exquo mp)::P
--JHD p2:=(p2 exquo mp):P
--JHD mp*gcdPrim(p1,p2,mv1)
--JHD 1$P
--JHD
--JHD -- Gcd for a list of primitive multivariate polynomials
--JHD gcdprim(listp:List P) : P ==
--JHD lf:=sort(better,listp)
--JHD f:=lf.first
--JHD for g in lf.rest repeat
--JHD f:=gcdprim(f,g)
--JHD if f=1$P then return f
--JHD f
--JHD
--JHD -- Gcd and cofactors for a list of primitive polynomials
--JHD gcdcofactprim(listp : List P) : List P ==
--JHD h:=gcdprim listp
--JHD cons(h,[(f exquo h) :: P for f in listp])
--JHD
--JHD -- content of a polynomial (with respect to its main var)
--JHD content(f:P):P ==
--JHD ground? f => f
--JHD x:=UV:=(mainVariable f)::OV
--JHD gcd sort(better,coefficients univariate(f,x))
--JHD
--JHD -- contents of a list of polynomials
--JHD content(listf:List P) : List P == [content f for f in listf]
--JHD
--JHD -- contents and primitive parts of a list of polynomials
--JHD contprim(listf:List P) : List ContPrim ==
--JHD prelim :List P := content listf
package GENUPS GenerateUnivariatePowerSeries

-- GenerateUnivariatePowerSeries.input --

)set break resume
)sys rm -f GenerateUnivariatePowerSeries.output
)spool GenerateUnivariatePowerSeries.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show GenerateUnivariatePowerSeries
--R
--R GenerateUnivariatePowerSeries(R: Join(IntegralDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),FE: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(R))) is a package constructor
--R Abbreviation for GenerateUnivariatePowerSeries is GENUPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GENUPS
--R
--R-------------------------------------- Operations -------------------------------------
--R laurent : ((Integer -> FE),Equation(FE),UniversalSegment(Integer)) -> Any
--R laurent : (FE,Symbol,Equation(FE),UniversalSegment(Integer)) -> Any
--R puiseux : ((Fraction(Integer) -> FE),Equation(FE),UniversalSegment(Fraction(Integer)),Fraction(Integer)) -> Any
--R puiseux : (FE,Symbol,Equation(FE),UniversalSegment(Fraction(Integer)),Fraction(Integer)) -> Any
--R series : ((Integer -> FE),Equation(FE)) -> Any
--R series : (FE,Symbol,Equation(FE)) -> Any
--R series : ((Integer -> FE),Equation(FE),UniversalSegment(Integer)) -> Any
--R series : (FE,Symbol,Equation(FE),UniversalSegment(Integer)) -> Any
--R series : ((Fraction(Integer) -> FE),Equation(FE),UniversalSegment(Fraction(Integer)),Fraction(Integer)) -> Any
--R series : (FE,Symbol,Equation(FE),UniversalSegment(Fraction(Integer)),Fraction(Integer)) -> Any
--R taylor : ((Integer -> FE),Equation(FE)) -> Any
package GENUPS GENERATEUNIVARIATEPOWERSERIES

--R taylor : (FE,Symbol,Equation(FE)) -> Any
--R taylor : ((Integer -> FE),Equation(FE),UniversalSegment(NonNegativeInteger)) -> Any
--R taylor : (FE,Symbol,Equation(FE),UniversalSegment(NonNegativeInteger)) -> Any
--R
--E 1

)spool
)lisp (bye)

---

--- GenerateUnivariatePowerSeries.help ---

====================================================================
GenerateUnivariatePowerSeries examples
====================================================================

GenerateUnivariatePowerSeries provides functions that create
power series from explicit formulas for their n-th coefficient.

See Also:
o )show GenerateUnivariatePowerSeries

---

GenerateUnivariatePowerSeries (GENUPS)

Exports:
  laurent puiseux series taylor

--- package GENUPS GenerateUnivariatePowerSeries ---

)abbrev package GENUPS GenerateUnivariatePowerSeries
GenerateUnivariatePowerSeries(R,FE): Exports == Implementation where
R : Join(IntegralDomain,OrderedSet,RetractableTo Integer,_
   LinearlyExplicitRingOver Integer)
FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,_
   FunctionSpace R)
ANY1 ==> AnyFunctions1
EQ ==> Equation
I ==> Integer
NNI ==> NonNegativeInteger
RN ==> Fraction Integer
SEG ==> UniversalSegment
ST ==> Stream
SY ==> Symbol
UTS ==> UnivariateTaylorSeries
ULS ==> UnivariateLaurentSeries
UPXS ==> UnivariatePuiseuxSeries
Exports == with
taylor: (I -> FE,EQ FE) -> Any
   ++ \spad{taylor(n +-> a(n),x = a)} returns
   ++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.
taylor: (FE,SY,EQ FE) -> Any
   ++ \spad{taylor(a(n),n,x = a)} returns \spad{sum(n = 0..,a(n)*(x-a)**n)}.
taylor: (I -> FE,EQ FE,SEG NNI) -> Any
   ++ \spad{taylor(n +-> a(n),x = a,n0..)} returns
   ++ \spad{sum(n=n0..,a(n)*(x-a)**n)};
   ++ \spad{taylor(n +-> a(n),x = a,n0..n1)} returns
   ++ \spad{sum(n=n0..n1,a(n)*(x-a)**n)}.
taylor: (FE,SY,EQ FE,SEG NNI) -> Any
   ++ \spad{taylor(a(n),n,x=a,n0..)} returns
   ++ \spad{sum(n=n0..,a(n)* (x-a)**n)};
   ++ \spad{taylor(a(n),n,x=a,n0..n1)} returns
   ++ \spad{sum(n=n0..n1,a(n)* (x-a)**n)}.
laurent: (I -> FE,EQ FE,SEG I) -> Any
   ++ \spad{laurent(n +-> a(n),x = a,n0..)} returns
   ++ \spad{sum(n=n0..,a(n) * (x - a)**n)};
   ++ \spad{laurent(n +-> a(n),x = a,n0..n1)} returns
   ++ \spad{sum(n=n0..n1,a(n) * (x - a)**n)}.
laurent: (FE,SY,EQ FE,SEG I) -> Any
   ++ \spad{laurent(a(n),n,x=a,n0..)} returns
   ++ \spad{sum(n=n0..,a(n) * (x - a)**n)};
   ++ \spad{laurent(a(n),n,x=a,n0..n1)} returns
++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

puiseux: (RN -> FE, EQ FE, SEG RN, RN) -> Any
++ \spad{puiseux(n +-> a(n),x = a,r0..,r)} returns
++ \spad{sum(n = r0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{puiseux(n +-> a(n),x = a,r0..r1,r)} returns
++ \spad{sum(n = r0 + k*r while n <= r1, a(n) * (x - a)**n)}.

puiseux: (FE, SY, EQ FE, SEG RN, RN) -> Any
++ \spad{puiseux(a(n),n,x = a,r0..,r)} returns
++ \spad{sum(n = n0..n1, a(n) * (x - a)**n)};
++ \spad{puiseux(a(n),n,x = a,r0..r1,r)} returns
++ \spad{sum(n = n0 + k*r while n <= r1, a(n) * (x - a)**n)}.

series: (I -> FE, EQ FE) -> Any
++ \spad{series(n +-> a(n),x = a)} returns
++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.

series: (FE, SY, EQ FE) -> Any
++ \spad{series(a(n),n,x = a)} returns
++ \spad{sum(n = 0..,a(n)*(x-a)**n)}.

series: (I -> FE, EQ FE, SEG I) -> Any
++ \spad{series(n +-> a(n),x = a,n0..)} returns
++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
++ \spad{series(n +-> a(n),x = a,n0..n1)} returns
++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

series: (FE, SY, EQ FE, SEG I) -> Any
++ \spad{series(a(n),n,x=a,n0..)} returns
++ \spad{sum(n = n0..,a(n) * (x - a)**n)};
++ \spad{series(a(n),n,x=a,n0..n1)} returns
++ \spad{sum(n = n0..n1,a(n) * (x - a)**n)}.

series: (RN -> FE, EQ FE, SEG RN, RN) -> Any
++ \spad{series(n +-> a(n),x = a,r0..,r)} returns
++ \spad{sum(n = n0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{series(n +-> a(n),x = a,r0..r1,r)} returns
++ \spad{sum(n = n0 + k*r while n <= r1, a(n) * (x - a)**n)}.

series: (FE, SY, EQ FE, SEG RN, RN) -> Any
++ \spad{series(a(n),n,x = a,r0..,r)} returns
++ \spad{sum(n = n0,r0 + r,r0 + 2*r..., a(n) * (x - a)**n)};
++ \spad{series(a(n),n,x = a,r0..r1,r)} returns
++ \spad{sum(n = n0 + k*r while n <= r1, a(n) * (x - a)**n)}.

Implementation ==> add

genStream: (I -> FE, I) -> ST FE
\text{genStream}(f,n) == \text{delay} \text{concat}(f(n),\text{genStream}(f,n + 1))

genFiniteStream: (I -> FE, I, I) -> ST FE
\text{genFiniteStream}(f,n,m) == \text{delay}
\text{if} n > m \Rightarrow \text{empty}()
\text{concat}(f(n),\text{genFiniteStream}(f,n + 1,m))
taylor(f, eq) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  coerce(series(genStream(f,0))$UTS(FE,x,a))$ANY1(UTS(FE,x,a))

taylor(an:FE,n:SY,eq:EQ FE) ==
  taylor((i:I):FE <+> eval(an,(n::FE) = (i::FE)),eq)

taylor(f:I -> FE,eq:EQ FE,seg:SEG NNI) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  hasHi seg =>
    n0 := lo seg; n1 := hi seg
    if n1 < n0 then (n0,n1) := (n1,n0)
    uts := series(genFiniteStream(f,n0,n1))$UTS(FE,x,a)
    uts := uts * monomial(1,n0)$UTS(FE,x,a)
    coerce(uts)$ANY1(UTS(FE,x,a))
  n0 := lo seg
  uts := series(genStream(f,n0))$UTS(FE,x,a)
  uts := uts * monomial(1,n0)$UTS(FE,x,a)
  coerce(uts)$ANY1(UTS(FE,x,a))

taylor(an,n,eq,seg) ==
  taylor((i:I):FE <+> eval(an,(n::FE) = (i::FE)),eq,seg)

laurent(f, eq, seg) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "taylor: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
  hasHi seg =>
    n0 := lo seg; n1 := hi seg
    if n1 < n0 then (n0,n1) := (n1,n0)
    uts := series(genFiniteStream(f,n0,n1))$UTS(FE,x,a)
    coerce(laurent(n0,uts)$ULS(FE,x,a))$ANY1(ULS(FE,x,a))
  n0 := lo seg
  uts := series(genStream(f,n0))$UTS(FE,x,a)
  coerce(laurent(n0,uts)$ULS(FE,x,a))$ANY1(ULS(FE,x,a))

laurent(an,n,eq,seg) ==
  laurent((i:I):FE <+> eval(an,(n::FE) = (i::FE)),eq,seg)

modifyFcn:(RN -> FE,I,I,I,I) -> FE
modifyFcn(f,n0,nn,q,m) == (zero?((m - n0) rem nn) => f(m/q); 0)

puiseux(f, eq, seg, r) ==
  (xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
    error "puiseux: left hand side must be a variable"
  x := xx :: SY; a := rhs eq
_PACKAGE GENUPS GENERATEUNIVARIATEPOWERSERIES

not positive? r => error "puiseux: last argument must be positive"

hasHi seg =>
    r0 := lo seg; r1 := hi seg
    if r1 < r0 then (r0,r1) := (r1,r0)
    p0 := numer r0; q0 := denom r0
    p1 := numer r1; q1 := denom r1
    p2 := numer r; q2 := denom r
    q := lcm(lcm(q0,q1),q2)
    n0 := p0 * (q quo q0); n1 := p1 * (q quo q1)
    nn := p2 * (q quo q2)
    ulsUnion :=
        laurent((i:I):FE+->modifyFcn(f,n0,nn,q,i),eq,segment(n0,n1))
    uls := retract(ulsUnion)$ANY1(ULS(FE,x,a))
    coerce(puiseux(1/q,uls)$UPXS(FE,x,a))$ANY1(UPXS(FE,x,a))

puiseux(an,n,eq,r0,m) ==
    puiseux((r:RN):FE+->eval(an,(n::FE) = (r::FE)),eq,r0,m)

series(f:I -> FE,eq:EQ FE) == puiseux(r+->f(numer r),eq,segment 0,1)

series(an:FE,n:SY,eq:EQ FE) == puiseux(an,n,eq,segment 0,1)

series(f:I -> FE,eq:EQ FE,seg:SEG I) ==
    ratSeg : SEG RN := map(x+->x::RN,seg)$UniversalSegmentFunctions2(I,RN)
    puiseux((r:RN):FE+->f(numer r),eq,ratSeg.1)

series(an:FE,n:SY,eq:EQ FE,seg:SEG I) ==
    ratSeg : SEG RN := map(i+->i::RN,seg)$UniversalSegmentFunctions2(I,RN)
    puiseux(an,n,eq,ratSeg.1)

series(f:RN -> FE,eq:EQ FE,seg:SEG RN,r:RN) == puiseux(f,eq,seg,r)

series(an:FE,n:SY,eq:EQ FE,seg:SEG RN,r:RN) == puiseux(an,n,eq,seg,r)
package GENE EZ GenExEuclid

--- GenExEuclid.input ---

)set break resume
)sys rm -f GenExEuclid.output
)spool GenExEuclid.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GenExEuclid
   --R
   --R GenExEuclid(R: EuclideanDomain,BP: UnivariatePolynomialCategory(R)) is a package constructor
   --R Abbreviation for GenExEuclid is GENE EZ
   --R This constructor is not exposed in this frame.
   --R Issue )edit bookvol10.4.pamphlet to see algebra source code for GENE EZ
   --R
   --R------------------------------- Operations --------------------------------
   --R reduction : (BP,R) -> BP
   --R compBound : (BP,List(BP)) -> NonNegativeInteger
   --R solveid : (BP,R,Vector(List(BP))) -> Union(List(BP),"failed")
   --R tablePow : (NonNegativeInteger,R,List(BP)) -> Union(Vector(List(BP)),"failed")
   --R testModulus : (R,List(BP)) -> Boolean
   --R
   --E 1

)spool
)lisp (bye)

--- GenExEuclid.help ---

====================================================================
GenExEuclid examples
====================================================================

The equation Af+Bg=h and its generalization to n polynomials
is solved for solutions over the R, euclidean domain.

A table containing the solutions of Af+Bg=x**k is used.

The operations are performed modulus a prime which are in principle
big enough, but the solutions are tested and, in case of failure, a
hensel lifting process is used to get to the right solutions. It will
be used in the factorization of multivariate polynomials over finite
field, with $R = \mathbb{F}[x]$. 

See Also:
- )show GenExEuclid

---

**GenExEuclid (GENEEZ)**

Exports:
- compBound
- reduction
- solveid
- tablePow
- testModulus

---

```lisp
)abbrev package GENEEZ GenExEuclid
++ Author : P.Gianni.
++ Date Created: January 1990
++ Description:

GenExEuclid(R,BP) : C == T
where
  R : EuclideanDomain
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  BP : UnivariatePolynomialCategory R
  L ==> List

C == with
  reduction: (BP,R) -> BP
  ++ reduction(p,prime) reduces the polynomial p modulo prime of R.
  ++ Note that this function is exported only because it’s conditional.
  compBound: (BP,L BP) -> NNI
  ++ compBound(p,lp)
```
++ computes a bound for the coefficients of the solution
++ polynomials.
++ Given a polynomial right hand side p,
++ and a list lp of left hand side polynomials.
++ Exported because it depends on the valuation.

\[\text{tablePow} : (\text{NNI}, \text{R}, \text{L BP}) \to \text{Union}(\text{Vector}(\text{L BP}), \text{"failed"})\]
++ tablePow(maxdeg,prime,lpol) constructs the table with the
++ coefficients of the Extended Euclidean Algorithm for lpol.
++ Here the right side is \spad{x**k}, for k less or equal to maxdeg.
++ The operation returns "failed" when the elements
++ are not coprime modulo prime.

\[\text{sloveid} : (\text{BP}, \text{R}, \text{Vector L BP}) \to \text{Union}(\text{L BP}, \text{"failed"})\]
++ solveid(h,table) computes the coefficients of the
++ extended euclidean algorithm for a list of polynomials
++ whose tablePow is table and with right side h.

\[\text{testModulus} : (\text{R}, \text{L BP}) \to \text{Boolean}\]
++ testModulus(p,lp) returns true if the the prime p
++ is valid for the list of polynomials lp, i.e. preserves
++ the degree and they remain relatively prime.

T == add
if R has multiplicativeValuation then
  \[\text{compBound}(m:BP, \text{listpolys:L BP}) : \text{NNI} ==\]
  \[\text{ldeg}:=[\text{degree } f \text{ for } f \text{ in listpolys}]
  \text{n}:\text{NNI}:=(+/[df \text{ for } df \text{ in } \text{ldeg}])
  \text{normlist}:=[+/[\text{euclideanSize}(u)**2 \text{ for } u \text{ in coefficients } f]
  \text{for } f \text{ in listpolys}]
  \text{nm}:=(+/[\text{euclideanSize}(u)**2 \text{ for } u \text{ in coefficients } m]
  \text{normprod}:=(+/[\text{g}**((n-df)::\text{NNI}) \text{ for } g \text{ in normlist for } df \text{ in } \text{ldeg}]
  2*(\text{approxSqrt}(\text{normprod} * \text{nm})\text{$\text{IntegerRoots}(\text{Integer})$})::\text{NNI}\]
else if R has additiveValuation then
  -- a fairly crude Hadamard-style bound for the solution
  -- based on regarding the problem as a system of linear equations.
  \[\text{compBound}(m:BP, \text{listpolys:L BP}) : \text{NNI} ==\]
  "max"/[\text{euclideanSize } u \text{ for } u \text{ in coefficients } m] +
  "max"/[\text{euclideanSize } u \text{ for } u \text{ in coefficients } p]
  \text{for } p \text{ in listpolys}]
else
  \[\text{compBound}(m:BP, \text{listpolys:L BP}) : \text{NNI} ==\]
  \text{error} "attempt to use compBound without a well-understood valuation"
if R has IntegerNumberSystem then
  \[\text{reduction}(u:BP,p:R):BP ==\]
  \[p = 0 \Rightarrow u\]
  \text{map}(x \mapsto \text{symmetricRemainder}(x,p),u)
else
  \[\text{reduction}(u:BP,p:R):BP ==\]
  \[p = 0 \Rightarrow u\]
  \text{map}(x \mapsto x \text{ rem } p,u)

\[\text{merge}(p:R,q:R):\text{Union}(R, \text{"failed"}) ==\]
\[ p = q \Rightarrow p \\
p = 0 \Rightarrow q \\
q = 0 \Rightarrow p \]
"failed"

```lisp
modInverse(c:R,p:R):R ==
  (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

  invlcv:=modInverse(leadingCoefficient v,p)
  r:=monicDivide(u,reduction(invlcv*v,p))
  reduction(r.remainder,p) ^=0 => "failed"
  reduction(invlcv*r.quotient,p)

FP:=EuclideanModularRing(R,BP,R,exactquo)

--make table global variable!
table:Vector L BP
import GeneralHenselPackage(R,BP)

--local functions
makeProducts : L BP -> L BP
liftSol: (L BP,BP,R,R,Vector L BP,BP,NNI) -> Union(L BP,"failed")
reduceList(lp:L BP,lmod:R): L FP ==[reduce(ff,lmod) for ff in lp]
coerceLFP(lf:L FP):L BP == [fm::BP for fm in lf]

liftSol(oldsol:L BP,err:BP,lmod:R,lmodk:R,
  table:Vector L BP,m:BP,bound:NNI):Union(L BP,"failed") ==
  euclideanSize(lmodk) > bound => "failed"
  d:=degree err
  ftab:Vector L FP :=
    map(x +-> reduceList(x,lmod),table)$VectorFunctions2(List BP,List FP)
  sln:L FP:=[reduce(ff,lmod) for ff in ftab.1]
  for i in 0 .. d |(cc:=coefficient(err,i)) ^=0 repeat
    sln:=[slp+reduce(cc::BP,lmod)*pp
      for pp in ftab.(i+1) for slp in sln]
  nsol:=[f-lmodk*reduction(g::BP,lmod) for f in oldsol for g in sln]
  lmodk1:=lmod*lmodk
  nsol:=[reduction(slp,lmodk1) for slp in nsol]
  lpolys:L BP:=table.(#table)
  (fs:=[f*g for f in lpolys for g in nsol]) = m => nsol
  a:BP:=((fs=m) exquo lmodk1):BP
  liftSol(nsol,a,lmod,lmodk1,table,m,bound)

makeProducts(listPol:L BP):L BP ==
  #listPol < 2 => listPol
  #listPol = 2 => reverse listPol
  f:= first listPol
ll := rest listPol
[*/ll,[f*g for g in makeProducts ll]]

(testModulus(pmod, listPol) ==
redListPol := reduceList(listPol, pmod)
for pol in listPol for rpol in redListPol repeat
  degree(pol) ^= degree(rpol::BP) => return false
while not empty? redListPol repeat
  rpol := first redListPol
  redListPol := rest redListPol
  for rpol2 in redListPol repeat
    gcd(rpol, rpol2) ^= 1 => return false
true

if R has Field then
  (tablePow(mdeg:NNI, pmod:R, listPol:L BP) ==
multiE:=multiEuclidean(listPol, 1$BP)
multiE case "failed" => "failed"
ptable: Vector L BP := new(mdeg+1, [])
ptable.1 := multiE
x:BP := monomial(1, 1)
for i in 2..mdeg repeat ptable.i :=
  [tpol*x rem fpol for tpol in ptable.(i-1) for fpol in listPol]
ptable.(mdeg+1) := makeProducts listPol
ptable

(solveid(m:BP, pmod:R, table: Vector L BP) : Union(L BP, "failed") ==
  -- Actually, there’s no possibility of failure
  d := degree m
  sln: L BP := [0$BP for xx in table.1]
  for i in 0 .. d | coefficient(m, i)^=0 repeat
    sln :=[slp+coefficient(m, i)*pp
        for pp in table.(i+1) for slp in sln]
sln

else

  (tablePow(mdeg:NNI, pmod:R, listPol:L BP) ==
   listP: L FP := [reduce(pol,pmod) for pol in listPol]
   multiE:=multiEuclidean(listP, 1$FP)
   multiE case "failed" => "failed"
   ftable: Vector L FP := new(mdeg+1, [])
   f1: L FP := [ff::FP for ff in multiE]
   ftable.1 := [fpol for fpol in f1]
   x:FP := reduce(monomial(1, 1), pmod)
   for i in 2..mdeg repeat ftable.i :=
     [tpol*x rem fpol for tpol in ftable.(i-1) for fpol in listP]
   ptable := map(coerceLFP, ftable)$VectorFunctions2(List FP, List BP)
   ptable.(mdeg+1) := makeProducts listPol
   ptable

solveid(m:BP,pmod:R,table:Vector L BP) : Union(L BP,"failed") ==
  d:=degree m
  ftab:Vector L FP:=
    map(x+->reduceList(x,pmod),table)$VectorFunctions2(List BP,List FP)
  lpolys:L BP:=table.(#table)
  sln:L FP:=[0$FP for xx in ftab.1]
  for i in 0 .. d | coefficient(m,i)^=0 repeat
    sln:=[slp+reduce(coefficient(m,i)::BP,pmod)*pp
        for pp in ftab.(i+1) for slp in sln]
  soln:=[slp::BP for slp in sln]
  (fs:=/[f*g for f in lpolys for g in soln]) = m=> soln

-- Compute bound
bound:=compBound(m,lpolys)
a:BP:=((fs-m) exquo pmod)::BP
liftSol(soln,a,pmod,pmod,table,m,bound)

— GENEEZ.dotabb —

"GENEEZ" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GENEEZ"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"GENEEZ" -> "PFECAT"

— package GENUFACT GenUFactorize —

仁 GenUFactorize.input —

仁 set break resume
仁 sys rm -f GenUFactorize.output
仁 spool GenUFactorize.output
仁 set message test on
仁 set message auto off
仁 clear all

仁 --S 1 of 1
仁 )show GenUFactorize
仁 --R
仁 --R GenuFact is a package constructor
仁 --R Abbreviation for GenUFactorize is GENUFACT
仁 --R This constructor is not exposed in this frame.
--- Issue )edit bookvol10.4.pamphlet to see algebra source code for GENUFACT
---
--- Operations --------------------------------
---
--- factor : SparseUnivariatePolynomial(R) -> Factored(SparseUnivariatePolynomial(R))
---
--- )spool
--- )lisp (bye)

GenUFactorize (GENUFACT)

Exports:
factor

--- package GENUFACT GenUFactorize ---
**PACKAGE GENUFACT GENUFACCTORIZE**

)abbrev package GENUFACT GenUFactorize
++ Description:
++ This package provides operations for the factorization
++ of univariate polynomials with integer
++ coefficients. The factorization is done by "lifting" the
++ finite "berlekamp's" factorization

GenUFactorize(R) : public == private where
  R : EuclideanDomain
  PR ==> SparseUnivariatePolynomial R -- with factor
  -- should be UnivariatePolynomialCategory
  NNI ==> NonNegativeInteger
  SUP ==> SparseUnivariatePolynomial

public == with
  factor : PR -> Factored PR
  ++ factor(p) returns the factorisation of p

private == add

-- Factorisation currently fails when algebraic extensions have multiple
-- generators.
factorWarning(f:OutputForm):Void ==
  import AnyFunctions1(String)
  import AnyFunctions1(OutputForm)
  outputList(["WARNING (genufact): No known algorithm to factor "::Any, _,
    f::Any, _,
    ", trying square-free."::Any])$OutputPackage

factor(f:PR) : Factored PR ==
  R is Integer => (factor f)$GaloisGroupFactorizer(PR)
  R is Fraction Integer => (factor f)$RationalFactorize(PR)
  R has Field and R has Finite =>
    (factor f)$DistinctDegreeFactorize(R,PR)
  R has (Complex Integer) => (factor f)$ComplexFactorization(Integer,PR)
  R has (Complex Fraction Integer) =>
    (factor f)$ComplexFactorization(Fraction Integer,PR)
  R is AlgebraicNumber => (factor f)$AlgFactor(PR)

-- following is to handle SAE
  R has generator : () -> R =>
    var := symbol(convert(generator():OutputForm)@InputForm)
CHAPTER 8. CHAPTER G

up := UnivariatePolynomial(var, Fraction Integer)
R has MonogenicAlgebra(Fraction Integer, up) =>
factor(f)$SimpleAlgebraicExtensionAlgFactor(up, R, PR)
up := UnivariatePolynomial(var, Fraction Polynomial Integer)
R has MonogenicAlgebra(Fraction Polynomial Integer, up) =>
factor(f)$SAERationalFunctionAlgFactor(upp, R, PR)
factorWarning(f::OutputForm)
squareFree f
factorWarning(f::OutputForm)
squareFree f

——

package INTG0 GenusZeroIntegration

--- GenusZeroIntegration.input ---

)set break resume
)sys rm -f GenusZeroIntegration.output
)spool GenusZeroIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GenusZeroIntegration
--R
--R GenusZeroIntegration(R: Join(GcdDomain,RetractableTo(Integer),OrderedSet,CharacteristicZero,
--R Abbreviation for GenusZeroIntegration is INTG0
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTG0
--R
--R----------------------------- Operations -----------------------------
--R lift : (SparseUnivariatePolynomial(F),Kernel(F)) -> SparseUnivariatePolynomial(Fraction(Spar
Rationalization of several types of genus 0 integrands; This internal package rationalises integrands on curves of the form:

\[ y^2 = a x^2 + b x + c \]
\[ y^2 = \frac{a x + b}{c x + d} \]
\[ f(x, y) = 0 \] where \( f \) has degree 1 in \( x \)

The rationalization is done for integration, limited integration, extended integration and the risch differential equation;

See Also:
- )show GenusZeroIntegration
GenusZeroIntegration (INTG0)

Exports:
- lift
- multivariate
- palgLODE0
- palgRDE0
- palgextint0
- palgint0
- palglimint0
- univariate

--- package INTG0 GenusZeroIntegration ---

)abbrev package INTG0 GenusZeroIntegration
++ Author: Manuel Bronstein
++ Date Created: 11 October 1988
++ Date Last Updated: 24 June 1994
++ Description:
++ Rationalization of several types of genus 0 integrands;
++ This internal package rationalises integrands on curves of the form:
++ \[ y^2 = a x^2 + b x + c \]
++ \[ y^2 = \frac{a x + b}{c x + d} \]
++ \[ f(x, y) = 0 \] where \( f \) has degree 1 in \( x \)
++ The rationalization is done for integration, limited integration,
++ extended integration and the risch differential equation;

GenusZeroIntegration(R, F, L): Exports == Implementation where
R: Join(GcdDomain, RetractableTo Integer, OrderedSet, CharacteristicZero, LinearlyExplicitRingOver Integer)
F: Join(FunctionSpace R, AlgebraicallyClosedField, TranscendentalFunctionCategory)
L: SetCategory
SY => Symbol
Q => Fraction Integer
K => Kernel F
P => SparseMultivariatePolynomial(R, K)
UP => SparseUnivariatePolynomial F
RF => Fraction UP
UPUP => SparseUnivariatePolynomial RF
IR => IntegrationResult F
LOG => Record(coeff:F, logand:F)
U1 ==> Union(F, "failed")
U2 ==> Union(Record(ratpart:F, coeff:F), "failed")
U3 ==> Union(Record(mainpart:F, limitedlogs:List LOG), "failed")
REC ==> Record(coeff:F, var:List K, val:List F)
ODE ==> Record(particular: Union(F, "failed"), basis: List F)
LODO ==> LinearOrdinaryDifferentialOperator1 RF

Exports ==> with
  palgint0 : (F, K, F, UP) -> IR
  ++ palgint0(f, x, y, d, p) returns the integral of \(\text{spad}\{f(x,y)dx\} \)
  ++ where \(y\) is an algebraic function of \(x\) satisfying
  ++ \(\text{spad}\{d(x)^2 y(x)^2 = P(x)\}\).
  palgint0 : (F, K, K, F, RF) -> IR
  ++ palgint0(f, x, y, z, t, c) returns the integral of \(\text{spad}\{f(x,y)dx\} \)
  ++ where \(y\) is an algebraic function of \(x\) satisfying
  ++ \(\text{spad}\{f(x,y)dx = c f(t,y) dy\}; \(c\) and \(t\) are rational functions of \(y\).
  ++ Argument \(z\) is a dummy variable not appearing in \(\text{spad}\{f(x,y)\}\).
  palgextint0: (F, K, K, F, F, UP) -> U2
  ++ palgextint0(f, x, y, g, d, p) returns functions \(\text{spad}\{[h, c]\}\) such
  ++ that \(\text{spad}\{dh/dx = f(x,y) - c g\}, \text{where}\ y\ is an algebraic function
  ++ of \(x\) satisfying \(\text{spad}\{d(x)^2 y(x)^2 = P(x)\}\),
  ++ or "failed" if no such functions exist.
  palgextint0: (F, K, K, K, F, RF) -> U2
  ++ palgextint0(f, x, y, g, z, t, c) returns functions \(\text{spad}\{[h, d]\}\) such
  ++ that \(\text{spad}\{dh/dx = f(x,y) - d g\}, \text{where}\ y\ is an algebraic function
  ++ of \(x\) satisfying \(\text{spad}\{f(x,y)dx = c f(t,y) dy\}; \(c\) and \(t\) are
  ++ rational functions of \(y\).
  ++ Argument \(z\) is a dummy variable not appearing in \(\text{spad}\{f(x,y)\}\).
  ++ The operation returns "failed" if no such functions exist.
  palglimint0: (F, K, List F, F, UP) -> U3
  ++ palglimint0(f, x, y, [u1,...,un], d, p) returns functions \(\text{spad}\{[h, [c1, ui]]\}\) such that the \(ui\)'s are among \(\text{spad}\{[u1,...,un]\}\)
  ++ and \(\text{spad}\{dh + \sum(c1 \log(ui))/dx = f(x,y)\}\) if such functions exist,
  ++ and "failed" otherwise.
  ++ Argument \(y\) is an algebraic function of \(x\) satisfying
  ++ \(\text{spad}\{d(h + \sum ui)/dx = f(x,y)\}\).
  palglimint0: (F, K, K, List F, K, F, RF) -> U3
  ++ palglimint0(f, x, y, [u1,...,un], z, t, c) returns functions
  ++ \(\text{spad}\{[h, [c1, ui]]\}\) such that the \(ui\)'s are among \(\text{spad}\{[u1,...,un]\}\)
  ++ and \(\text{spad}\{dh + \sum(c1 \log(ui))/dx = f(x,y)\}\) if such functions exist,
  ++ and "failed" otherwise.
  ++ Argument \(y\) is an algebraic function of \(x\) satisfying
  ++ \(\text{spad}\{f(x,y)dx = c f(t,y) dy\}; \(c\) and \(t\) are rational functions of \(y\).
  palgRDE0 : (F, F, K, K, (F, F, SY) -> U1, F, UP) -> U1
  ++ palgRDE0(f, g, x, y, foo, d, p) returns a function \(\text{spad}\{z(x,y)\}\)
  ++ such that \(\text{spad}\{dz/dx + n * df/dx z(x,y) = g(x,y)\}\) if such a \(z\) exists,
  ++ and "failed" otherwise.
  ++ Argument \(y\) is an algebraic function of \(x\) satisfying
  ++ \(\text{spad}\{dz/dx + n * df/dx z = g(x,y)\}\).
  ++ Argument \(foo\), called by \(\text{spad}\{foo(a, b, x)\}\), is a function that solves
\[
\text{\texttt{palgRDE0}} : (F, F, K, K, (F, F, SY)\rightarrow U1, K, F, RF)\rightarrow U1
\]
\[
\text{\texttt{palgLODE0}} : (L, F, K, K, K, F, RF)\rightarrow ODE
\]
\[
\text{\texttt{univariate}} : (F, K, K, UP)\rightarrow UP
\]
\[
\text{\texttt{multivariate}} : (UPUP, K, F)\rightarrow F
\]
\[
\text{\texttt{lifting}} : (UP, K)\rightarrow UP
\]
\[
\text{\texttt{if L has LinearOrdinaryDifferentialOperatorCategory F then}}
\]
\[\text{\texttt{Implementation}} \rightarrow \text{add}
\]
\[
\text{\texttt{import RationalIntegration(F, UP)}}
\]
\[
\text{\texttt{import AlgebraicManipulations(R, F)}}
\]
\[
\text{\texttt{import IntegrationResultFunctions2(RF, F)}}
\]
\[
\text{\texttt{import ElementaryFunctionStructurePackage(R, F)}}
\]
\[
\text{\texttt{import SparseUnivariatePolynomialFunctions2(F, RF)}}
\]
\[
\text{\texttt{import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)}}
\]

```plaintext
mkRat : (F, REC, List K)\rightarrow RF
mkRatlX : (F, K, F, K, RF)\rightarrow RF
quads subst : (K, K, F, UP)\rightarrow Record(diff:F, subs:REC, newk:List K)
kcriff : (F, F)\rightarrow List K
checkroot : (F, List K)\rightarrow F
univ : (F, List K, K)\rightarrow RF

dummy := \text{\texttt{kernel(new())$SY}}@K

kcriff(sa, a) == setDifference(kernels sa, kernels a)
checkroot(f, l) == (empty? l \Rightarrow f; rootNormalize(f, first l))
univ(c, l, x) == univariate(checkroot(c, l), x)
univariate(f, x, y, p) == lift(univariate(f, y, p), x)
lift(p, k) == map(x1++\Rightarrow univariate(x1, k), p)
```
package integ0 genuszerointegration

palgint0(f, x, y, den, radi) ==
  -- y is a square root so write f as f1 y + f0 and integrate separately
  ff := univariate(f, x, y, minPoly y)
  f0 := reductum ff
  pr := quadsubst(x, y, den, radi)
  map(f1+->f1(x::F), integrate(retract(f0)@RF)) +
    map(f1+->f1(pr.diff),
        integrate
        mkRat(multivariate(leadingMonomial ff,x,y::F), pr.subs, pr.newk))

-- the algebraic relation is (den * y)**2 = p where p is a * x**2 + b * x + c
-- if p is squarefree, then parametrize in the following form:
-- u = y - x \sqrt{a}
-- x = (u**2 - c) / (b - 2 u \sqrt{a}) = h(u)
-- dx = h'(u) du
-- y = (u + a h(u)) / den = g(u)
-- if a is a perfect square,
-- u = (y - \sqrt{c}) / x
-- x = (b - 2 u \sqrt{c}) / (u**2 - a) = h(u)
-- dx = h'(u) du
-- y = (u h(u) + \sqrt{c}) / den = g(u)
-- otherwise.
-- if p is a square p = a t^2, then we choose only one branch for now:
-- u = x
-- x = u = h(u)
-- dx = du
-- y = t \sqrt{a} / den = g(u)
-- returns [u(x,y), [h'(u), [x,y], [h(u), g(u)]] in both cases,
-- where l is empty if no new square root was needed,
-- l := [k] if k is the new square root kernel that was created.
quadsubst(x, y, den, p) ==
  u := dummy::F
  b := coefficient(p, 1)
  c := coefficient(p, 0)
  sa := rootSimp sqrt(a := coefficient(p, 2))
  zero?(b * b - 4 * a * c) => -- case where p = a (x + b/(2a))^2
    [x::F, [1, [x, y], [u, sa * (u + b / (2*a)) / eval(den,x,u)]]], empty()]
empty? kerdiff(sa, a) =>
  bm2u := b - 2 * u * sa
  q := eval(den, x, xx := (u**2 - c) / bm2u)
  yy := (ua := u + xx * sa) / q
  [y::F - x::F * sa, [2 * ua / bm2u, [x, y], [xx, yy]], empty()]
  u2ma := u**2 - a
  sc := rootSimp sqrt c
  q := eval(den, x, xx := (b - 2 * u * sc) / u2ma)
  yy := (ux := xx * u + sc) / q
  [(y::F - sc) / x::F, [- 2 * ux / u2ma, [x, y], [xx, yy]], kerdiff(sc, c)]
mkRatlx(f,x,y,t,z,dx) ==
rat := univariate(eval(f, [x, y], [t, z::F]), z) * dx
num(rat) / denom(rat)

mkRat(f, rec, l) ==
rat := univariate(checkroot(rec.coeff * eval(f, rec.var, rec.val), l), dummy)
num(rat) / denom(rat)

palgint0(f, x, y, z, xx, dx) ==
map(x1+->multivariate(x1, y), integrate mkRatlx(f, x, y, xx, z, dx))

palgextint0(f, x, y, g, z, xx, dx) ==
map(x1+->multivariate(x1, y),
extendedint(mkRatlx(f, x, y, xx, z, dx), mkRatlx(g, x, y, xx, z, dx)))

palglimint0(f, x, y, lu, z, xx, dx) ==
map(x1+->multivariate(x1, y), limitedint(mkRatlx(f, x, y, xx, z, dx),
[mkRatlx(u, x, y, xx, z, dx) for u in lu]))

palgRDE0(f, g, x, y, rischde, z, xx, dx) ==
(u := rischde(eval(f, [x, y], [xx, z::F]),
multivariate(dx, z) * eval(g, [x, y], [xx, z::F]),
symbolIfCan(z)::SY)) case "failed" => "failed"
eval(u::F, z, y::F)

-- given p = sum_i a_i(X) Y^i, returns sum_i a_i(x) y^i

multivariate(p, x, y) ==
(map((x1:RF):F+->multivariate(x1, x),
p)$SparseUnivariatePolynomialFunctions2(RF, F))

y

if L has LinearOrdinaryDifferentialOperatorCategory F then
import RationalLODE(F, UP)

palgLODE0(eq, g, x, y, den, radi) ==
pr := quadsubst(x, y, den, radi)
d := monomial(univ(inv(pr.subs.coeff), pr.newk, dummy), 1)$LODO
di:LODO := 1 -- will accumulate the powers of d
op:LODO := 0 -- will accumulate the new LODO
for i in 0..degree eq repeat
  op := op + univ(eval(coefficient(eq, i), pr.subs.var, pr.subs.val), pr.newk, dummy) * di
di := d * di
rec := ratDsolve(op, univ(eval(g, pr.subs.var, pr.subs.val), pr.newk, dummy))
bas:List(F) := [b(pr.diff) for b in rec.basis]
rec.particular case "failed" => ["failed", bas]
[(rec.particular)::RF) (pr.diff), bas]

palgLODE0(eq, g, x, y, kz, xx, dx) ==
d := monomial(univariate(inv multivariate(dx, kz), kz), 1)$LODO
di:LODO := 1 -- will accumulate the powers of d
op:LODO := 0 -- will accumulate the new LODO
lk:List(K) := [x, y]
 lv:List(F) := [xx, kz::F]
for i in 0..degree eq repeat
  op := op + univariate(eval(coefficient(eq, i), lk, lv), kz) * di
di := d * di
rec := ratDsolve(op, univariate(eval(g, lk, lv), kz))
bas:List(F) := [multivariate(b, y) for b in rec.basis]
rec.particular case "failed" => ["failed", bas]
[multivariate((rec.particular)::RF, y), bas]

— INTG0.dotabb —

"INTG0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTG0"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"INTG0" -> "FS"
"INTG0" -> "ACF"

package GDRAW GnuDraw

— GnuDraw.input —
CHAPTER 8. CHAPTER G

)set break resume
)sys rm -f GnuDraw.output
)spool GnuDraw.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GnuDraw
--R
--R GnuDraw is a package constructor
--R Abbreviation for GnuDraw is GDRAW
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GDRAW
--R
--R-------------------------------------------- Operations --------------------------------------------
--R gnuDraw : (Expression(Float),SegmentBinding(Float),String,List(DrawOption)) -> Void
--R gnuDraw : (Expression(Float),SegmentBinding(Float),String) -> Void
--R gnuDraw : (Expression(Float),SegmentBinding(Float),SegmentBinding(Float),String,List(DrawOption)) -> Void
--R gnuDraw : (Expression(Float),SegmentBinding(Float),SegmentBinding(Float),String) -> Void
--R
--E 1

)spool
)lisp (bye)

——

--- GnuDraw.help ---

====================================================================
GnuDraw examples
====================================================================

This package provides support for gnuplot. These routines generate output files contain gnuplot scripts that may be processed directly by gnuplot. This is especially convenient in the axiom-wiki environment where gnuplot is called from LaTeX via gnuplottex.

See Also:
o )show GnuDraw

——
GnuDraw (GDRAW)

Exports:
GospersMethod

— package GDRAW GnuDraw —

)abbrev package GDRAW GnuDraw
++ Author: Bill Page and David Cyganski
++ Date: June 25, 2008
++ Description:
++ This package provides support for gnuplot. These routines
++ generate output files contain gnuplot scripts that may be
++ processed directly by gnuplot. This is especially convenient
++ in the axiom-wiki environment where gnuplot is called from
++ LaTeX via gnuplottex.
GnuDraw():Exports == Implementation where

EF ==> Expression Float
SBF ==> SegmentBinding Float
DROP ==> DrawOption
DROP0 ==> DrawOptionFunctions0
STR ==> String
Exports ==> with

gnuDraw:(EF, SBF, STR, List DROP)->Void
++ \spad{gnuDraw} provides 2d plotting with options
++
++X gnuDraw(D(cos(exp(z))/exp(z^2),z),z=-5..5,"out2d.dat",title=="out2d")
++X )sys gnuplot -persist out2d.dat

gnuDraw:(EF, SBF, STR)->Void
++ \spad{gnuDraw} provides 2d plotting, default options
++
++X gnuDraw(D(cos(exp(z))/exp(z^2),z),z=-5..5,"out2d.dat")
++X )sys gnuplot -persist out2d.dat
gnuDraw: (EF, SBF, SBF, STR, List DROP) -> Void
++ \spad{gnuDraw} provides 3d surface plotting with options
++
++ gnuDraw(sin(x)*cos(y),x=-6..4,y=-4..6,"out3d.dat",title="out3d")
++ )sys gnuplot -persist out3d.dat

Implementation ==> add
-- 2-d plotting

import TwoDimensionalViewport, GraphImage, TopLevelDrawFunctions EF
f1:TextFile:=open(filename::FileName,"output")
-- handle optional parameters
writeLine!(f1, concat(\["set title _",title(opts,"")$DROP0,"_"\]))
writeLine!(f1,"plot '-' title '' lw 3 with lines")
-- extract data as List List Point DoubleFloat
p2:=pointLists(getGraph(draw(f, segbind),1));
for p1 in p2 repeat
    for p in p1 repeat
        writeLine!(f1, concat(\[unparse(convert(p.1)@InputForm)," ",
        unparse(convert(p.2)@InputForm)\])
writeLine!(f1); -- blank line need to mark a "branch"
close! f1

-- default title is ""
gnuDraw(f, segbind, filename, [title("")$DROP])

-- 3-d plotting

import SubSpace, ThreeSpace DoubleFloat, TopLevelDrawFunctions EF
f1:TextFile:=open(filename::FileName,"output")
-- process optional parameters
writeLine!(f1, concat(\["set title _",title(opts,"")$DROP0,"_"\]))
writeLine!(f1,"splot '-' title '' with pm3d")
-- extract data as List List Point DoubleFloat
p2:=mesh(subspace(draw(f, segbind1, segbind2)));
for p1 in p2 repeat
    for p in p1 repeat
        writeLine!(f1, concat(\[unparse(convert(p.1)@InputForm)," ",
        unparse(convert(p.2)@InputForm)," ",
        unparse(convert(p.3)@InputForm)\])
writeLine!(f1); -- blank line need to mark a "branch"
close! f1
-- default title is ""
gnuDraw(f, segbind1, segbind2, filename, [title(""$DROP$)])

---

GDRAW.dotabb ---

"GDRAW" [color="#FF4488", href="bookvol10.4.pdf#nameddest=GDRAW"]
"FS" [color="#4488FF", href="bookvol10.2.pdf#nameddest=FS"]
"GDRAW" -> "FS"

---

package GOSPER GosperSummationMethod

--- GosperSummationMethod.input ---

)set break resume
)sys rm -f GosperSummationMethod.output
)spool GosperSummationMethod.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GosperSummationMethod
--R
--R GosperSummationMethod(E: OrderedAbelianMonoidSup,V: OrderedSet,R: IntegralDomain,P: PolynomialCategory(R),Q)
--R Q: Join(RetractableTo(Fraction(Integer)),Field) with
--R coerce : P -> %
--R numer : % -> P
--R denom : % -> P is a package constructor
--R Abbreviation for GosperSummationMethod is GOSPER
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GOSPER
--R
--R------------------------------------------------------------- Operations --------------------------------
--R
--R GospersMethod : (Q,V,(O) -> V)) -> Union(Q,"failed")
--R
--E 1

)spool
)lisp (bye)
GosperSummationMethod examples

Gosper's summation algorithm.

See Also:
- )show GosperSummationMethod
Q: Join(RetractableTo Fraction Integer, Field with
(coerce: P -> %; numer : % -> P; denom : % -> P))

I ==> Integer
RN ==> Fraction I
PQ ==> SparseMultivariatePolynomial(RN, V)
RQ ==> Fraction PQ

Exports ==>
with
  GospersMethod: (Q, V, () -> V) -> Union(Q, "failed")
  + GospersMethod(b, n, new) returns a rational function
  + \( \text{spad}\{\text{rf}(n)\} \) such that \( \text{spad}\{a(n) \ast \text{rf}(n)\} \) is the indefinite
  + sum of \( \text{spad}\{a(n)\}\)
  + with respect to upward difference on \( \text{spad}\{n\} \), i.e.
  + \( \text{spad}\{a(n+1) \ast \text{rf}(n+1) - a(n) \ast \text{rf}(n) = a(n)\}\),
  + where \( \text{spad}\{b(n) = a(n)/a(n-1)\} \) is a rational function.
  + Returns "failed" if no such rational function \( \text{spad}\{\text{rf}(n)\}\)
  + exists.
  + Note that \( \text{spad}\{\text{new}\} \) is a nullary function returning a new
  + \( \text{V} \) every time.
  + The condition on \( \text{spad}\{a(n)\} \) is that \( \text{spad}\{a(n)/a(n-1)\}\)
  + is a rational function of \( \text{spad}\{n\}\).
  +++ \( \text{spad}\{\text{sum}(a(n), n) = \text{rf}(n) \ast a(n)\}\).

Impl ==>
add
import PolynomialCategoryQuotientFunctions(E, V, R, P, Q)
import LinearSystemMatrixPackage(RQ,Vector RQ,Vector RQ,Matrix RQ)

InnerGospersMethod: (RQ, V, () -> V) -> Union(RQ, "failed")
GosperPQR: (PQ, PQ, V, () -> V) -> List PQ
GosperDegBd: (PQ, PQ, PQ, V, () -> V) -> I
GosperF: (I, PQ, PQ, PQ, V, () -> V) -> Union(I, "failed")
deg0: (PQ, V) -> I -- degree with deg 0 = -1.
pCoef: (PQ, PQ) -> PQ -- pCoef(p, a*b**2)
RF2QIfCan: Q -> Union(RQ, "failed")
UP2QIfCan: P -> Union(PQ,"failed")
RFQ2R : RQ -> Q
PQ2R : PQ -> Q
rat? : R -> Boolean

deg0(p, v) == (zero? p => -1; degree(p, v))
rat? x == retractIfCan(x::P::Q)@Union(RN, "failed") case RN
RFQ2R f == PQ2R(numer f) / PQ2R(denom f)
PQ2R p ==
  map(x+->x::P::Q, y+->y::Q, p)$PolynomialCategoryLifting(
    IndexedExponents V, V, RN, PQ, Q)

GospersMethod(aquo, n, newV) ==
((q := RF2QIfCan aquo) case "failed") or
  ((u := InnerGospersMethod(q::RQ, n, newV)) case "failed") =>
    "failed"
RFQ2R(u::RQ)

RFQ2QIfCan f ==
  (n := UP2QIfCan numer f) case "failed" => "failed"
  (d := UP2QIfCan denom f) case "failed" => "failed"
  n::PQ / d::PQ

UP2QIfCan p ==
  every?(rat?, coefficients p) =>
    map(x +-> x::PQ,
      y +-> (retractIfCan(y::P::Q)@Union(RN, "failed"))::RN::PQ,p).
  $PolynomialCategoryLifting(E, V, R, P, PQ)
  "failed"

InnerGospersMethod(aquo, n, newV) ==
  -- 1. Define coprime polys an,anm1 such that
  --    an/anm1=a(n)/a(n-1)
  --    an := numer aquo
  --    anm1 := denom aquo

  -- 2. Define p,q,r such that
  --    a(n)/a(n-1) = (p(n)/p(n-1)) * (q(n)/r(n))
  --    and
  --    gcd(q(n), r(n+j)) = 1, for all j: NNI.
  pqr := GosperPQR(an, anm1, n, newV)
  pn := first pqr; qn := second pqr; rn := third pqr

  -- 3. If the sum is a rational fn, there is a poly f with
  --    sum(a(n), n) = q(n+1)/p(n) * a(n) * f(n).

  -- 4. Bound the degree of f(n).
  (k := GosperDegBd(pn, qn, rn, n, newV)) < 0 => "failed"

  -- 5. Find a polynomial f of degree at most k, satisfying
  --    p(n) = q(n+1)*f(n) - r(n)*f(n-1)
  (ufn := GosperF(k, pn, qn, rn, n, newV)) case "failed" =>
    "failed"
  fn := ufn::RQ

  -- 6. The sum is q(n-1)/p(n)*f(n) * a(n). We leave out a(n).
  --    qnm1 := eval(qn,n,n::PQ - 1)
  --    qnm1/pn * fn
  qn1 := eval(qn,n,n::PQ + 1)
  qn1/pn * fn

GosperF(k, pn, qn, rn, n, newV) ==
  mv := newV(); mp := mv::PQ; np := n::PQ
fn: \( P := \frac{\sum_{i=0}^{k} m^{i+1} n^{i}}{\sum_{i=0}^{k} n^{i}} \) for \( i \) in 0..k
fnminus1: \( P := \text{eval}(fn, n, np-1) \)
qnplus1 = \( \text{eval}(qn, n, np+1) \)
zro := \( qnplus1 \cdot fn - rn \cdot fnminus1 - pn \)
zron := \( \text{univariate}(zro, n) \)
dz := degree zron
mat: \( \text{Matrix} RQ := \text{zero}(dz+1, (k+1)::\text{NonNegativeInteger}) \)
vec: \( \text{Vector} RQ := \text{new}(dz+1, 0) \)
while zron ^= 0 repeat
  cz := leadingCoefficient zron
  dz := degree zron
  zron := \( \text{reductum}(zron) \)
  mz := \( \text{univariate}(cz, mv) \)
  while mz ^= 0 repeat
    cmz := leadingCoefficient(mz)::RQ
    dmz := degree mz
    mz := \( \text{reductum}(mz) \)
    dmz = 0 \( \Rightarrow \) vec(dz + minIndex vec) := -cmz
  qsetelt_!(mat, dz + minRowIndex mat, dmz + minColIndex(mat) - 1, cmz)
(soln := \( \text{particularSolution}(mat, vec) \)) case "failed" => "failed"
vec := soln::Vector RQ
(\( +\sum_{i=0}^{k} np^{i} \cdot vec(i + minIndex vec) \) for \( i \) in 0..k)@RQ

GosperPQR(an, anm1, n, newV) ==
  np := n::PQ -- polynomial version of \( n \)
  -- Initial guess.
  pn: \( P := 1 \)
  qn: \( P := an \)
  rn: \( P := anm1 \)
  -- Find all \( j: \text{NNI} \) giving common factors to \( q(n) \) and \( r(n+j) \).
  j := \( \text{newV}(\) \)
  rnj := \( \text{eval}(rn, n, np + j::PQ) \)
  res := \( \text{resultant}(qn, rnj, n) \)
  fres := \( \text{factor}(\text{res})$\text{MRationalFactorize}(\text{IndexedExponents V, V, I, PQ}) \)
  js := \( [rt::I \text{ for } fe \text{ in } \text{fres} \]
  | (rt := linearAndNNIntRoot(fe.factor,j)) \text{ case I}] \)
  -- For each such \( j \), change variables to remove the gcd.
  for rt in js repeat
    rtp:= rt::PQ -- polynomial version of \( rt \)
    gn := \( \text{gcd}(qn, \text{eval}(rn,n,np+rtp)) \)
    qn := (qn exquo gn)::PQ
    rn := (rn exquo \text{eval}(gn, n, np-rtp))::PQ
    pn := pn * \( /[\text{eval}(gn, n, np-i::PQ) \text{ for } i \text{ in } 0..rt-1] \)
   [pn, qn, rn]
  -- Find a degree bound for the polynomial \( f(n) \) which satisfies
  -- \( p(n) = q(n+1) \cdot f(n) - r(n) \cdot f(n-1) \).
  GosperDegBd(pn, qn, rn, n, newV) ==
np := n::PQ
qnplus1 := eval(qn, n, np+1)
lplus := deg0(qnplus1 + rn, n)
lminus := deg0(qnplus1 - rn, n)
degp := deg0(pn, n)
k := degp - max(lplus-1, lminus)
lplus <= lminus => k
-- Find L(k), such that
-- p(n) = L(k)*c[k]*n**k + ... + ...
-- To do this, write f(n) and f(n-1) symbolically.
-- f(n) = c[k]*n**k + c[k-1]*n**(k-1) + O(n**(k-2))
-- f(n-1) = c[k]*n**k + (c[k-1] - k*c[k])*n**(k-1) + O(n**(k-2))
kk := newV()::PQ
ck := newV()::PQ
ckm1 := newV()::PQ
nkm1 := newV()::PQ
nk := np*nkm1
headfn := ck*nk + ckm1*nkm1
headfnm1 := ck*nk + (ckm1 - kk*ck)*nkm1
-- Then p(n) = q(n+1)*f(n) - r(n)*f(n-1) gives L(k).
pk := qnplus1 * headfn - rn * headfnm1
lcpk := pCoef(pk, ck*np*nkm1)
-- The degree bd is now given by k, and the root of L.
k0 := linearAndNNIntRoot(lcpk, mainVariable(kk)::V)
k0 case "failed" => k
max(k0::I, k)

pCoef(p, nom) ==
not monomial? nom =>
  error "pCoef requires a monomial 2nd arg"
vlist := variables nom
for v in vlist while p ^= 0 repeat
  unom := univariate(nom, v)
  pow := degree unom
  nom := leadingCoefficient unom
  up := univariate(p, v)
  p := coefficient(up, pow)
  p

linearAndNNIntRoot(mp, v) ==
p := univariate(mp, v)
degree p ^= 1 => "failed"
(pl := retractIfCan(coefficient(p, 1)) @ Union(RN, "failed"))
case "failed" or
  (p0 := retractIfCan(coefficient(p, 0)) @ Union(RN, "failed"))
case "failed" => "failed"
rt := -(p0::RN)/(p1::RN)
rt < 0 or denom rt ^= 1 => "failed"
numer rt
package GRDEF GraphicsDefaults

---

--- GraphicsDefaults.input ---

)set break resume
)sys rm -f GraphicsDefaults.output
)spool GraphicsDefaults.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GraphicsDefaults
--R
--R GraphicsDefaults is a package constructor
--R Abbreviation for GraphicsDefaults is GRDEF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GRDEF
--R
--R-------------------------------------- Operations -----------------------------------
--R adaptive : () -> Boolean           adaptive : Boolean -> Boolean
--R clipPointsDefault : () -> Boolean  drawToScale : () -> Boolean
--R drawToScale : Boolean -> Boolean   maxPoints : () -> Integer
--R maxPoints : Integer -> Integer    minPoints : () -> Integer
--R minPoints : Integer -> Integer    screenResolution : () -> Integer
--R clipPointsDefault : Boolean -> Boolean
--R screenResolution : Integer -> Integer

--E 1

)spool
)lisp (bye)

---
GraphicsDefaults examples

TwoDimensionalPlotSettings sets global flags and constants for 2-dimensional plotting.

See Also:
- )show GraphicsDefaults

Exports:
  adaptive clipPointsDefault drawToScale maxPoints minPoints screenResolution

---

package GRDEF GraphicsDefaults ---

)abbrev package GRDEF GraphicsDefaults
++ Author: Clifton J. Williamson
++ Date Created: 8 January 1990
++ Date Last Updated: 8 January 1990
++ Description:
  ++ TwoDimensionalPlotSettings sets global flags and constants
  ++ for 2-dimensional plotting.

GraphicsDefaults(): Exports == Implementation where
  B ==> Boolean
  I ==> Integer
SF ==> DoubleFloat
maxWidth ==> 1000
maxHeight ==> 1000

Exports ==> with
  clipPointsDefault: () -> B
    ++ clipPointsDefault() determines whether or not automatic clipping is
    ++ to be done.
  drawToScale: () -> B
    ++ drawToScale() determines whether or not plots are to be drawn to scale.

clipPointsDefault: B -> B
  ++ clipPointsDefault(true) turns on automatic clipping;
  ++ \spad{clipPointsDefault(false)} turns off automatic clipping.
  ++ The default setting is true.
  drawToScale: B -> B
  ++ drawToScale(true) causes plots to be drawn to scale.
  ++ \spad{drawToScale(false)} causes plots to be drawn so that they
  ++ fill up the viewport window.
  ++ The default setting is false.

--% settings from the two-dimensional plot package

  adaptive: () -> B
    ++ adaptive() determines whether plotting will be done adaptively.
  maxPoints: () -> I
    ++ maxPoints() returns the maximum number of points in a plot.
  minPoints: () -> I
    ++ minPoints() returns the minimum number of points in a plot.
  screenResolution: () -> I
    ++ screenResolution() returns the screen resolution n.

  adaptive: B -> B
    ++ adaptive(true) turns adaptive plotting on;
    ++ \spad{adaptive(false)} turns adaptive plotting off.
  maxPoints: I -> I
    ++ maxPoints() sets the maximum number of points in a plot.
  minPoints: I -> I
    ++ minPoints() sets the minimum number of points in a plot.
  screenResolution: I -> I
    ++ screenResolution(n) sets the screen resolution to n.

Implementation ==> add

--% global flags and constants

  CLIPPOINTSDEFAULT : B := true
  TOSCALE : B := false

--% functions
clipPointsDefault() == CLIPPOINTSDEFAULT
drawToScale() == TOSCALE

clipPointsDefault b == CLIPPOINTSDEFAULT := b
drawToScale b == TOSCALE := b

--% settings from the two-dimensional plot package

adaptive() == adaptive?()$Plot
minPoints() == minPoints()$Plot
maxPoints() == maxPoints()$Plot
screenResolution() == screenResolution()$Plot

adaptive b == setAdaptive(b)$Plot
minPoints n == setMinPoints(n)$Plot
maxPoints n == setMaxPoints(n)$Plot
screenResolution n == setScreenResolution(n)$Plot


package GRAPHVIZ Graphviz

--- Graphviz.input ---

)set break resume
)sys rm -f Graphviz.output
)spool Graphviz.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
header:=standardDotHeader()
--R
--R
--R (1)
--R ["digraph graphname {", "graph [rankdir="LR" ranksep="3.0"]", "node [style=filled];", "edge [penwidth="0.5" color="blue"];"
--R ]
--R Type: List(String)
--E 1

--S 2 of 5
graph:=sampleDotGraph()
--R
--R
--R (2)
--R ["I1 [fillcolor="white"];", "I2 [fillcolor="white"];", "N1 [fillcolor="cadetblue"];", "N2 [fillcolor="coral"];", "N3 [fillcolor="green"];", "N4 [fillcolor="gold"];", "N5 [fillcolor="cyan"];", "N6 [fillcolor="red"];", "N7 [fillcolor="yellow"];", "N8 [fillcolor="orange"];", "O1 [fillcolor="white"];", "O2 [fillcolor="white"];", "I1 -> N1;", "I1 -> N2;", "I2 -> N1;", "I2 -> N2;", "I2 -> N3;", "N1 -> N4;", "N1 -> N5;", "N1 -> N6;", "N2 -> N4;", "N2 -> N5;", "N2 -> N6;", "N3 -> N4;", "N3 -> N5;", "N3 -> N6 [color="red" penwidth="3"];", "N4 -> N7;", "N4 -> N8;", "N5 -> N7;", "N5 -> N8;", "N6 -> N7;", "N6 -> N8;", "N7 -> O1;", "N8 -> O2;"]
--R
--R Type: List(String)
--E 2

--S 3 of 5
writeDotGraph(header,graph,"NeuralNet")
--R
--R
--E 3

--S 4 of 5
dot2eps "NeuralNet"
--R
--R
--E 4

--S 5 of 5
}show Graphviz
--R
--R Graphviz is a package constructor
--R Abbreviation for Graphviz is GRAPHVIZ
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GRAPHVIZ
--R
--R----------------------------- Operations -----------------------------
--R dot2eps : String -> Void
dotview : (String,String) -> Void
--R sampleDotGraph : () -> List(String)
--R standardDotHeader : () -> List(String)
Graphviz dot files have a header which might look like

```plaintext
digraph graphname {
    graph [rankdir="LR" ranksep="3.0"]
    node [style=filled];
    edge [penwidth="0.5" color="blue"];  
}
```

This default header can be created using `standardDotHeader()`

`header:=standardDotHeader()`

A graph is the body of the dot file containing information about what the nodes are and how they are connected. It might contain node information such as

```plaintext
I1 [fillcolor="white"];  
N1 [fillcolor="cadetblue"];  
```

and line information such as

```plaintext
I1 -> N1 [color="red" penwidth="3"];  
```

A sample graph can be created using `sampleDotGraph()`

`graph:=sampleDotGraph()`

Once the header and graph information is created we can write that information into a file using the function `writeDotGraph`. Two things to note are

* a trailing '}' line is automatically added
* the supplied name adds the ".dot" extension

writeDotGraph(header, graph, "NeuralNet")

will create the file "NeuralNet.dot"

The "NeuralNet.dot" file can be graphed and stored in an encapsulated postscript (.eps) format with

\texttt{dot2eps "NeuralNet"}

will create the file "NeuralNet.eps"

You can call a viewer on your platform. Linux has 'evince'. MAC has 'gv'. MAC has 'open'. Most places have 'firefox'

\texttt{dotview("evince","NeuralNet")}

See Also: o )show Graphviz

_____

Graphviz (GRAPHVIZ)

\begin{center}
\begin{tikzpicture}
  \node[above] at (0,0) {GRAPHVIZ};
  \node[below] at (0,-1) {STRING};
  \draw[->] (0,0) -- (0,-1);
\end{tikzpicture}
\end{center}

Exports: dot2eps dotview sampleDotGraph standardDotHeader writeDotGraph

--- package GRAPHVIZ Graphviz ---

)abbrev package GRAPHVIZ Graphviz
++ Author: Mark Botch
Graphviz(): Exports == Implementation where

HEADER ==> List String
BODY   ==> List String
GRAPH  ==> List String
DOTFILE ==> List String
FILENAME ==> String

Exports ==> with

standardDotHeader: () -> HEADER
++ create the standard dot
++
++X header:=standardDotHeader()

sampleDotGraph: () -> GRAPH
++ creates a sample graph file
++
++X graph:=sampleDotGraph()

writeDotGraph: (HEADER, GRAPH, FILENAME) -> Void
++ creates a graphviz dot file and writes it out
++
++X header:=standardDotHeader()
++X graph:=sampleDotGraph()
++X writeDotGraph(header, graph,"NeuralNet")

dot2eps: FILENAME -> Void
++ dot2eps runs dot -T eps filename.dot >filename.eps
++
++X dot2eps "NeuralNet"

dotview:(String,String) -> Void
++ dotview runs "viewer filename".
++ A file extension of ".eps" is added.
++
++X dotview("evince","NeuralNet") -- on Linux
++X dotview("gv","NeuralNet") -- on MAC
++X dotview("firefox","NeuralNet") -- most places

Implementation == add

standardDotHeader() ==
["digraph graphname {",
 "graph [rankdir="LR", ranksep="3.0"]",_
 "node [style=filled]",_
 "edge [penwidth="0.5", color="blue"]"];"]
sampleDotGraph() ==
["I1 [fillcolor="white"]; ",
 "I2 [fillcolor="white"]; ",
 "N1 [fillcolor="cadetblue"]; ",
 "N2 [fillcolor="coral"]; ",
 "N3 [fillcolor="green"]; ",
 "N4 [fillcolor="gold"]; ",
 "N5 [fillcolor="cyan"]; ",
 "N6 [fillcolor="red"]; ",
 "N7 [fillcolor="yellow"]; ",
 "N8 [fillcolor="orange"]; ",
 "O1 [fillcolor="white"]; ",
 "O2 [fillcolor="white"]; ",
 "I1 -> N1; ",
 "I1 -> N2; ",
 "I1 -> N3; ",
 "I2 -> N1; ",
 "I2 -> N2; ",
 "I2 -> N3; ",
 "N1 -> N4; ",
 "N1 -> N5; ",
 "N1 -> N6; ",
 "N2 -> N4; ",
 "N2 -> N5; ",
 "N2 -> N6; ",
 "N3 -> N4; ",
 "N3 -> N5; ",
 "N3 -> N6 [color="red" penwidth="3"]; ",
 "N4 -> N7; ",
 "N4 -> N8; ",
 "N5 -> N7; ",
 "N5 -> N8; ",
 "N6 -> N7; ",
 "N6 -> N8; ",
 "N7 -> O1; ",
 "N8 -> O2; "]
]

writeDotGraph(header:HEADER, graph:GRAPH, name:FILENAME):Void ==
file:TextFile:=open(concat(name,".dot")::FileName,"output")
for line in header repeat writeLine!(file,line)
for line in graph repeat writeLine!(file,line)
write!(file,"}
close!(file)
void()

dot2eps(file) ==
instr:String:=concat(file,".dot >")
outstr:String:=concat(file,".eps")
command:=concat("dot -T eps ",concat(instr,outstr))
SYSTEM(command)$Lisp
void()

dotview(viewr,file) ==
outstr:String:=concat(file,".eps")
SYSTEM(concat(viewr,concat(" ",outstr)))$Lisp
void()

package GRAY GrayCode

--- GrayCode.input ---

)set break resume
)sys rm -f GrayCode.output
)spool GrayCode.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GrayCode
--R
--R GrayCode is a package constructor
--R Abbreviation for GrayCode is GRAY
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GRAY
--R
--R---------------------------------------- Operations ----------------------------------------
--R firstSubsetGray : PositiveInteger -> Vector(Vector(Integer))
GrayCode provides a function for efficiently running through all subsets of a finite set, only changing one element by another one.

See Also:
- )show GrayCode

Exports:
- firstSubsetGray
- nextSubsetGray

++ Authors: Johannes Grabmeier, Oswald Gschnitzer
++ Date Created: 7 August 1989
++ Date Last Updated: 23 August 1990
++ References:
++ Henryk Minc: Evaluation of Permanents,
++ Nijenhuis and Wilf: Combinatorical Algorithms, Academic
++ S.G.Williamson, Combinatorics for Computer Science,
++ Description:
++ GrayCode provides a function for efficiently running
++ through all subsets of a finite set, only changing one element
++ by another one.

GrayCode: public == private where

PI ==> PositiveInteger
I ==> Integer
V ==> Vector

public ==> with

nextSubsetGray: (V V I, PI) -> V V I
++ nextSubsetGray(ww, n) returns a vector vv whose components
++ have the following meanings:
++ vv.1: a vector of length n whose entries are 0 or 1. This
++ can be interpreted as a code for a subset of the set 1, ..., n;
++ vv.1 differs from ww.1 by exactly one entry;
++ vv.2.1 is the number of the entry of vv.1 which
++ will be changed next time;
++ vv.2.1 = n+1 means that vv.1 is the last subset;
++ trying to compute nextSubsetGray(vv) if vv.2.1 = n+1
++ will produce an error!
++ The other components of vv.2 are needed to compute
++ nextSubsetGray efficiently.
++ Note that this is an implementation of [Williamson, Topic II, 3.54,
++ p. 112] for the special case r1 = r2 = ... = rn = 2;
++ Note that nextSubsetGray produces a side-effect, i.e.
++ nextSubsetGray(vv) and vv := nextSubsetGray(vv)
++ will have the same effect.

firstSubsetGray: PI -> V V I
++ firstSubsetGray(n) creates the first vector ww to start a
++ loop using nextSubsetGray(ww, n)

private ==> add

firstSubsetGray(n : PI) ==
vv : V V I := new(2, [])
vv.1 := new(n,0) @ V I
vv.2 := new(n+1,1) @ V I
for i in 1..(n+1) repeat
  vv.2.i := i
vv

nextSubsetGray(vv : V V I,n : PI) ==
  subs : V I := vv.1 -- subset
  lab : V I := vv.2 -- labels
  c : I := lab(1) -- element which is to be changed next
  lab(1):= 1
  if subs.c = 0 then subs.c := 1
  else subs.c := 0
  lab.c := lab(c+1)
  lab(c+1) := c+1
vv

/package GBF GroebnerFactorizationPackage/

---

GRAY.dotabb ---

"GRAY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GRAY"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"GRAY" -> "IVECTOR"

---

package GBF GroebnerFactorizationPackage

--- GroebnerFactorizationPackage.input ---

)set break resume
)sys rm -f GroebnerFactorizationPackage.output
)spool GroebnerFactorizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 3
mfzn : SqMatrix(6,DMP([x,y,z],Fraction INT)) := [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y], [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
--R
--R
--R +0 1 1 1 1 1+
--R | |
--R | 8 8 |
--R | 1 0 1 - x - |
--R | 3 3 |
--R | |
--R | 8 |
--R | 1 1 0 1 - y |
--R | 3 |
--R |
--R (1) | 8 8 |
--R | 1 - 1 0 1 - |
--R | 3 3 |
--R | |
--R | 8 |
--R | 1 x - 1 0 1 |
--R | 3 |
--R |
--R | 8 8 |
--R | 1 - y - 1 0 |
--R + 3 3 +
--R Type: SquareMatrix(6,DistributedMultivariatePolynomial([x,y,z],Fraction(Integer)))
--E 1

--S 2 of 3
eq := determinant mfzn
--R
--R
--R (2) 22 2 25 2 22 2 388 250 25 2 250 14575
--R - x y + -- x y - -- x - -- x y - -- x y - -- x y - -- x y + -----
--R 3 9 3 9 3 9 9 9 9 81
--R Type: DistributedMultivariatePolynomial([x,y,z],Fraction(Integer))
--E 2

--S 3 of 3
groebnerFactorize [eq, eval(eq, [x,y,z],[y,z,x]), eval(eq,[x,y,z],[z,x,y])]
--R
--R
--R (3)
--R [22 22 22 121
--R [x y + x z - -- x + y z - -- y - -- z + --,
--R 3 3 3 3
--R 2 22 25 2 22 25 2 22 2 388 250
--R x z - -- x z - -- x + y z - -- y z + -- y - -- z + -- z + --,
--R 3 3 9 3 3 9 3 9 27
--R 22 25 2 22 2 388 250 25 2 250 14575
--R y z - -- y z - -- y - -- y z + -- y z + -- y + -- z + -- z - -----]
--R 3 9 3 9 27 9 27 81
--R ,
--R 21994 2 21994 4427 463
--R  [x + y - ----,y - ----- y + ----,z - ---],
--R  5625  5625  675  87
--R  2  1  11  5  265  2  38  265
--R  [x - --,y - --,z - --], [x - --,y - --,z - --], [x + --,y + --,z + --],
--R  2  2  6  18  3  9
--R  25  11  11  11  11  5  5  5
--R  [x - --,y - --,z - --], [x - --,y - --,z - --], [x + --,y + --,z + --],
--R  9  3  3  3  3  3  3  3
--R  19  5  5
--R  [x - --,y + --,z + --]
--R  3  3  3
--R Type: List(List(DistributedMultivariatePolynomial([x,y,z],Fraction(Integer))))

--E 3

--S 4 of 4
SHOW GroebnerFactorizationPackage

--R
--R GroebnerFactorizationPackage(Dom: Join(EuclideanDomain,CharacteristicZero),Expon: OrderedAbelianMonoidSup,VarSet: OrderedSet,Dpol: PolynomialCategory(Dom,Expon,VarSet)) is a package constructor
--R Abbreviation for GroebnerFactorizationPackage is GBF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GBF
--R
--R------------------------------------------------------ Operations -----------------------------
--R factorGroebnerBasis : List(Dpol) -> List(List(Dpol))
--R factorGroebnerBasis : (List(Dpol),Boolean) -> List(List(Dpol))
--R groebnerFactorize : (List(Dpol),List(Dpol)) -> List(List(Dpol))
--R groebnerFactorize : (List(Dpol),List(Dpol),Boolean) -> List(List(Dpol))
--R groebnerFactorize : List(Dpol) -> List(List(Dpol))
--R groebnerFactorize : (List(Dpol),Boolean) -> List(List(Dpol))

--E 4

)spool
)lisp (bye)

—— GroebnerFactorizationPackage.help ——

GroebnerFactorizationPackage provides the function “groebnerFactor” which uses the factorization routines of Axiom to factor each polynomial under consideration while doing the groebner basis algorithm. Then it writes the ideal as an intersection of ideals determined by the irreducible factors. Note that the whole ring may occur as well as other redundancies. We also use the fact, that from the second factor on we can assume that
the preceding factors are not equal to 0 and we divide all polynomials under considerations by the elements of this list of "nonZeroRestrictions".

The result is a list of groebner bases, whose union of solutions of the corresponding systems of equations is the solution of the system of equation corresponding to the input list. The term ordering is determined by the polynomial type used.

Suggested types include
* DistributedMultivariatePolynomial
* HomogeneousDistributedMultivariatePolynomial
* GeneralDistributedMultivariatePolynomial

Solving systems of polynomial equations with the Groebner basis algorithm can often be very time consuming because, in general, the algorithm has exponential run-time. These systems, which often come from concrete applications, frequently have symmetries which are not taken advantage of by the algorithm. However, it often happens in this case that the polynomials which occur during the Groebner calculations are reducible. Since Axiom has an excellent polynomial factorization algorithm, it is very natural to combine the Groebner and factorization algorithms.

GroebnerFactorizationPackage exports the groebnerFactorize operation which implements a modified Groebner basis algorithm. In this algorithm, each polynomial that is to be put into the partial list of the basis is first factored. The remaining calculation is split into as many parts as there are irreducible factors. Call these factors p1,...,pN. In the branches corresponding to p2,...,pN, the factor p1 can be divided out, and so on. This package also contains operations that allow you to specify the polynomials that are not zero on the common roots of the final Groebner basis.

Here is an example from chemistry. In a theoretical model of the cyclohexan C6H12, the six carbon atoms each sit in the center of gravity of a tetrahedron that has two hydrogen atoms and two carbon atoms at its corners. We first normalize and set the length of each edge to 1. Hence, the distances of one fixed carbon atom to each of its immediate neighbours is 1. We will denote the distances to the other three carbon atoms by x, y and z.

A. Dress developed a theory to decide whether a set of points and distances between them can be realized in an n-dimensional space. Here, of course, we have n = 3.

```plaintext
mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) := _
[ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y],
  [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
```

For the cyclohexan, the distances have to satisfy this equation.

\[
\text{eq} := \text{determinant \text{mfzn}} \\
2 \ 2 \ 2 \ 2 \ 22 \ 2 \ 22 \ 2 \ 22 \ 388 \ 250 \ 25 \ 2 \ 250 \ 14575 \\
-x \ y + \frac{-- x \ y - -- x + -- x \ y - --- x \ y - --- x - -- y - --- y + -----}{3 \ 9 \ 3 \ 9 \ 27 \ 9 \ 27 \ 81} \\
\text{Type: DistributedMultivariatePolynomial([x,y,z], Fraction Integer)}
\]

They also must satisfy the equations given by cyclic shifts of the indeterminates.

\[
grobnerFactorize [\text{eq,eval(eq, [x,y,z],[y,z,x]), eval(eq,[x,y,z],[z,x,y])}] \\
[ \\
22 \ 22 \ 22 \ 121 \\
2 \ 22 \ 25 \ 2 \ 22 \ 2 \ 25 \ 2 \ 22 \ 388 \ 250 \\
-x \ z - -- x \ z + -- x + y \ z - -- y \ z + -- y - --- y + ----- \\
3 \ 9 \ 3 \ 9 \ 3 \ 9 \ 27 \\
2 \ 22 \ 2 \ 25 \ 2 \ 22 \ 2 \ 388 \ 250 \\
-y \ z - -- y \ z + -- y - --- y \ z + -- y + --- y + --- z + --- z - -----] \\
3 \ 9 \ 3 \ 9 \ 27 \ 27 \ 81
\]

\[
, 21994 \ 2 \ 21994 \ 4427 \ 463 \\
2 \ 1 \ 11 \ 5 \ 265 \ 2 \ 38 \ 265 \\
[x + y - -----,y - ----- y + -----,z - -----] \\
5625 \ 5625 \ 675 \ 87 \\
2 \ 1 \ 11 \ 5 \ 265 \ 2 \ 38 \ 265 \\
[x - - x \ z - -- x - - z + --,y - z, z - -- z + --],
\]
The union of the solutions of this list is the solution of our original problem. If we impose positivity conditions, we get two relevant ideals. One ideal is zero-dimensional, namely \( x = y = z = \frac{11}{3} \), and this determines the "boat" form of the cyclohexan. The other ideal is one-dimensional, which means that we have a solution space given by one parameter. This gives the "chair" form of the cyclohexan. The parameter describes the angle of the "back of the chair."

groebnerFactorize has an optional Boolean-valued second argument. When it is true partial results are displayed, since it may happen that the calculation does not terminate in a reasonable time. See the source code for GroebnerFactorizationPackage in groebf.spad.pamphlet for more details about the algorithms used.

See Also:
o) display operations groebnerFactorize
o) show GroebnerFactorizationPackage
o) show GroebnerPackage
o) show EuclideanGroebnerBasisPackage

---

GroebnerFactorizationPackage (GBF)

Exports:
`factorGroebnerBasis`  
groebnerFactorization

--- package GBF GroebnerFactorizationPackage ---

```spad
)abbrev package GBF GroebnerFactorizationPackage
++ Author: H. Michael Moeller, Johannes Grabmeier
++ Date Created: 24 August 1989
++ Date Last Updated: 01 January 1992
++ Description:
++ \spadtype{GroebnerFactorizationPackage} provides the function
++ \texttt{groebnerFactor} which uses the factorization routines of Axiom to
++ factor each polynomial under consideration while doing the groebner basis
++ algorithm. Then it writes the ideal as an intersection of ideals
++ determined by the irreducible factors. Note that the whole ring may
++ occur as well as other redundancies. We also use the fact, that from the
++ second factor on we can assume that the preceding factors are
++ not equal to 0 and we divide all polynomials under considerations
++ by the elements of this list of "nonZeroRestrictions".
++ The result is a list of groebner bases, whose union of solutions
++ of the corresponding systems of equations is the solution of
++ the system of equation corresponding to the input list.
++ The term ordering is determined by the polynomial type used.
++ Suggested types include
++ \spadtype{DistributedMultivariatePolynomial},
++ \spadtype{HomogeneousDistributedMultivariatePolynomial},
++ \spadtype{GeneralDistributedMultivariatePolynomial}.

GroebnerFactorizationPackage(\texttt{Dom}, \texttt{Expon}, \texttt{VarSet}, \texttt{Dpol}): \texttt{T} == \texttt{C} where

\texttt{Dom} : Join(\texttt{EuclideanDomain},\texttt{CharacteristicZero})
\texttt{Expon} : \texttt{OrderedAbelianMonoidSup}
\texttt{VarSet} : \texttt{OrderedSet}
\texttt{Dpol}: \texttt{PolynomialCategory}(\texttt{Dom}, \texttt{Expon}, \texttt{VarSet})
\texttt{MF} === \texttt{MultivariateFactorize}(\texttt{VarSet}, \texttt{Expon}, \texttt{Dom}, \texttt{Dpol})
\texttt{sugarPol} === \texttt{Record(totdeg: \texttt{NonNegativeInteger}, \texttt{pol} : \texttt{Dpol})}
\texttt{critPair} === \texttt{Record(lcmfij: \texttt{Expon}, \texttt{totdeg}: \texttt{NonNegativeInteger}, \texttt{poli}: \texttt{Dpol}, \texttt{polj}: \texttt{Dpol})}
\texttt{L} === \texttt{List}
\texttt{B} === \texttt{Boolean}
\texttt{NNI} === \texttt{NonNegativeInteger}
\texttt{OUT} === \texttt{OutputForm}

\texttt{T} === with

\texttt{factorGroebnerBasis} : \texttt{L \texttt{Dpol}} \to \texttt{L \texttt{L \texttt{Dpol}}}
++ \texttt{factorGroebnerBasis(basis)} checks whether the basis contains
++ reducible polynomials and uses these to split the basis.
\texttt{factorGroebnerBasis} : (\texttt{L \texttt{Dpol}}, \texttt{Boolean}) \to \texttt{L \texttt{L \texttt{Dpol}}}
++ \texttt{factorGroebnerBasis(basis,info)} checks whether the basis contains
++ reducible polynomials and uses these to split the basis.
```
++ If argument info is true, information is printed about
++ partial results.
groebnerFactorize : (L Dpol, L Dpol) -> L L Dpol
++ groebnerFactorize(listOfPolys, nonZeroRestrictions) returns
++ a list of groebner basis. The union of their solutions
++ is the solution of the system of equations given by listOfPolys
++ under the restriction that the polynomials of nonZeroRestrictions
++ don’t vanish.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p, a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor
++ in place of p.
groebnerFactorize : (L Dpol, L Dpol, Boolean) -> L L Dpol
++ groebnerFactorize(listOfPolys, nonZeroRestrictions, info) returns
++ a list of groebner basis. The union of their solutions
++ is the solution of the system of equations given by listOfPolys
++ under the restriction that the polynomials of nonZeroRestrictions
++ don’t vanish.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor in place of p.
++ If argument info is true, information is printed about
++ partial results.
groebnerFactorize : L Dpol -> L L Dpol
++ groebnerFactorize(listOfPolys) returns
++ a list of groebner bases. The union of their solutions
++ is the solution of the system of equations given by listOfPolys.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p, a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor
++ in place of p.
++
++X mfzn : SQMATRIX(6,DMP([x,y,z],Fraction INT)) :=
++X [ [0,1,1,1,1,1], [1,0,1,8/3,x,8/3], [1,1,0,1,8/3,y],
++X [1,8/3,1,0,1,8/3], [1,x,8/3,1,0,1], [1,8/3,y,8/3,1,0] ]
++X eq := determinant mfzn
++X groebnerFactorize eq
++X [eq,eval(eq, [x,y,z],[y,z,x]), eval(eq, [x,y,z],[z,x,y])]
groebnerFactorize : (L Dpol, Boolean) -> L L Dpol
++ groebnerFactorize(listOfPolys, info) returns
++ a list of groebner bases. The union of their solutions
++ is the solution of the system of equations given by listOfPolys.
++ At each stage the polynomial p under consideration (either from
++ the given basis or obtained from a reduction of the next S-polynomial)
++ is factorized. For each irreducible factors of p, a
++ new createGroebnerBasis is started
++ doing the usual updates with the factor
++ in place of p.
++ If info is true, information is printed about partial results.

C ==> add

import GroebnerInternalPackage(Dom,Expon,VarSet,Dpol)
-- next to help compiler to choose correct signatures:
info: Boolean
-- signatures of local functions

newPairs : (L sugarPol, Dpol) -> L critPair
++ newPairs(lp, p) constructs list of critical pairs from the list of
++ lp of input polynomials and a given further one p.
++ It uses criteria M and T to reduce the list.
updateCritPairs : (L critPair, L critPair, Dpol) -> L critPair
++ updateCritPairs(lcP1,lcP2,p) applies criterion B to lcP1 using
++ p. Then this list is merged with lcP2.
updateBasis : (L sugarPol, Dpol, NNI) -> L sugarPol
++ updateBasis(li,p,deg) every polynomial in li is dropped if
++ its leading term is a multiple of the leading term of p.
++ The result is this list enlarged by p.
createGroebnerBases : (L sugarPol, L Dpol, L Dpol, L Dpol, L critPair,
++ L L Dpol, Boolean) -> L L Dpol
++ createGroebnerBases(basis, redPols, nonZeroRestrictions, inputPolys,
++ lcP,listOfBases): This function is used to be called from
++ groebnerFactorize.
++ basis: part of a Groebner basis, computed so far
++ redPols: Polynomials from the ideal to be used for reducing,
++ we don't throw away polynomials
++ nonZeroRestrictions: polynomials not zero in the common zeros
++ of the polynomials in the final (Groebner) basis
++ inputPolys: assumed to be in descending order
++ lcP: list of critical pairs built from polynomials of the
++ actual basis
++ listOfBases: Collects the (Groebner) bases constructed by this
++ recursive algorithm at different stages.
++ we print info messages if info is true
createAllFactors: Dpol -> L Dpol
++ factor reduced critpair polynomial

-- implementation of local functions

createGroebnerBases(basis, redPols, nonZeroRestrictions, inputPolys,
++ lcP, listOfBases, info) ==
doSplitting? : B := false
terminateWithBasis : B := false
allReducedFactors : L Dpol := []
nP : Dpol -- actual polynomial under consideration
p : Dpol -- next polynomial from input list
h : Dpol -- next polynomial from critical pairs
stopDividing : Boolean

-- STEP 1 do the next polynomials until a splitting is possible
-- In the first step we take the first polynomial of "inputPolys"
-- if empty, from list of critical pairs "lcP" and do the following:
-- Divide it, if possible, by the polynomials from "nonZeroRestrictions".
-- We factorize it and reduce each irreducible factor with respect to
-- "basis". If 0$Dpol occurs in the list we update the list and continue
-- with next polynomial.
-- If there are at least two (irreducible) factors
-- in the list of factors we finish STEP 1 and set a boolean variable
-- to continue with STEP 2, the splitting step.
-- If there is just one of it, we do the following:
-- If it is 1$Dpol we stop the whole calculation and put
-- [1$Dpol] into the listOfBases
-- Otherwise we update the "basis" and the other lists and continue
-- with next polynomial.

while (not doSplitting?) and (not terminateWithBasis) repeat
  terminateWithBasis := (null inputPolys and null lcP)
  not terminateWithBasis => -- still polynomials left
    -- determine next polynomial "nP"
    nP :=
    not null inputPolys =>
      p := first inputPolys
      inputPolys := rest inputPolys
      -- we know that p is not equal to 0 or 1, but, although,
      -- the inputPolys and the basis are ordered, we cannot assume
      -- that p is reduced w.r.t. basis, as the ordering is only quasi
      -- and we could have equal leading terms, and due to factorization
      -- polynomials of smaller leading terms, hence reduce p first:
      hMonic redPol(p,redPols)
      -- now we have inputPolys empty and hence lcP is not empty:
      -- create S-Polynomial from first critical pair:
      h := sPol first lcP
      lcP := rest lcP
      hMonic redPol(h,redPols)
    nP = 1$Dpol =>
      basis := [[0,1$Dpol]$sugarPol]
      terminateWithBasis := true
    -- if "nP" = 0, then we continue, otherwise we determine next "nP"
    nP ^= 0$Dpol =>
      -- now we divide "nP", if possible, by the polynomials
      -- from "nonZeroRestrictions"
      for q in nonZeroRestrictions repeat
stopDividing := false
until stopDividing repeat
nPq := nP exquo q
stopDividing := (nPq case "failed")
if not stopDividing then nP := autoCoerce nPq
stopDividing := stopDividing or zero? degree nP

zero? degree nP =>
basis := [[0,1$Dpol]$sugarPol]
terminateWithBasis := true -- doSplitting? is still false

-- a careful analysis has to be done, when and whether the
-- following reduction and case nP=1 is necessary

nP := hMonic redPol(nP,redPols)
zero? degree nP =>
basis := [[0,1$Dpol]$sugarPol]
terminateWithBasis := true -- doSplitting? is still false

-- if "nP" ^= 0, then we continue, otherwise we determine next "nP"
nP ^= 0$Dpol =>
-- now we factorize "nP", which is not constant
irreducibleFactors : L Dpol := createAllFactors(nP)
-- if there are more than 1 factors we reduce them and split
(doSplitting? := not null rest irreducibleFactors) =>
-- and reduce and normalize the factors
for fnP in irreducibleFactors repeat
fnP := hMonic redPol(fnP,redPols)
-- no factor reduces to 0, as then "fP" would have been
-- reduced to zero,
-- but 1 may occur, which we will drop in a later version.
allReducedFactors := cons(fnP, allReducedFactors)
-- end of "for fnP in irreducibleFactors repeat"

-- we want that the smaller factors are dealt with first
allReducedFactors := reverse allReducedFactors
-- now the case of exactly 1 factor, but certainly not
-- further reducible with respect to "redPols"
nP := first irreducibleFactors
-- put "nP" into "basis" and update "lcP" and "redPols":
lcP : L critPair := updateCritPairs(lcP,newPairs(basis,nP),nP)
basis := updateBasis(basis,nP,virtualDegree nP)
redPols := concat(redPols,nP)
-- end of "while not doSplitting? and not terminateWithBasis repeat"

-- STEP 2 splitting step
doSplitting? =>
for fnP in allReducedFactors repeat
if fnP ^= 1$Dpol
then
newInputPolys : L Dpol := _
   sort((x,y) -> degree x > degree y , cons(fnP,inputPolys))
listOfBases := createGroebnerBases(basis, redPols, _
   nonZeroRestrictions,newInputPolys,lcP,listOfBases,info)
-- update "nonZeroRestrictions"
nonZeroRestrictions := cons(fnP,nonZeroRestrictions)
else
   if info then
      messagePrint("we terminated with [1"")$OUT
      listOfBases := cons([1$Dpol],listOfBases)
-- we finished with all the branches on one level and hence
-- finished this call of createGroebnerBasis. Therefore
-- we terminate with the actual "listOfBasis" as
-- everything is done in the recursions
listOfBases
-- end of "doSplitting? =>"

-- STEP 3 termination step

-- we found a groebner basis and put it into the list "listOfBases"
-- (auto)reduce each basis element modulo the others
newBasis :=
   minGbasis(sort((x,y)->degree x > degree y,[p.pol for p in basis]))
   now check whether the normalized basis again has reducible
   -- polynomials, in this case continue splitting!
   if info then
      messagePrint("we found a groebner basis and check whether it ")$OUT
      messagePrint("contains reducible polynomials")$OUT
      print(newBasis::OUT)$OUT
   -- here we should create an output form which is reusable by the system
   -- print(convert(newBasis::OUT)$InputForm :: OUT)$OUT
   removeDuplicates append(factorGroebnerBasis(newBasis, info), listOfBases)

cREATEALLFACTORS(p: Dpol) ==
loF : L Dpol := [el.fctr for el in factorList factor(p)$MF]
sort((x,y) -> degree x < degree y, loF)
newPairs(lp : L sugarPol,p : Dpol) ==
totdegreeOfp : NNI := virtualDegree p
-- next list lcP contains all critPair constructed from
-- p and and the polynomials q in lp
lcP: L critPair := _
   --[[sup(degree q, degreeOfp), q, p]$critPair for q in lp]
   [makeCrit(q, p, totdegreeOfp) for q in lp]
   -- application of the criteria to reduce the list lcP
critMTonD1 sort(critpOrder,lcP)
updateCritPairs(oldListOfcritPairs, newListOfcritPairs, p)==
updatD (newListOfcritPairs, critBonD(p,oldListOfcritPairs))
updateBasis(lp, p, deg) == updatF(p,deg,lp)
-- exported functions

factorGroebnerBasis basis == factorGroebnerBasis(basis, false)

factorGroebnerBasis (basis, info) ==
  foundAReducible : Boolean := false
  for p in basis while not foundAReducible repeat
    -- we use fact that polynomials have content 1
    foundAReducible := 1 < #(el.fctr for el in factorList factor(p)$MF)
    not foundAReducible =>
      if info then messagePrint("factorGroebnerBasis: no reducible polynomials in this basis")$OUT
        [basis]
    -- improve! Use the fact that the irreducible ones already
    -- build part of the basis, use the done factorizations, etc.
    if info then messagePrint("factorGroebnerBasis: we found reducible polynomials and continue splitting")$OUT
    createGroebnerBases([],[],[],basis,[],[],info)

groebnerFactorize(basis, nonZeroRestrictions) ==
  groebnerFactorize(basis, nonZeroRestrictions, false)

groebnerFactorize(basis, nonZeroRestrictions, info) ==
  basis = [] => [basis]
  basis := remove((x:Dpol):Boolean +->(x = 0$Dpol),basis)
  basis = [] => [[0$Dpol]]
  -- normalize all input polynomial
  basis := [hMonic p for p in basis]
  member?(1$Dpol,basis) => [[1$Dpol]]
  basis := sort((x,y) +-> degree x > degree y, basis)
  createGroebnerBases([],[],nonZeroRestrictions,basis,[],[],info)

groebnerFactorize(basis) == groebnerFactorize(basis, [], false)
groebnerFactorize(basis,info) == groebnerFactorize(basis, [], info)
package GBINTERN GroebnerInternalPackage

--- GroebnerInternalPackage.input ---

)set break resume
)sys rm -f GroebnerInternalPackage.output
)spool GroebnerInternalPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GroebnerInternalPackage
--R
--R GroebnerInternalPackage(Dom: GcdDomain,Expon: OrderedAbelianMonoidSup,VarSet: OrderedSet,Dpol: PolynomialCategory(Dom,Expon,VarSet)) is a package constructor
--R Abbreviation for GroebnerInternalPackage is GBINTERN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GBINTERN
--R
--R------------------------------- Operations --------------------------------
--R credPol : (Dpol,List(Dpol)) -> Dpol
--R hMonic : Dpol -> Dpol
--R lepol : Dpol -> Integer
--R minGbasis : List(Dpol) -> List(Dpol)
--R prinb : Integer -> Void
--R prinpolINFO : List(Dpol) -> Void
--R prinshINFO : Dpol -> Void
--R redPol : (Dpol,List(Dpol)) -> Dpol
--R critB : (Expon,Expon,Expon) -> Boolean
--R critBonD : (Dpol,List(Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol))) -> Boolean
--R critMonD1 : (Expon,List(Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol))) -> Boolean
--R critT : Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol) -> Boolean
--R critpOrder : (Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol),Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol)) -> Boolean
--R fprindINFO : (Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol),Dpol,Dpol,Integer,Integer,Integer,Integer) -> Integer
--R gbasis : (List(Dpol),Integer,Integer) -> List(Dpol)
--R makeCrit : (Record(totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol),Dpol,NonNegativeInteger) -> Record(Dpol)
--R prinINFO : (Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol),Dpol,Dpol) -> Void
--R redPo : (Dpol,List(Dpol)) -> Record(poli: Dpol,mult: Dom)
--R sPol : Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol) -> Dpol
--R updD : (List(Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol)),List(Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol))) -> List(Record(lcmfij: Expon,totdeg: NonNegativeInteger,poli: Dpol,polj: Dpol))
--R virtualDegree : Dpol -> NonNegativeInteger

--E 1

)spool
)lisp (bye)
--- GroebnerInternalPackage.help ---

====================================================================
GroebnerInternalPackage examples
====================================================================

This package provides low level tools for Groebner basis computations

See Also:
o )show GroebnerInternalPackage

---

GroebnerInternalPackage (GBINTERN)

Exports:
credPol  critB  critBonD  critM  critMTonD1
critMonD1 critpOrder  critT  fprindINFO  gbasis
hMonic  lepol  makeCrit  minGbasis  prinb
prindINFO prinpolINFO  prinshINFO  redPo  redPol
sPol  updatD  updatF  virtualDegree

--- package GBINTERN GroebnerInternalPackage ---

)abbrev package GBINTERN GroebnerInternalPackage
++ Description:
++ This package provides low level tools for Groebner basis computations

GroebnerInternalPackage(Dom, Expon, VarSet, Dpol): T == C where
Dom: GcdDomain
Expon: OrderedAbelianMonoidSup
VarSet: OrderedSet
Dpol: PolynomialCategory(Dom, Expon, VarSet)
NNI == NonNegativeInteger

Definition of Record critPair and Prinp

critPair ==> Record( lcmfij: Expon, totdeg: NonNegativeInteger, poli: Dpol, polj: Dpol )
sugarPol ==> Record( totdeg: NonNegativeInteger, pol: Dpol)
Prinp ==> Record( ci:Dpol,tci:Integer,cj:Dpol,tcj:Integer,c:Dpol, tc:Integer,rc:Dpol,trc:Integer,tF:Integer,tD:Integer)

T== with

credPol: (Dpol, List(Dpol)) -> Dpol
++ credPol undocumented
redPol: (Dpol, List(Dpol)) -> Dpol
++ redPol undocumented
gbasis: (List(Dpol), Integer, Integer) -> List(Dpol)
++ gbasis undocumented
critT: critPair -> Boolean
++ critT undocumented
critM: (Expon, Expon) -> Boolean
++ critM undocumented
critB: (Expon, Expon, Expon) -> Boolean
++ critB undocumented
critBonD: (Dpol, List(critPair)) -> List(critPair)
++ critBonD undocumented
critMTonD1: (List(critPair)) -> List(critPair)
++ critMTonD1 undocumented
critMonD1: (Expon, List(critPair)) -> List(critPair)
++ critMonD1 undocumented
redPo: (Dpol, List(Dpol) ) -> Record(poly:Dpol, mult:Dom)
++ redPo undocumented
hMonic: Dpol -> Dpol
++ hMonic undocumented
updatF: (Dpol, NNI, List(sugarPol) ) -> List(sugarPol)
++ updatF undocumented
sPol: critPair -> Dpol
++ sPol undocumented
updatD: (List(critPair), List(critPair)) -> List(critPair)
++ updatD undocumented
minGbasis: List(Dpol) -> List(Dpol)
++ minGbasis undocumented
lepol: Dpol -> Integer
++ lepol undocumented
prinshINFO : Dpol -> Void
++ prinshINFO undocumented
prindINFO: (critPair, Dpol, Dpol,Integer,Integer,Integer) -> Integer
++ prindINFO undocumented
fprindINFO: (critPair, Dpol, Dpol, Integer,Integer,Integer
\[\text{makeCrit: (sugarPol, Dpol, NonNegativeInteger) -> critPair}\]
\[\text{virtualDegree : Dpol -> NonNegativeInteger}\]
D:= nil
while _^ null Pol1 repeat
  h:= hMonic(first(Pol1))
  Pol1:= rest(Pol1)
  toth := virtualDegree h
  D1:= [makeCrit(x,h,toth) for x in basPols]
  D:= updatD(critMTonD1(sort(critpOrder, D1)),
             critBonD(h,D))
  basPols:= updatF(h,toth,basPols)
D:= sort(critpOrder, D)
xx:= xx2
------ loop

redPols := [x.pol for x in basPols]
while _^ null D repeat
  D0:= first D
  s:= hMonic(sPol(D0))
  D:= rest(D)
  h:= hMonic(redPol(s,redPols))
  if xx1 = 1 then
    prinshINFO(h)
  h = 0 =>
  if xx2 = 1 then
    prindINFO(D0,s,h,# basPols, # D,xx)
    xx:= 2
  " go to top of while "
  degree(h) = 0 =>
  D:= nil
  if xx2 = 1 then
    prindINFO(D0,s,h,# basPols, # D,xx)
    xx:= 2
  basPols:= updatF(h,0,[])
  leave "out of while"
D1:= [makeCrit(x,h,D0.totdeg) for x in basPols]
D:= updatD(critMTonD1(sort(critpOrder, D1)),
            critBonD(h,D))
basPols:= updatF(h,D0.totdeg,basPols)
redPols := concat(redPols,h)
if xx2 = 1 then
  prindINFO(D0,s,h,# basPols, # D,xx)
  xx:= 2
Pol := [x.pol for x in basPols]
if xx2 = 1 then
  prinpolINFO(Pol)
messagePrint(" THE GROEBNER BASIS POLYNOMIALS")
if xx1 = 1 and xx2 ^= 1 then
  messagePrint(" THE GROEBNER BASIS POLYNOMIALS")
Pol

-----------------------------------------------------------
--- erase multiple of e in D2 using crit M

critMonD1(e: Expon, D2: List(critPair)) ==
  null D2 => nil
  x := first(D2)
  critM(e, x.lcmfij) => critMonD1(e, rest(D2))
  cons(x, critMonD1(e, rest(D2)))

----------------------------

--- reduce D1 using crit T and crit M

critMTonD1(D1: List(critPair)) ==
  null D1 => nil
  f1 := first(D1)
  s1 := #(D1)
  cT1 := critT(f1)
  s1 = 1 and cT1 => nil
  s1 = 1 => D1
  e1 := f1.lcmfij
  r1 := rest(D1)
  e1 = (first r1).lcmfij =>
    cT1 => critMTonD1(cons(f1, rest(r1)))
    critMTonD1(r1)
  D1 := critMonD1(e1, r1)
  cT1 => critMTonD1(D1)
  cons(f1, critMTonD1(D1))

----------------------------

--- erase elements in D fullfilling crit B

critBonD(h:Dpol, D: List(critPair)) ==
  null D => nil
  x := first(D)
  critB(degree(h), x.lcmfij, degree(x.poli), degree(x.polj)) =>
    critBonD(h, rest(D))
  cons(x, critBonD(h, rest(D)))

----------------------------

--- concat F and h and erase multiples of h in F

updatF(h: Dpol, deg:NNI, F: List(sugarPol)) ==
  null F => [[deg, h]]
  f1 := first(F)
  critM(degree(h), degree(f1.pol)) => updatF(h, deg, rest(F))
  cons(f1, updatF(h, deg, rest(F)))
--- concat ordered critical pair lists D1 and D2

updatD(D1: List(critPair), D2: List(critPair)) ==
null D1 => D2
null D2 => D1
d11:= first(D1)
d12:= first(D2)
critpOrder(d11,d12) => cons(d11, updatD(D1.rest, D2))
cons(d12, updatD(D1, D2.rest))

--- remove gcd from pair of coefficients

gcdCo(c1:Dom, c2:Dom):Record(co1:Dom, co2:Dom) ==
d:=gcd(c1,c2)
[(c1 exquo d)::Dom, (c2 exquo d)::Dom]

--- calculate S-polynomial of a critical pair

sPol(p:critPair)==
Tij := p.lcmfij
fi := p.poli
fj := p.polj
cc := gcdCo(leadingCoefficient fi, leadingCoefficient fj)
reductum(fi)*monomial(cc.co2,subtractIfCan(Tij, degree fi)::Expon) -
reductum(fj)*monomial(cc.co1,subtractIfCan(Tij, degree fj)::Expon)

--- reduce critpair polynomial mod F
--- iterative version

redPo(s: Dpol, F: List(Dpol)) ==
m:Dom := 1
Fh := F
while _^ ( s = 0 or null F ) repeat
f1:= first(F)
s1:= degree(s)
e: Union(Expon, "failed")
(e:= subtractIfCan(s1, degree(f1))) case Expon =>
cc:=gcdCo(leadingCoefficient f1, leadingCoefficient s)
s:=cc.co1*reductum(s) - monomial(cc.co2,e)*reductum(f1)
m := m*cc.co1
F:= Fh
F:= rest F
[s,m]
\begin{verbatim}
redPol(s: Dpol, F: List(Dpol)) ==  credPol(redPo(s,F).poly,F)

----------------------------
--- crit T  true, if e1 and e2 are disjoint
\end{verbatim}

\begin{verbatim}
critT(p: critPair) == p.lcmfij = (degree(p.poli) + degree(p.polj))

----------------------------
--- crit M - true, if lcm#2 multiple of lcm#1
\end{verbatim}

\begin{verbatim}
critM(e1: Expon, e2: Expon) ==
en: Union(Expon, "failed")
(en:=subtractIfCan(e2, e1)) case Expon

----------------------------
--- crit B - true, if eik is a multiple of eh and eik =\equal lcm(eh,ei) and eik =\equal lcm(eh,ek)
\end{verbatim}

\begin{verbatim}
critB(eh:Expon, eik:Expon, ei:Expon, ek:Expon) ==
critM(eh, eik) and (eik = sup(eh, ei)) and (eik = sup(eh, ek))

----------------------------
--- make polynomial monic case Domain a Field
\end{verbatim}

\begin{verbatim}
hMonic(p: Dpol) ==
p= 0 => p
-- inv(leadingCoefficient(p))*p
primitivePart p

----------------------------
--- reduce all terms of h mod F (iterative version )
\end{verbatim}

\begin{verbatim}
credPol(h: Dpol, F: List(Dpol) ) ==
null F => h
h0:Dpol:= monomial(leadingCoefficient h, degree h)
while (h:=reductum h) ^= 0 repeat
  hred:= redPo(h, F)
  h := hred.poly
  h0:=(hred.mult)*h0 + monomial(leadingCoefficient(h),degree h)
  h0

----------------------------
--- calculate minimal basis for ordered F
\end{verbatim}
minGbasis(F: List(Dpol)) ==
null F => nil
newbas := minGbasis rest F
cons(hMonic credPol( first(F), newbas), newbas)

-----------------------------

---- calculate number of terms of polynomial

lepol(p1:Dpol)==
n: Integer
n:= 0
while p1 ^= 0 repeat
n:= n + 1
p1:= reductum(p1)
n

----- print blanc lines

prinb(n: Integer)==
for x in 1..n repeat
messagePrint("\n")

----- print reduced critpair polynom

prinshINFO(h: Dpol)==
prinb(2)
messagePrint(" reduced Critpair - Polynom :")
prinb(2)
print(h::Ex)
prinb(2)

-----------------------------

----- print info string

prindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer,
i2:Integer, n:Integer) ==
ll: List Prinp
a: Dom
cpi:= cp.poli
cpj:= cp.polj
if n = 1 then
prinb(1)
messagePrint("you choose option -info- ")
messagePrint(" abbrev. for the following information strings are")
messagePrint(" ci => Leading monomial for critpair calculation")
messagePrint(" tci => Number of terms of polynomial i")
messagePrint(" cj => Leading monomial for critpair calculation")
messagePrint(" tcj => Number of terms of polynomial j")
messagePrint(" c => Leading monomial of critpair polynomial")
messagePrint(" tc => Number of terms of critpair polynomial")
messagePrint(" rc => Leading monomial of redcritpair polynomial")
messagePrint(" trc => Number of terms of redcritpair polynomial")
messagePrint(" tF => Number of polynomials in reduction list F")
messagePrint(" tD => Number of critpairs still to do")
prinb(4)
n := 2
prinb(1)
a := 1
ph = 0 =>
pn = 0 =>
   ll := [[monomial(a, degree(cpi)), lepol(cpi),
   monomial(a, degree(cpj)),
   lepol(cpj), ps, 0, ph, 0, i1, i2]$Prinp]
   print(ll::Ex)
   prinb(1)
n
ll := [[monomial(a, degree(cpi)), lepol(cpi),
   monomial(a, degree(cpj)), lepol(cpj), monomial(a, degree(ps)),
   lepol(ps), ph, 0, i1, i2]$Prinp]
   print(ll::Ex)
   prinb(1)
n
ll := [[monomial(a, degree(cpi)), lepol(cpi),
   monomial(a, degree(cpj)), lepol(cpj), monomial(a, degree(ps)),
   lepol(ps), monomial(a, degree(ph)), lepol(ph), i1, i2]$Prinp]
   print(ll::Ex)
   prinb(1)
n
----------------------------------------

---- print the groebner basis polynomials

prinpolINFO(pl: List(Dpol)) ==
n:Integer
n := # pl
prinb(1)
n = 1 =>
   messagePrint(" There is 1 Groebner Basis Polynomial ")
   prinb(2)
   messagePrint(" There are ")
   prinb(1)
   print(n::Ex)
   prinb(1)
   messagePrint(" Groebner Basis Polynomials. ")
   prinb(2)

fprindINFO(cp: critPair, ps: Dpol, ph: Dpol, i1:Integer,
\( i2: \text{Integer}, \ i3: \text{Integer}, \ n: \text{Integer} \) ==

\( \text{ll: List Prinpp} \)
\( a: \text{Dom} \)
\( \text{cpi:= cp.poli} \)
\( \text{cpj:= cp.polj} \)
\( \text{if n = 1 then} \)
\( \text{prinb(1)} \)
\( \text{messagePrint("you choose option -info- ")} \)
\( \text{messagePrint("abbrev. for the following information strings are")} \)
\( \text{messagePrint(" ci => Leading monomial for critpair calculation")} \)
\( \text{messagePrint(" tci => Number of terms of polynomial i")} \)
\( \text{messagePrint(" cj => Leading monomial for critpair calculation")} \)
\( \text{messagePrint(" tcj => Number of terms of polynomial j")} \)
\( \text{messagePrint(" c => Leading monomial of critpair polynomial")} \)
\( \text{messagePrint(" tc => Number of terms of critpair polynomial")} \)
\( \text{messagePrint(" rc => Leading monomial of redcritpair polynomial")} \)
\( \text{messagePrint(" trc => Number of terms of redcritpair polynomial")} \)
\( \text{messagePrint(" tF => Number of polynomials in reduction list F")} \)
\( \text{messagePrint(" tD => Number of critpairs still to do")} \)
\( \text{messagePrint(" tDF => Number of subproblems still to do")} \)
\( \text{prinb(4)} \)
\( n:= 2 \)
\( \text{prinb(1)} \)
\( a:= 1 \)
\( \text{ph = 0 =>} \)
\( \text{ps = 0 =>} \)
\( \text{ll:= [[monomial(a,degree(cpi)),lepol(cpi)],} \)
\( \text{monomial(a,degree(cpj)),} \)
\( \text{lepol(cpj),ps,0,ph,0,i1,i2,i3]$Prinpp] \)
\( \text{print(ll::Ex)} \)
\( \text{prinb(1)} \)
\( n \)
\( \text{ll:= [[monomial(a,degree(cpi)),lepol(cpi)],} \)
\( \text{monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),} \)
\( \text{lepol(ps), ph,0,i1,i2,i3]$Prinpp] \)
\( \text{print(ll::Ex)} \)
\( \text{prinb(1)} \)
\( n \)
\( \text{ll:= [[monomial(a,degree(cpi)),lepol(cpi)],} \)
\( \text{monomial(a,degree(cpj)),lepol(cpj),monomial(a,degree(ps)),} \)
\( \text{lepol(ps),monomial(a,degree(ph)),lepol(ph),i1,i2,i3]$Prinpp] \)
\( \text{print(ll::Ex)} \)
\( \text{prinb(1)} \)
\( n \)

— GBINTERN.dotabb —
package GB GroebnerPackage

-- GroebnerPackage.input --

)set break resume
)sys rm -f GroebnerPackage.output
)spool GroebnerPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 25
s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
--R
--R (1) 45p + 35s - 165b - 36
--R Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 1

--S 2 of 25
s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
--R
--R (2) 35p + 40z + 25t - 27s
--R Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 2

--S 3 of 25
s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
--R
--R (3) 15w + 25p s + 30z - 18t - 165b
--R Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 3

--S 4 of 25
s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
--R
--R (4) - 9w + 15p t + 20z s
--R Type: DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E 4
s5: DMP([w, p, z, t, s, b], FRAC(INT)) := w*p + 2*z*t - 11*b**3

(5) \( w \cdot p + 2z \cdot t - 11b \)

Type: DistributedMultivariatePolynomial([w, p, z, t, s, b], Fraction(Integer))

s6: DMP([w, p, z, t, s, b], FRAC(INT)) := 99*w - 11*b*s + 3*b**2

(6) \( 99w - 11s \cdot b + 3b \)

Type: DistributedMultivariatePolynomial([w, p, z, t, s, b], Fraction(Integer))

s7: DMP([w, p, z, t, s, b], FRAC(INT)) := b**2 + 33/50*b + 2673/10000

(7) \( b^2 + \frac{33}{50}b + \frac{2673}{10000} \)

Type: DistributedMultivariatePolynomial([w, p, z, t, s, b], Fraction(Integer))

sn7 := [s1, s2, s3, s4, s5, s6, s7]

(8) \([45p + 35s - 165b - 36, 35p + 40z + 25t - 27s,\)

(9) \([15w + 25p \cdot s + 30z - 18t - 165b, -9w + 15p \cdot t + 20z \cdot s, w \cdot p + 2z \cdot t - 11b,\)

(10) \([99w - 11s \cdot b + 3b,\]

Type: List(DistributedMultivariatePolynomial([w, p, z, t, s, b], Fraction(Integer)))

groebner(sn7)

(9) \([19, 1323, 31, 153, 49, 1143, 37, 27,\)

(10) \([w + --- b + -----, p - -- b - ----, z + -- b + -----, t - -- b + ---,\)

(11) \([120, 20000, 18, 200, 36, 2000, 15, 250,\)

(12) \([5, 9, 2, 33, 2673,\]

(13) \([s - - b - ----, b + -- b + -----,\)

(14) \([2, 200, 50, 10000,\]

Type: List(DistributedMultivariatePolynomial([w, p, z, t, s, b], Fraction(Integer)))
--S 10 of 25

groebner(sn7,"redcrit")

--R reduced Critpair - Polynom :

--R

--R

--R 5 61 77 7

--R z + - t - s + b +

--R 8 45 24 10

--R

--R reduced Critpair - Polynom :

--R

--R

--R 66 603 278 2 11 672 2277 415881

--R t s - t b + t - s + s b - s - b -

--R 29 1450 435 29 725 7250 725000

--R

--R reduced Critpair - Polynom :

--R

--R

--R 100 2 160 104 37 79

--R t + s - s b - s - b -

--R 189 63 63 105 125

--R

--R reduced Critpair - Polynom :

--R

--R

--R 3 1026 2 5424 2 2529 1326807 12717 660717

--R s - s b - s - s b - s + b -

--R 145 3625 725 362500 6250 3625000

--R

--R reduced Critpair - Polynom :

--R

--R

--R 2 91248294 2 6550614 7087292937 20020838931

--R s b + s - s b + s - b
\begin{verbatim}
--R 128176525  5127061  12817652500  12817652500
--R +
--R  37595502243
--R -
--R  51270610000
--R

reduced Critpair - Polynom:
--R
--R
--R  2  4746183626079988  1015195815329760  30723564870033201
--R s -
--R  987357073521193
--R b -
--R  987357073521193
--R
--R  24683926838029825
--R

reduced Critpair - Polynom:
--R
--R
--R  0
--R

reduced Critpair - Polynom:
--R
--R
--R  16827373608076633182513471  1262793163581645698534964
--R s b +
--R  2306371414246644859914108300
--R s -
--R  5765928561661214978527075
--R b -
--R  91594345205981119652436033
--R +

reduced Critpair - Polynom:
--R
--R
--R  144148214041530374463176875
--R

reduced Critpair - Polynom:
--R
--R
--R  5  9
--R s -
--R b -
\end{verbatim}
--R 2 200
--R
--R reduced Critpair - Polynom:
--R
--R 0
--R
--R reduced Critpair - Polynom:
--R
--R 0
--R
--R
--R THE GROEBNER BASIS POLYNOMIALS
--R
--R (10)
--R [w + --- b + -----, p - -- b - ----, z + -- b + -----, t - -- b + ---,
--R 120 20000 18 200 36 2000 15 250
--R 5 9 2 33 2673
--R s - - b - ---, b + -- b + ------]
--R 2 200 50 10000
--R
--R Type: List(DistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 10

--S 11 of 25
groebner(sn7,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R ci => Leading monomial for critpair calculation
--R tci => Number of terms of polynomial i
--R cj => Leading monomial for critpair calculation
--R tcj => Number of terms of polynomial j
--R c => Leading monomial of critpair polynomial
--R tc => Number of terms of critpair polynomial
--R rc => Leading monomial of redcritpair polynomial
--R trc => Number of terms of redcritpair polynomial
--R tF => Number of polynomials in reduction list F
--R tD => Number of critpairs still to do
--R
--R
--R
There are 6 Grobner Basis Polynomials.
--R (11)
--R [w + --- b + -----, p -- b - ---, z + -- b + -----, t -- b + ---]
--R  120  20000  18  200  36  2000  15  250
--R  5   9   2   33  2673
--R  2   200  50  10000
--R Type: List(DistributedMultivariatePolynomial([w,z,t,s,b],Fraction(Integer)))
--E 11
--S 12 of 25
groebner(sn7,"redcrit","info")
--R
--R reduced Critpair - Polynom :
--R
--R  5  61  77  7
--R  z + t -- s + b +--
--R  8  45  24  10
--R
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R  ci => Leading monomial for critpair calculation
--R  tci => Number of terms of polynomial i
--R  cj => Leading monomial for critpair calculation
--R  tcj => Number of terms of polynomial j
--R  c  => Leading monomial of critpair polynomial
--R  tc  => Number of terms of critpair polynomial
--R  rc  => Leading monomial of redcritpair polynomial
--R  trc => Number of terms of redcritpair polynomial
--R  tF  => Number of polynomials in reduction list F
--R  tD  => Number of critpairs still to do
--R
--R
--R
--R [[ci= p,tci= 4,cj= p,tcj= 4,c= z,tc= 5,rc= z,trc= 5,tF= 4,tD= 3]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R  66  603  278  2  11  672  2277  415881
--R  t s -- t b + ----- t -- s +-- s b - ----- s ---- b -- -------
--R  29  1450  435  29  725  7250  725000
reduced Critpair - Polynom:

\begin{align*}
\text{100} & \quad 2 & \quad 160 & \quad 104 & \quad 37 & \quad 79 \\
\text{t + --- s} & \quad \text{--- s b} & \quad \text{--- s} & \quad \text{--- b} & \quad \text{---} \\
\text{189} & \quad 63 & \quad 63 & \quad 105 & \quad 125 \\
\end{align*}

reduced Critpair - Polynom:

\begin{align*}
\text{3} & \quad 1026 & \quad 2 & \quad 5424 & \quad 2 & \quad 2529 & \quad 1326807 & \quad 12717 & \quad 660717 \\
\text{s} & \quad \text{---- s b} & \quad \text{---- s} & \quad \text{---- s b} & \quad \text{--- s} & \quad \text{--- b} & \quad \text{---} \\
\text{145} & \quad 3625 & \quad 725 & \quad 362500 & \quad 6250 & \quad 3625000 \\
\text{128176525} & \quad 5127061 & \quad 12817652500 & \quad 12817652500 \\
\text{37595502243} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} \\
\text{51270610000} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} \\
\end{align*}

reduced Critpair - Polynom:

\begin{align*}
\text{2} & \quad 91248294 & \quad 2 & \quad 6550614 & \quad 7087292937 & \quad 20020838931 \\
\text{s b} & \quad \text{--- s} & \quad \text{--- s} & \quad \text{--- s b} & \quad \text{--- s b} & \quad \text{--- s} & \quad \text{--- b} & \quad \text{---} \\
\text{128176525} & \quad 5127061 & \quad 12817652500 & \quad 12817652500 & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} \\
\text{37595502243} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} \\
\text{51270610000} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} & \quad \text{---} \\
\end{align*}
reduced Critpair - Polynom :

2 4746183626079988 1015195815329760 30723564870033201
s b s b s b
987357073521193 987357073521193 24683926838029825
+ 3696123458901625353
s b
2468392683802982500
2 2 2
[[ci= b ,tci= 3,cj= s b,tcj= 6,c= s b,tc= 6,rc= s ,trc= 5,tF= 6,tD= 2]]

reduced Critpair - Polynom :

0

2 2 2
[[ci= s b,tci= 6,cj= s ,tcj= 5,c= s ,tc= 7,rc= 0,trc= 0,tF= 6,tD= 1]]

reduced Critpair - Polynom :

1682737360807663182513471 1262793163581645698534964
s b s b
23063714246644859914108300 5765928561661214978527075
+ 91594345205981119652436033
s b
14414821404153037463176875
3 2 2
[[ci= s ,tci= 7,cj= s ,tcj= 5,c= s b,tc= 6,rc= s b,trc= 4,tF= 7,tD= 2]]
reduced Critpair - Polynom:

\[
\begin{array}{c}
5 \\
9 \\
s - \ b - \\
2 \\
200
\end{array}
\]

reduced Critpair - Polynom:

\[
\begin{array}{c}
0 \\
0 \\
0 \\
[ci= b, tci= 3, cj= s, tcj= 4, c= s, tc= 4, rc= s, trc= 3, tF= 6, tD= 2]
\end{array}
\]

reduced Critpair - Polynom:

\[
\begin{array}{c}
0 \\
[ci= s b, tci= 4, cj= s, tcj= 3, c= s, tc= 4, rc= 0, trc= 0, tF= 6, tD= 2]
\end{array}
\]

reduced Critpair - Polynom:

\[
\begin{array}{c}
2 \\
2 \\
[ci= s, tci= 5, cj= s, tcj= 3, c= s b, tc= 4, rc= 0, trc= 0, tF= 6, tD= 0]
\end{array}
\]

There are

6

Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

\[
\begin{array}{c}
w + b + \ldots, p - b - \ldots, z + b + \ldots, t - b + \ldots
\end{array}
\]

\[
\begin{array}{c}
w + b + \ldots, p - b - \ldots, z + b + \ldots, t - b + \ldots
\end{array}
\]

\[
\begin{array}{c}
120 \\
20000 \\
18 \\
200 \\
36 \\
2000 \\
15 \\
250
\end{array}
\]

\[
\begin{array}{c}
5 \\
9 \\
2 \\
33 \\
2673
\end{array}
\]
hs1:= HDMP([w,p,z,t,s,b],FRAC(INT)) := 45*p + 35*s - 165*b - 36

hs2:= HDMP([w,p,z,t,s,b],FRAC(INT)) := 35*p + 40*z + 25*t - 27*s

hs3:= HDMP([w,p,z,t,s,b],FRAC(INT)) := 15*w + 25*p*s + 30*z - 18*t - 165*b**2

hs4:= HDMP([w,p,z,t,s,b],FRAC(INT)) := -9*w + 15*p*t + 20*z*s

hs5:= HDMP([w,p,z,t,s,b],FRAC(INT)) := w*p + 2*z*t - 11*b**3

hs6:= HDMP([w,p,z,t,s,b],FRAC(INT)) := 99*w - 11*b*s + 3*b**2
hs7: HDMP([w,p,z,t,s,b], FRAC(INT)) := b**2 + 33/50*b + 2673/10000
--R
--R 2 33 2673
--R (19) b + -- b + -----
--R 50 10000
--RT: HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b], Fraction(Integer))
--E 19

--S 20 of 25
hsn7:=[hs1,hs2,hs3,hs4,hs5,hs6,hs7]
--R
--R (20)
--R [45p + 35s - 165b - 36, 35p + 40z + 25t - 27s,
--R 2 3 25p s - 165b + 15w + 30z - 18t, 15p t + 20z s - 9w, - 11b + w p + 2z t,
--R 2 2 33 2673
--R - 11s b + 3b + 99w, b + -- b + -----
--R 50 10000
--RT: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 20

--S 21 of 25
groebner(hsn7)
--R
--R (21)
--R [2 33 2673 19 1323 31 153 49 1143
--R b + -- b + -----, w + --- b + -----, p - -- b - ---, z + -- b + ----,
--R 50 10000 120 20000 18 200 36 2000
--R 37 27 5 9
--R t - -- b + ---, s - - b - ---]
--R 15 250 2 200
--RT: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer)))
--E 21

--S 22 of 25
groebner(hsn7,"redcrit")
--R
--R
--R reduced Critpair - Polynom :
--R
--R
--R 5 61 77 7
--R z + - t - -- s + -- b + --
--R 8 45 24 10
--R
--R
--R
--R reduced Critpair - Polynom :
--R
reduced Critpair - Polynom:

\[ \begin{align*}
2 & \quad 216 & \quad 189 & \quad 78 & \quad 99 & \quad 10557 \\
& \quad s & \quad w & \quad t & \quad s & \quad b \\
5 & \quad 100 & \quad 25 & \quad 500 & \quad 12500
\end{align*} \]

reduced Critpair - Polynom:

\[ \begin{align*}
66 & \quad 17541 & \quad 5886 & \quad 10588 & \quad 9273 & \quad 8272413 \\
t & \quad s & \quad t & \quad b & \quad w & \quad t & \quad s & \quad b
\end{align*} \]

\[ \begin{align*}
29 & \quad 725 & \quad 3625 & \quad 3625 & \quad 36250 & \quad 7250000
\end{align*} \]

reduced Critpair - Polynom:

\[ \begin{align*}
2 & \quad 28 & \quad 44 & \quad 143 & \quad 962712 & \quad 420652 & \quad 5166944 \\
t & \quad w & \quad s & \quad b & \quad w & \quad t & \quad s
\end{align*} \]

\[ \begin{align*}
45 & \quad 15 & \quad 725 & \quad 18125 & \quad 90625 & \quad 815625
\end{align*} \]

\[ \begin{align*}
5036339 & \quad 83580953
\end{align*} \]

\[ \begin{align*}
5437500 & \quad 90625000
\end{align*} \]

reduced Critpair - Polynom:

\[ \begin{align*}
33 & \quad 297 & \quad 81 \\
w & \quad b & \quad w & \quad s & \quad b
\end{align*} \]

\[ \begin{align*}
50 & \quad 10000 & \quad 10000
\end{align*} \]

reduced Critpair - Polynom:

\[ \begin{align*}
21 & \quad 33 & \quad 6723 & \quad 2031 & \quad 104247 \\
w & \quad s & \quad t & \quad b & \quad w & \quad s & \quad b
\end{align*} \]

\[ \begin{align*}
100 & \quad 250 & \quad 50000 & \quad 25000 & \quad 5000000
\end{align*} \]
reduced Critpair - Polynom:

\[ \begin{array}{cccccccc}
2373 & 41563 & 17253 & 578853 & 258751 & 11330361 \\
7250 & 36250 & 290000 & 7250000 & 36250000 & \\
2962071220563579 & 1229379913128787 & 4524811449715289 & \\
51061712 & 91248294 & 1516761889 & 481096937 & 5789482077 & \\
5127061 & 128176525 & 1922647875 & 1281765250 & 51270610000 & \\
29924014031872273 & & & & & \\
98138188260880 & 36801820597830 & 490690941304400 & \\
0 & & & & & \\
\end{array} \]
THE GROEBNER BASIS POLYNOMIALS

\[(22)\]
\[2\quad 33\quad 2673\quad 19\quad 1323\quad 31\quad 153\quad 49\quad 1143\]
\[\{b + -- b + ------, w + --- b + ------, p - -- b - ---, z + -- b + ----,\]
\[50\quad 10000\quad 120\quad 20000\quad 18\quad 200\quad 36\quad 2000\]
\[37\quad 27\quad 5\quad 9\]
\[t - -- b + ---, s - - b - ---\}\]
--R  15  250  2  200
--RType: List(HomogeneousDistributedMultivariatePolynomial([w,p,z,t,s,b],Fraction(Integer))
--E  22

--S  23 of  25
groebner(hsn7,"info")
--R
--R you choose option -info-
--R abbrev. for the following information strings are
--R  ci => Leading monomial for critpair calculation
--R  tci => Number of terms of polynomial i
--R  cj => Leading monomial for critpair calculation
--R  tcj => Number of terms of polynomial j
--R  c  => Leading monomial of critpair polynomial
--R  tc  => Number of terms of critpair polynomial
--R  rc  => Leading monomial of redcritpair polynomial
--R  trc => Number of terms of redcritpair polynomial
--R  tF  => Number of polynomials in reduction list F
--R  tD  => Number of critpairs still to do

--R

[[ci=  p,tci= 4,cj=  p,tcj= 4,c=  z,tc= 5,rc=  z,trc= 5,tF= 4,tD= 5]]
--R

2
[[ci=  p s,tci= 5,cj=  p,tcj= 4,c=  z s,tc= 7,rc=  s,trc= 6,tF= 5,tD= 5]]
--R

[[ci=  p t,tci= 3,cj=  p,tcj= 4,c=  z t,tc= 5,rc=  t s,trc= 7,tF= 6,tD= 6]]
--R

3  2
[[ci=  b  ,tci= 3,cj=  b  ,tcj= 3,c=  w p,tc= 4,rc=  t ,trc= 9,tF= 7,tD= 6]]
--R

2  3
[[ci=  s b,tci= 3,cj=  b  ,tcj= 3,c=  b  ,tc= 4,rc=  w b,trc= 4,tF= 8,tD= 7]]
--R

2
[[ci=  s b,tci= 3,cj=  s  ,tcj= 6,c=  s b ,tc= 7,rc=  w s,trc= 6,tF= 9,tD= 9]]
--R

2
[[ci=  s b,tci= 3,cj=  t s,tcj= 7,c=  t b ,tc= 7,rc=  w t,trc= 7,tF= 10,tD= 11]]
--R

--R
There are

6

Groebner Basis Polynomials.

THE GROEBNER BASIS POLYNOMIALS

(23)

\[\begin{array}{cccccccccc}
2 & 33 & 2673 & 19 & 1323 & 31 & 153 & 49 & 1143 \\
[b + \text{--} b + \text{----}, & w + \text{--} b + \text{----}, & p - \text{--} b - \text{---}, & z + \text{--} b + \text{----}, \\
50 & 10000 & 120 & 20000 & 18 & 200 & 36 & 2000 \\
37 & 27 & 5 & 9 \\
t - \text{--} b + \text{----}, & s - \text{--} b - \text{---}] \\
15 & 250 & 2 & 200 \\
\end{array}\]

Type: List(HomogeneousDistributedMultivariatePolynomial([w,p,t,s,b],Fraction(Integer)))

reduced Critpair - Polynom :

5 \ 61 \ 77 \ 7

z + t - s + b +

8 \ 45 \ 24 \ 10

you choose option -info-

abbrev. for the following information strings are

\begin{itemize}
\item \text{ci} => Leading monomial for critpair calculation
\item \text{tci} => Number of terms of polynomial i
\item \text{cj} => Leading monomial for critpair calculation
\item \text{tcj} => Number of terms of polynomial j
\item \text{c} =>Leading monomial of critpair polynomial
\item \text{tc} => Number of terms of critpair polynomial
\item \text{rc} => Leading monomial of redcritpair polynomial
\item \text{trc} => Number of terms of redcritpair polynomial
\item \text{tF} => Number of polynomials in reduction list F
\end{itemize}
tD => Number of critpairs still to do

[[ci= p, tci= 4, cj= p, tcj= 4, c= z, tc= 5, rc= z, trc= 5, tF= 4, tD= 5]]

reduced Critpair - Polynom :

\[
\begin{align*}
2 & 216 \quad 189 \quad 78 \quad 99 \quad 10557 \\
5 & 100 \quad 25 \quad 500 \quad 12500 \\
\end{align*}
\]

[[ci= p, s, tci= 5, cj= p, tcj= 4, c= z, s, tc= 7, rc= s, trc= 6, tF= 5, tD= 5]]

reduced Critpair - Polynom :

\[
\begin{align*}
66 & 17541 \quad 5886 \quad 10588 \quad 9273 \quad 8272413 \\
29 & 725 \quad 3625 \quad 3625 \quad 36250 \quad 7250000 \\
\end{align*}
\]

[[ci= p, t, tci= 3, cj= p, tcj= 4, c= z, t, tc= 5, rc= t, s, trc= 7, tF= 6, tD= 6]]

reduced Critpair - Polynom :

\[
\begin{align*}
2 & 28 \quad 44 \quad 143 \quad 962712 \quad 420652 \quad 5166944 \\
45 & 15 \quad 725 \quad 18125 \quad 90625 \quad 815625 \\
\end{align*}
\]

\[
\begin{align*}
5036339 & \quad 83580953 \\
5437500 & \quad 90625000 \\
\end{align*}
\]
\[
\[ w b + \frac{2 \cdot 3}{50}, \frac{10000}{10000}, \frac{2}{3}, \frac{3}{100}, \frac{25000}{50000}, \frac{100}{250}, \frac{500000}{50000}, \frac{1}{1} \]
\]

\[
\[ w s + \frac{2 \cdot 3}{7250}, \frac{36250}{36250}, \frac{290000}{725000}, \frac{725000}{3625000}, \frac{1}{1} \]
\]

\[
\[ w t + \frac{2 \cdot 3}{2373}, \frac{41563}{17253}, \frac{578853}{258751}, \frac{11330361}{11330361}, \frac{1}{1} \]
\]
reduced Critpair - Polynom :

\[ 2 \]

2

reduced Critpair - Polynom :

2

reduced Critpair - Polynom :

2
reduced Critpair - Polynom :

\[ \begin{align*}
172832706542351932 & \quad 47302810289036749 & \quad 2736061156820726 \\
155991468675747195 & \quad 155991468675747195 & \quad 17332385408416355
\end{align*} \]


2

\[ \begin{align*}
[& \text{ci= s b, tci= 3, cji= t b, tcji= 6, c= t b, tcc= 7, rc= t, trc= 4, tf= 7, tD= 11}] \\
[& \text{ci= s b, tci= 3, cji= w b, tcji= 4, c= w b, tcc= 5, rc= s, trc= 3, tf= 6, tD= 9}] \\
[& \text{ci= w b, tci= 4, cji= t b, tcji= 6, c= w, tcc= 7, rc= 0, trc= 0, tf= 6, tD= 8}] \\
\end{align*} \]

reduced Critpair - Polynom :

\[ \begin{align*}
5 & \quad 9 \\
2 & \quad 200 \\
2
\end{align*} \]

2

\[ \begin{align*}
[& \text{ci= s b, tci= 3, cji= w b, tcji= 4, c= w b, tcc= 5, rc= s, trc= 3, tf= 6, tD= 9}] \\
[& \text{ci= w b, tci= 4, cji= t b, tcji= 6, c= w, tcc= 7, rc= 0, trc= 0, tf= 6, tD= 8}] \\
\end{align*} \]

reduced Critpair - Polynom :

\[ \begin{align*}
0
\end{align*} \]

2

\[ \begin{align*}
[& \text{ci= w b, tci= 4, cji= t b, tcji= 6, c= w, tcc= 7, rc= 0, trc= 0, tf= 6, tD= 8}] \\
\end{align*} \]

reduced Critpair - Polynom :

\[ \begin{align*}
0
\end{align*} \]
reduced Critpair - Polynom :

\[
\begin{align*}
&[[\text{ci} = s, \text{tc} = s, \text{rc} = 3, \text{trc} = 0, \text{tF} = 6, \text{tD} = 7]] \\
&\text{reduced Critpair - Polynom :}
\end{align*}
\]

\[
\begin{align*}
&[[\text{ci} = t, \text{tc} = 4, \text{rc} = 0, \text{trc} = 0, \text{tF} = 6, \text{tD} = 6]] \\
&\text{reduced Critpair - Polynom :}
\end{align*}
\]

\[
\begin{align*}
&[[\text{ci} = w, \text{tc} = 4, \text{rc} = 0, \text{trc} = 0, \text{tF} = 6, \text{tD} = 5]] \\
&\text{reduced Critpair - Polynom :}
\end{align*}
\]

\[
\begin{align*}
&[[\text{ci} = s, \text{tc} = s, \text{rc} = 6, \text{trc} = 0, \text{tF} = 6, \text{tD} = 4]] \\
&\text{reduced Critpair - Polynom :}
\end{align*}
\]

\[
\begin{align*}
&[[\text{ci} = t, \text{tc} = 4, \text{rc} = 0, \text{trc} = 0, \text{tF} = 6, \text{tD} = 3]] \\
&\text{reduced Critpair - Polynom :}
\end{align*}
\]
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R [[ci= w s,tci= 6,cj= w,tcj= 5,c= t s,tc= 8,rc= 0,trc= 0,tF= 6,tD= 2]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R 2
--R [[ci= t ,tci= 9,cj= t,tcj= 4,c= w s,tc= 9,rc= 0,trc= 0,tF= 6,tD= 1]]
--R
--R
--R reduced Critpair - Polynom :
--R
--R 0
--R
--R
--R 2
--R [[ci= w t,tci= 7,cj= w,tcj= 5,c= t ,tc= 8,rc= 0,trc= 0,tF= 6,tD= 0]]
--R
--R
--R There are
--R
--R 6
--R
--R Groebner Basis Polynomials.
--R
--R
--R THE GROEBNER BASIS POLYNOMIALS
--R
--R (24)
--R 2 33 2673 19 1323 31 153 49 1143
--R [b  +  --  b  +  -----, w  +  ---  b  +  ------, p  +  --  b  +  ---, z  +  --  b  +  ----,
--R 50 10000 120 20000 18 200 36 2000
--R 37 27 5 9
--R t  +  ---  b  +  ---, s  +  b  +  ---]
GroebnerPackage computes groebner bases for polynomial ideals. The basic computation provides a distinguished set of generators for polynomial ideals over fields. This basis allows an easy test for membership: the operation normalForm returns zero on ideal members. When the provided coefficient domain, Dom, is not a field, the result is equivalent to considering the extended ideal with Fraction(Dom) as coefficients, but considerably more efficient since all calculations are performed in Dom.

Additional arguments "info" and "redcrit" can be given to provide incremental information during computation.

Argument "info" produces a computational summary for each s-polynomial. Argument "redcrit" prints out the reduced critical pairs.

The term ordering is determined by the polynomial type used. Suggested types include
- DistributedMultivariatePolynomial
- HomogeneousDistributedMultivariatePolynomial
Example to call groebner:

\[
\begin{align*}
\text{s1}: DMP[w,p,z,t,s,b]RN&:= 45p + 35s - 165b - 36 \\
\text{s2}: DMP[w,p,z,t,s,b]RN&:= 35p + 40z + 25t - 27s \\
\text{s3}: DMP[w,p,z,t,s,b]RN&:= 15w + 25ps + 30z - 18t - 165b^2 \\
\text{s4}: DMP[w,p,z,t,s,b]RN&:= -9w + 15pt + 20zs \\
\text{s5}: DMP[w,p,z,t,s,b]RN&:= w^2 + zst - 11b^3 \\
\text{s6}: DMP[w,p,z,t,s,b]RN&:= 99w - 11bst + 3b^2 \\
\text{s7}: DMP[w,p,z,t,s,b]RN&:= b^2 + 33/50b + 2673/10000 \\
\end{align*}
\]

\[sn7 := [s1,s2,s3,s4,s5,s6,s7]\]

\text{groebner(sn7,info)}

groebner calculates a minimal Groebner Basis
all reductions are TOTAL reductions

To get the reduced critical pairs do:

\text{groebner(sn7, "redcrit")}

You can get other information by calling:

\text{groebner(sn7, "info")}

which returns:

\begin{align*}
\text{ci} &\Rightarrow \text{Leading monomial for critpair calculation} \\
\text{tc_i} &\Rightarrow \text{Number of terms of polynomial i} \\
\text{cj} &\Rightarrow \text{Leading monomial for critpair calculation} \\
\text{tc_j} &\Rightarrow \text{Number of terms of polynomial j} \\
\text{c} &\Rightarrow \text{Leading monomial of critpair polynomial} \\
\text{tc} &\Rightarrow \text{Number of terms of critpair polynomial} \\
\text{rc} &\Rightarrow \text{Leading monomial of redcritpair polynomial} \\
\text{tcrc} &\Rightarrow \text{Number of terms of redcritpair polynomial} \\
\text{tF} &\Rightarrow \text{Number of polynomials in reduction list F} \\
\text{tD} &\Rightarrow \text{Number of critpairs still to do}
\end{align*}

See Also:
\text{o display operations groebner}
\text{o show GroebnerPackage}
\text{o show DistributedMultivariatePolynomial}
\text{o show HomogeneousDistributedMultivariatePolynomial}
\text{o show EuclideanGroebnerBasisPackage}
GroebnerPackage (GB)

Exports:
\texttt{groebner}  \texttt{normalForm}

— package GB GroebnerPackage —

\texttt{groebnerPackage} (GB)
groebner: List(Dpol) \rightarrow List(Dpol)
++ groebner(lp) computes a groebner basis for a polynomial ideal
++ generated by the list of polynomials lp.
++
++X s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
++X s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
++X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
++X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
++X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
++X s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
++X s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
++X sn7:=[s1,s2,s3,s4,s5,s6,s7]
++X groebner(sn7)

groebner: ( List(Dpol), String ) \rightarrow List(Dpol)
++ groebner(lp, infoflag) computes a groebner basis
++ for a polynomial ideal
++ generated by the list of polynomials lp.
++ Argument infoflag is used to get information on the computation.
++ If infoflag is "info", then summary information
++ is displayed for each s-polynomial generated.
++ If infoflag is "redcrit", the reduced critical pairs are displayed.
++ If infoflag is any other string,
++ no information is printed during computation.
++
++X s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
++X s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
++X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
++X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
++X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
++X s6:DMP([w,p,z,t,s,b],FRAC(INT)):= 99*w - 11*b*s + 3*b**2
++X s7:DMP([w,p,z,t,s,b],FRAC(INT)):= b**2 + 33/50*b + 2673/10000
++X sn7:=[s1,s2,s3,s4,s5,s6,s7]
++X groebner(sn7,"info")
++X groebner(sn7,"redcrit")

groebner: ( List(Dpol), String, String ) \rightarrow List(Dpol)
++ groebner(lp, "info", "redcrit") computes a groebner basis
++ for a polynomial ideal generated by the list of polynomials lp,
++ displaying both a summary of the critical pairs considered ("info")
++ and the result of reducing each critical pair ("redcrit").
++ If the second or third arguments have any other string value,
++ the indicated information is suppressed.
++
++X s1:DMP([w,p,z,t,s,b],FRAC(INT)):= 45*p + 35*s - 165*b - 36
++X s2:DMP([w,p,z,t,s,b],FRAC(INT)):= 35*p + 40*z + 25*t - 27*s
++X s3:DMP([w,p,z,t,s,b],FRAC(INT)):= 15*w + 25*p*s + 30*z - 18*t - 165*b**2
++X s4:DMP([w,p,z,t,s,b],FRAC(INT)):= -9*w + 15*p*t + 20*z*s
++X s5:DMP([w,p,z,t,s,b],FRAC(INT)):= w*p + 2*z*t - 11*b**3
++X s6: DMP([w, p, z, t, s, b], FRAC(INT)) := 99*w - 11*b*s + 3*b**2
++X s7: DMP([w, p, z, t, s, b], FRAC(INT)) := b**2 + 33/50*b + 2673/10000
++X sn7 := [s1, s2, s3, s4, s5, s6, s7]
++X groebner(sn7, "info", "redcrit")

if Dom has Field then
  normalForm: (Dpol, List(Dpol)) -> Dpol
  ++ normalForm(poly, gb) reduces the polynomial poly modulo the
  ++ precomputed groebner basis gb giving a canonical representative
  ++ of the residue class.

C== add
  import OutputForm
  import GroebnerInternalPackage(Dom, Expon, VarSet, Dpol)

  if Dom has Field then
    monicize(p: Dpol): Dpol ==
    -- one?(lc := leadingCoefficient p) => p
    ((lc := leadingCoefficient p) = 1) => p
    inv(lc)*p
    normalForm(p : Dpol, l : List(Dpol)) : Dpol ==
    redPol(p, map(monicize, l))

    ------ MAIN ALGORITHM GROEBNER ------------------------
    groebner( Pol: List(Dpol) ) ==
    Pol=[] => Pol
    Pol:=[x for x in Pol | x ^= 0]
    Pol=[] => [0]
    minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol, 0, 0))

    groebner( Pol: List(Dpol), xx1: String) ==
    Pol=[] => Pol
    Pol:=[x for x in Pol | x ^= 0]
    Pol=[] => [0]
    xx1 = "redcrit" =>
    minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol, 1, 0)))
    xx1 = "info" =>
    minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol, 2, 1)))
    messagePrint(" ")
    messagePrint("WARNING: options are - redcrit and/or info - ")
    messagePrint(" you didn’t type them correct")
    messagePrint(" please try again")
    messagePrint(" ")
    []

    groebner( Pol: List(Dpol), xx1: String, xx2: String) ==
    Pol=[] => Pol
    Pol:=[x for x in Pol | x ^= 0]
    Pol=[] => [0]
(xx1 = "redcrit" and xx2 = "info") or
(xx1 = "info" and xx2 = "redcrit") =>
minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,1)))
xx1 = "redcrit" and xx2 = "info" =>
minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,1,0)))
xx1 = "info" and xx2 = "redcrit" =>
minGbasis(sort((x,y) +-> degree x > degree y, gbasis(Pol,2,1)))
messagePrint(""")
messagePrint("WARNING: options are - redcrit and/or info - ")
messagePrint(" you didn't type them correctly")
messagePrint(" please try again ")
messagePrint(""")

|---

— GB.dotabb —

"GB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GB"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"GB" -> "PFECAT"
"GB" -> "STRING"

|---

package GROEBSOL GroebnerSolve

|--- GroebnerSolve.input ---|

)set break resume
)sys rm -f GroebnerSolve.output
)spool GroebnerSolve.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show GroebnerSolve
--R
--R GroebnerSolve(lv: List(Symbol),F: GcdDomain,R: GcdDomain) is a package constructor
--R Abbreviation for GroebnerSolve is GROEBSOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GROEBSOL
Operations

\[ \text{genericPosition} : \left( \text{List}(\text{DistributedMultivariatePolynomial}(\text{lv},F)), \text{List}(\text{OrderedVariableList}(\text{lv})) \right) \rightarrow \text{Record}(\text{dpolys}: \text{List}(\text{DistributedMultivariatePolynomial}(\text{lv},F)), \text{coords}: \text{List}(\text{Integer})) \]

\[ \text{groebSolve} : \left( \text{List}(\text{DistributedMultivariatePolynomial}(\text{lv},F)), \text{List}(\text{OrderedVariableList}(\text{lv})) \right) \rightarrow \text{List}(\text{List}(\text{DistributedMultivariatePolynomial}(\text{lv},F))) \]

\[ \text{testDim} : \left( \text{List}(\text{HomogeneousDistributedMultivariatePolynomial}(\text{lv},F)), \text{List}(\text{OrderedVariableList}(\text{lv})) \right) \rightarrow \text{Union}(\text{List}(\text{HomogeneousDistributedMultivariatePolynomial}(\text{lv},F)), \text{"failed"}) \]

— GroebnerSolve.help —

====================================================================

GroebnerSolve examples
====================================================================

Solve systems of polynomial equations using Groebner bases Total order
Groebner bases are computed and then converted to lex ones.
This package is mostly intended for internal use.

See Also:
c )show GroebnerSolve

---

GroebnerSolve (GROEBSOL)

Exports:
genericPosition groebSolve testDim
\textbf{--- package GROEBSOL GroebnerSolve ---}

)abbrev package GROEBSOL GroebnerSolve
++ Author : P.Gianni, Summer '88, revised November '89
++ Description:
++ Solve systems of polynomial equations using Groebner bases
++ Total order Groebner bases are computed and then converted to lex ones
++ This package is mostly intended for internal use.

GroebnerSolve(lv,F,R) : C == T

where
R : GcdDomain
F : GcdDomain
lv : List Symbol

NNI ==> NonNegativeInteger
I ==> Integer
S ==> Symbol
OV ==> OrderedVariableList(lv)
IES ==> IndexedExponents Symbol

DP ==> DirectProduct(#lv,NonNegativeInteger)
DPoly ==> DistributedMultivariatePolynomial(lv,F)

HDP ==> HomogeneousDirectProduct(#lv,NonNegativeInteger)
HDPoly ==> HomogeneousDistributedMultivariatePolynomial(lv,F)

SUP ==> SparseUnivariatePolynomial(DPoly)
L ==> List
P ==> Polynomial

\textbf{C == with}

groebSolve : (L DPoly,L OV) -> L L DPoly
++ groebSolve(lp,lv) reduces the polynomial system lp in variables lv
++ to triangular form. Algorithm based on groebner bases algorithm
++ with linear algebra for change of ordering.
++ Preprocessing for the general solver.
++ The polynomials in input are of type \spadtype{DMP}.

testDim : (L HDPoly,L OV) -> Union(L HDPoly,"failed")
++ testDim(lp,lv) tests if the polynomial system lp
++ in variables lv is zero dimensional.

genericPosition : (L DPoly, L OV) -> Record(dpolys:L DPoly, coords: L I)
++ genericPosition(lp,lv) puts a radical zero dimensional ideal
++ in general position, for system lp in variables lv.
T == add
import PolToPol(lv,F)
import GroebnerPackage(F,DP,OV,DPoly)
import GroebnerInternalPackage(F,DP,OV,DPoly)
import GroebnerPackage(F,HDP,OV,HDPoly)
import LinGroebnerPackage(lv,F)

nv:NNI:=#lv

---- test if f is power of a linear mod (rad lpol) ----
--- f is monic ---
testPower(uf:SUP,x:OV,lpol:L DPoly) : Union(DPoly,"failed") ==
df:=degree(uf)
trailp:=coefficient(uf,(df-1)::NNI)
(testquo := trailp exquo (df::F)) case "failed" => "failed"
trailp := testquo::DPoly
gg:=gcd(lc:=leadingCoefficient(uf),trailp)
trailp := (trailp exquo gg)::DPoly
lc := (lc exquo gg)::DPoly
linp:=monomial(lc,1$NNI)$SUP + monomial(trailp,0$NNI)$SUP
g:=DPoly:=multivariate(uf-linp**df,x)
redPol(g,lpol) ^= 0 => "failed"
multivariate(linp,x)

-- is the 0-dimensional ideal I in general position ? --
--- internal function ---
testGenPos(lpol:L DPoly,lvar:L OV):Union(L DPoly,"failed") ==
rlpol:=reverse lpol
f:=rlpol.first
#lvar=1 => [f]
rlvar:=rest reverse lvar
newlpol:List(DPoly):=[f]
for f in rlpol.rest repeat
  x:=first rlvar
  fi:=univariate(f,x)
  if (mainVariable leadingCoefficient fi case "failed") then
    if ((g:= testPower(fi,x,newlpol)) case "failed")
      then return "failed"
    newlpol :=concat(redPol(g::DPoly,newlpol),newlpol)
  rlvar:=rest rlvar
else if redPol(f,newlpol)="0 then return"failed"
newlpol

-- change coordinates and out the ideal in general position ----
rlvar:=reverse lvar
lnp:=[dmpToHdmp(f) for f in lp]
x := first rlvar;rlvar:=rest rlvar
testfail:=true
for count in 1.. while testfail repeat
    ranvals:L I:=I+[1+(random(I) mod count)] for vv in rlvar
    val:=val+[/rv*(vv::HDPoly)
        for vv in rlvar for rv in ranvals]
    val:=val+x::HDPoly
    gb:L HDPoly:= elt(univariate(p,x),val) for p in lnp
    gb:=totolex gb
    (gbt:=testGenPos(gbt,lvar)) case "failed"=>"try again"
    testfail:=false
    [gb,gbt,ranvals,dmpToHdmp(last (gb1::L DPoly))]

genericPosition(lp:L DPoly,lvar:L OV) ==
    nans:=genPos(lp,lvar)
    [nans.lpolys, nans.coord]

---- select the univariate factors
select(lup:L L HDPoly) : L L HDPoly ==
    lup=[] => list []
    [:[cons(f,lsel) for lsel in select lup.rest] for f in lup.first]

---- in the non generic case, we compute the prime ideals ----
---- associated to leq, basis is the algebra basis ----
findCompon(leq:L HDPoly,lvar:L OV):L L DPoly ==
    teq:=totolex(leq)
    #teq = #lvar => [teq]
    -- "((teq1:=testGenPos(teq,lvar)) case "failed") => [teq1::L DPoly]
    gp:=genPos(teq,lvar)
    lgp:= gp.polys
    g:HDPoly:=gp.univp
    fg:=(factor g)$GeneralizedMultivariateFactorize(OV,HDP,R,F,HDPoly)
    lfact:=[ff.factor for ff in factors(fg::Factored(HDPoly))]
    result: L L HDPoly := []
    #lfact=1 => [teq]
    for tfact in lfact repeat
        tlfact:=concat(tfact,lgp)
        result:=concat(tlfact,result)
        ranvals:L I:=I+I for vv in rlvar
        x:=first rlvar
        rlvar:=rest rlvar
        val:=val+[/rv*(vv::HDPoly) for vv in rlvar for rv in ranvals]
        val:=(x::HDPoly)-val
        ans:=[totolex groebner elt(univariate(p,x),val) for p in lp]
            for lp in result]
    [ll for ll in ans | ll^=[1]]

zeroDim?(lp: List HDPoly,lvar:L OV) : Boolean ==
    empty? lp => false
n:NNI := #lvar
#lp < n => false
lvint1 := lvar
for f in lp while not empty?(lvint1) repeat
  g:= f - reductum f
  x:=mainVariable(g)::OV
  if ground?(leadingCoefficient(univariate(g,x))) then
    lvint1 := remove(x, lvint1)
  end if
end for

-- general solve, gives an error if the system not 0-dimensional
groebSolve(leq: L DPoly,lvar:L OV) : L L DPoly ==
  ln := [dmpToHdmp(f) for f in leq]
  leq1:=groebner ln
  #(leq1) = 1 and first(leq1) = 1 => list empty()
  ^(zeroDim?(leq1,lvar)) =>
    error "system does not have a finite number of solutions"
  -- add computation of dimension, for a more useful error
  basis:=computeBasis(leq1)
  lup:L HDPoly:=[]
  llfact:L Factored(HDPoly):=[]
  for x in lvar repeat
    g:=minPol(leq1,basis,x)
    fg:=(factor g)$GeneralizedMultivariateFactorize(OV,HDP,R,F,HDPoly)
    llfact:=concat(fg::Factored(HDPoly),llfact)
    if degree(g,x) = #basis then leave "stop factoring"
  end for
  result: L L DPoly := []
  -- selecting a factor from the lists of the univariate factors
  lfact:=select [[ff.factor for ff in factors llf]
    for llf in llfact]
  for tfact in lfact repeat
    tfact:=groebner concat(tfact,leq1)
    tfact=[1] => "next value"
    result:=concat(result,findCompon(tfact,lvar))
  end for
end groebSolve

testDim(leq : L HDPoly,lvar : L OV) : Union(L HDPoly,"failed") ==
  leq1:=groebner leq
  #(leq1) = 1 and first(leq1) = 1 => empty()
  ^(zeroDim?(leq1,lvar)) => "failed"
  leq1

-- GROEBSOL.dotabb --
package GUESS Guess

— Guess.input —

)set break resume
/sys rm -f Guess.output
/spool Guess.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Guess

--R
--R Guess(F: Field, S: GcdDomain, EXPRR, R: Join(OrderedSet, IntegralDomain), retract: (R -> F), coerce: (F -> EXPRR),
--R   EXPRR: Join(FunctionSpace(Integer), IntegralDomain, RetractableTo(R), RetractableTo(Symbol), RetractableTo(Integer))
--R   ?*? : (%,%) -> %
--R   ?/? : (%,%) -> %
--R   ?**? : (%,%) -> %
--R   numerator : % -> %
--R   denominator : % -> %
--R   ground? : % -> Boolean is a package constructor
--R Abbreviation for Guess is GUESS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESS

--R--------------------------------------------------------------- Operations --------------------------
--R diffHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(F) -> Stream(UnivariateFormalPowerSeries(F)))
--R guess : List(F) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guess : (List(F),List(GuessOption)) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guess : (List(F),List(GuessOption)) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guessADE : List(F) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guessADE : (List(F),List(GuessOption)) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guessADE : Symbol -> (List(F),List(GuessOption)) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guessAlg : List(F) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guessAlg : (List(F),List(GuessOption)) -> List(Record(function: EXPRR, order: NonNegativeInteger))
--R guessBinRat : List(F) -> List(Record(function: EXPRR, order: NonNegativeInteger))
The packages defined in this file enable {Axiom} to guess formulas for sequences of, for example, rational numbers or rational functions, given the first few terms. It extends and complements Christian Krattenthaler’s program Rate and the relevant parts of Bruno Salvy and Paul Zimmermann’s GFUN.

This package implements guessing of sequences. Packages for the most common cases are provided as GuessInteger, GuessPolynomial, etc.
Guess (GUESG)

Exports:
diffHP  guess  guessADE  guessAlg  guessBinRat
guessExpRat  guessHP  guessHolo  guessPRec  guessPade
guessRat  guessRec  shiftHP

The original code would not compile. This is a temporary replacement with changes marked with my initials. I will pick up the latest version sometime in the future and hope it compiles.

(Tim Daly)

— package GUESG Guess —

)abbrev package GUESG Guess
++ Author: Martin Rubey, Timothy Daly
++ Description:
++ This package implements guessing of sequences. Packages for the
++ most common cases are provided as \\spadtype{GuessInteger},
++ \\spadtype{GuessPolynomial}, etc.

Guess(F, S, EXPRR, R, retract, coerce): Exports == Implementation where
  F: Field -- zB.: FRAC POLY PF 5
  -- in F we interpolate und check
  S: GcdDomain
  -- in guessExpRat I would like to determine the roots of polynomials in F. When
  -- F is a quotientfield, I can get rid of the denominator. In this case F is
  -- roughly QFCAT S
  R: Join(OrderedSet, IntegralDomain) -- zB.: FRAC POLY INT
  -- results are given as elements of EXPRR
  -- EXPRR: Join(ExpressionSpace, IntegralDomain,
  --       EXPRR: Join(FunctionSpace Integer, IntegralDomain,
  --                      RetractableTo R, RetractableTo Symbol,
RetractableTo Integer, CombinatorialOpsCategory,
   PartialDifferentialRing Symbol) with
   _* : (%,%) -> %
   _/ : (%,%) -> %
   _*_* : (%,%) -> %
   numerator : % -> %
   denominator : % -> %
   ground? : % -> Boolean

   -- zB.: EXPR INT
   -- EXPR FRAC POLY INT is forbidden. Thus i cannot just use EXPR R

   -- EXPRR exists, in case at some point there is support for EXPR PF 5.

   -- the following I really would like to get rid of
   retract: R -> F -- zB.: i+->i
   coerce: F -> EXPRR -- zB.: i+->i
   -- attention: EXPRR ~= EXPR R

   LGOPT ==> List GuessOption
   GOPT0 ==> GuessOptionFunctions0

   NNI ==> NonNegativeInteger
   PI ==> PositiveInteger
   EXPRI ==> Expression Integer
   GUESSRESULT ==> List Record(function: EXPRR, order: NNI)

   UFPSF ==> UnivariateFormalPowerSeries F
   UFPS1 ==> UnivariateFormalPowerSeriesFunctions

   UFSS ==> UnivariateFormalPowerSeries S

   SUP ==> SparseUnivariatePolynomial

   UFPPSSUPF ==> UnivariateFormalPowerSeries SUP F

   FFFG ==> FractionFreeFastGaussian
   FFFGF ==> FractionFreeFastGaussianFractions

   -- CoeffAction
   DIFFSPECA ==> (NNI, NNI, SUP S) -> S
   DIFFSPECAF ==> (NNI, NNI, UFPSSUPF) -> SUP F
   DIFFSPECAX ==> (NNI, Symbol, EXPRR) -> EXPRR

   -- the diagonal of the C-matrix
   DIFFSPECC ==> NNI -> List S
HPSPEC ==> Record(guessStream: UFPSF -> Stream UFPSF,
degreeStream: Stream NNI,
testStream: UFPSSUPF -> Stream UFPSSUPF,
exprStream: (EXPRR, Symbol) -> Stream EXPRR,
A: DIFFSPECA,
AF: DIFFSPECAF,
AX: DIFFSPECAx,
C: DIFFSPECC)

-- note that empty?(guessStream.o) has to return always. In other words, if the
-- stream is finite, empty? should recognize it.

DIFFSPECN ==> EXPRR -> EXPRR  -- eg.: i+->q^i

GUESSER ==> (List F, LGOPT) -> GUESSRESULT

FSUPS ==> Fraction SUP S
FSUPF ==> Fraction SUP F

V ==> OrderedVariableList([''a1, 'A])
POLYF ==> SparseMultivariatePolynomial(F, V)
FPOLYF ==> Fraction POLYF
FSUPFPOLYF ==> Fraction SUP FPOLYF
POLYS ==> SparseMultivariatePolynomial(S, V)
FPOLYS ==> Fraction POLYS
FSUPFPOLYS ==> Fraction SUP FPOLYS

--@<<implementation: Guess - Hermite-Pade - Types for Operators>>
-- EXT ==> (Integer, V, V) -> FPOLYS
-- EXTEXPR ==> (Symbol, F, F) -> EXPRR
Exports == with

guess: List F -> GUESSRESULT
++ \spad{guess l} applies recursively \spadfun{guessRec} and
++ \spadfun{guessADE} to the successive differences and quotients of
++ the list. Default options as described in
++ \spadtype{GuessOptionFunctions0} are used.

guess: (List F, LGOPT) -> GUESSRESULT
++ \spad{guess(l, options)} applies recursively \spadfun{guessRec}
++ and \spadfun{guessADE} to the successive differences and quotients
++ of the list. The given options are used.

guess: (List F, List GUESSER, List Symbol) -> GUESSRESULT
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ guessSum and quotients if ops contains the symbol guessProduct to
++ the list. Default options as described in
\spadtype{GuessOptionFunctions0} are used.

\begin{verbatim}
++ \spad{guess} is used.

\spad{guess: (List F, List GUESSER, List Symbol, LGOPT) -> GUESSRESULT}
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ \spad{guessSum} and quotients if ops contains the symbol
++ \spad{guessProduct} to the list. The given options are used.

\spad{guessExpRat: List F -> GUESSRESULT}
++ \spad{guessExpRat l} tries to find a function of the form
++ \( n+\rightarrow (a+b n)^n r(n) \), where \( r(n) \) is a rational function, that fits
++ \( l \).

\spad{guessExpRat: (List F, LGOPT) -> GUESSRESULT}
++ \spad{guessExpRat(l, options)} tries to find a function of the
++ form \( n+\rightarrow (a+b n)^n r(n) \), where \( r(n) \) is a rational function, that
++ fits \( l \).

\spad{guessBinRat: List F -> GUESSRESULT}
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form \( n+\rightarrow \text{binomial}(a+b n, n) r(n) \), where \( r(n) \) is a rational
++ function, that fits \( l \).

\spad{guessBinRat: (List F, LGOPT) -> GUESSRESULT}
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form \( n+\rightarrow \text{binomial}(a+b n, n) r(n) \), where \( r(n) \) is a rational
++ function, that fits \( l \).

if \( F \) has \spad{RetractableTo Symbol} and \( S \) has \spad{RetractableTo Symbol} then

\spad{guessExpRat: Symbol -> GUESSER}
++ \spad{guessExpRat q} returns a guesser that tries to find a
++ function of the form \( n+\rightarrow (a+b q^n)^n r(q^n) \), where \( r(q^n) \) is a
++ \( q \)-rational function, that fits \( l \).

\spad{guessBinRat: Symbol -> GUESSER}
++ \spad{guessBinRat q} returns a guesser that tries to find a
++ function of the form \( n+\rightarrow q\text{binomial}(a+b n, n) r(n) \), where \( r(q^n) \) is a
++ \( q \)-rational function, that fits \( l \).

\spad{guessHP: (LGOPT -> HPSPEC) -> GUESSER}
++ \spad{guessHP f} constructs an operation that applies Hermite-Pade
++ approximation to the series generated by the given function \( f \).

\spad{guessADE: List F -> GUESSRESULT}
++ \spad{guessADE l} tries to find an algebraic differential equation
++ for a generating function whose first Taylor coefficients are
++ given by \( l \), using the default options described in
++ \spad{GuessOptionFunctions0}.
\end{verbatim}
guessADE: (List F, LGOPT) -> GUESSRESULT
   + \spad{guessADE(l, options)} tries to find an algebraic
equation for a generating function whose first Taylor
coefficients are given by \spad{l}, using the given options.

guessAlg: List F -> GUESSRESULT
   + \spad{guessAlg l} tries to find an algebraic equation for a
generating function whose first Taylor coefficients are given by
\spad{l}, using the default options described in
\spadtype{GuessOptionFunctions0}. It is equivalent to
\spadfun{guessADE}(l, \text{maxDerivative} == 0).

guessHolo: List F -> GUESSRESULT
   + \spad{guessHolo l} tries to find an ordinary linear differential
equation for a generating function whose first Taylor coefficients
are given by \spad{l}, using the default options described in
\spadtype{GuessOptionFunctions0}. It is equivalent to
\spadfun{guessADE}\spad{(l, \text{maxPower} == 1)}.

guessHolo: (List F, LGOPT) -> GUESSRESULT
   + \spad{guessHolo(l, options)} tries to find an ordinary linear
differential equation for a generating function whose first Taylor
coefficients are given by \spad{l}, using the given options. It is
equivalent to \spadfun{guessADE}\spad{(l, options)} with \spad{maxPower == 1}.

guessPade: (List F, LGOPT) -> GUESSRESULT
   + \spad{guessPade(l, options)} tries to find a rational function
whose first Taylor coefficients are given by \spad{l}, using the given
options. It is equivalent to \spadfun{guessADE}\spad{(l, \text{maxDerivative} == 0, \text{maxPower} == 1, \text{allDegrees} == true)}.

guessPade: List F -> GUESSRESULT
   + \spad{guessPade(l, options)} tries to find a rational function
whose first Taylor coefficients are given by \spad{l}, using the default
options described in \spadtype{GuessOptionFunctions0}. It is
equivalent to \spadfun{guessADE}\spad{(l, options)} with
\spad{maxDerivative == 0, maxPower == 1, allDegrees == true}.

guessRec: List F -> GUESSRESULT
   + \spad{guessRec l} tries to find an ordinary difference equation
whose first values are given by \spad{l}, using the default options
described in \spadtype{GuessOptionFunctions0}.
guessRec: \((\text{List } F, \text{LGOPT}) \to \text{GUESSRESULT}\)  
\quad \text{tries to find an ordinary difference equation whose first values are given by } l, \text{ using the given options.}

guessPRec: \((\text{List } F, \text{LGOPT}) \to \text{GUESSRESULT}\)  
\quad \text{tries to find a linear recurrence with polynomial coefficients whose first values are given by } l, \text{ using the given options. It is equivalent to } \text{\spadfun{guessRec}(l, \text{options}) with } \text{\spad{maxPower == 1}.}

guessPRec: \((\text{List } F) \to \text{GUESSRESULT}\)  
\quad \text{tries to find a linear recurrence with polynomial coefficients whose first values are given by } l, \text{ using the default options described in } \text{\spadtype{GuessOptionFunctions0}. It is equivalent to } \text{\spadfun{guessRec}(l, \text{maxPower == 1}).}

guessRat: \((\text{List } F, \text{LGOPT}) \to \text{GUESSRESULT}\)  
\quad \text{tries to find a rational function whose first values are given by } l, \text{ using the given options. It is equivalent to } \text{\spadfun{guessRec}(l, \text{maxShift == 0, maxPower == 1, allDegrees == true}).}

guessRat: \((\text{List } F) \to \text{GUESSRESULT}\)  
\quad \text{tries to find a rational function whose first values are given by } l, \text{ using the default options described in } \text{\spadtype{GuessOptionFunctions0}. It is equivalent to } \text{\spadfun{guessRec}(l, \text{maxShift == 0, maxPower == 1, allDegrees == true}).}

diffHP: \((\text{LGOPT}) \to \text{HPSPEC}\)  
\quad \text{returns a specification for Hermite-Pade approximation with the differential operator}

shiftHP: \((\text{LGOPT}) \to \text{HPSPEC}\)  
\quad \text{returns a specification for Hermite-Pade approximation with the shift operator}

if F has RetractableTo Symbol and S has RetractableTo Symbol then

\begin{verbatim}
shiftHP: Symbol -> (LGOPT -> HPSPEC)
  \quad \text{returns a specification for } \text{Hermite-Pade approximation with the } \text{$q$-shift operator}

diffHP: Symbol -> (LGOPT -> HPSPEC)
  \quad \text{returns a specification for } \text{Hermite-Pade approximation with the } \text{$q$-dilation operator}
\end{verbatim}

guessRec: Symbol -> GUESSER
++ \spad{guessRec q} returns a guesser that finds an ordinary ++ q-difference equation whose first values are given by \text{\textit{l}}, using ++ the given options.

guessPRec: Symbol -> GUESSER
++ \spad{guessPRec q} returns a guesser that tries to find ++ a linear q-recurrence with polynomial coefficients whose first ++ values are given by \text{\textit{l}}, using the given options. It is ++ equivalent to \spadfun{guessRec}(\text{\textit{q}}) with ++ \spad{\text{\textit{maxPower}} \text{\textit{==}} 1}.

guessRat: Symbol -> GUESSER
++ \spad{guessRat q} returns a guesser that tries to find a ++ q-rational function whose first values are given by \text{\textit{l}}, using ++ the given options. It is equivalent to \spadfun{guessRec} with ++ \spad{(\text{\textit{l}}, \text{\textit{maxShift}} \text{\textit{==}} 0, \text{\textit{maxPower}} \text{\textit{==}} 1, \text{\textit{allDegrees}} \text{\textit{==}} \text{\textit{true}})}.

guessADE: Symbol -> GUESSER
++ \spad{guessADE q} returns a guesser that tries to find an ++ algebraic differential equation for a generating function whose ++ first Taylor coefficients are given by \text{\textit{l}}, using the given ++ options.

-- We have to put this chunk at the beginning, because otherwise it will take -- very long to compile.

ord1(x: List Integer, i: Integer): Integer ==
n := \#x - 3 - i
x.(n+1)*reduce(_+, [x.j for j in 1..n], 0) + _
2*reduce(_+, [reduce(_+, [x.k*x.j for k in 1..j-1], 0) _
for j in 1..n], 0)

ord2(x: List Integer, i: Integer): Integer ==
if zero? i then
  n := \#x - 3 - i
  ord1(x, i) + reduce(_+, [x.j for j in 1..n], 0)*x.(n+2)-x.(n+1)
else
  ord1(x, i)

deg1(x: List Integer, i: Integer): Integer ==
m := \#x - 3
(x.(m+3)+x.(m+1)+x.(1+i))*reduce(_+, [x.j for j in 2+i..m], 0) + _
x.(m+3)*x.(m+1) + _
2*reduce(_+, [reduce(_+, [x.k*x.j for k in 2+i..j-1], 0) _
for j in 2+i..m], 0)

deg2(x: List Integer, i: Integer): Integer ==
\[
\begin{align*}
m & := x - 3 \\
de \deg(x, i) + \\
(x.(m+3) + \text{reduce}(\text{+}, [x.j \text{ for } j \text{ in } 2+i..m], 0)) \times \\
(x.(m+2)-x.(m+1))
\end{align*}
\]

\[
\text{checkResult}(\text{res: EXPRR, n: Symbol, l: Integer, list: List F,}
\text{ options: LGOPT): NNI ==}
\]

\[
\text{for } i \text{ in } l..1 \text{ by -1 repeat}
\]

\[
\text{den := eval(denominator res, n::EXPRR, (i-1)::EXPRR)}
\]

\[
\text{if den = 0 then return i::NNI}
\]

\[
\text{num := eval(numerator res, n::EXPRR, (i-1)::EXPRR)}
\]

\[
\text{if list.i ~= retract(retract(num/den)@R) then return i::NNI}
\]

\[
0$NNI
\]

\[
\text{SUPS2SUPF(p: SUP S): SUP F ==}
\]

\[
\text{if } F = S \text{ then}
\]

\[
p \text{ pretend SUP}(F)
\]

\[
\text{else if } F \text{ is Fraction } S \text{ then}
\]

\[
\text{map(coerce(#1)$Fraction(S), p)}
\]

\[
\text{SparseUnivariatePolynomialFunctions2(S, F)}
\]

\[
\text{else error "Type parameter } F \text{ should be either equal to } S \text{ or equal to Fraction } S"
\]

\[
\text{F2FPOLYS(p: F): FPOLYS ==}
\]

\[
\text{if } F = S \text{ then}
\]

\[
p::POLYF::FPOLYF \text{ pretend FPOLYS}
\]

\[
\text{else if } F \text{ is Fraction } S \text{ then}
\]

\[
n\text{umer(p)$Fraction(S)::POLYS/denom(p)$Fraction(S)::POLYS}
\]

\[
\text{else error "Type parameter } F \text{ should be either equal to } S \text{ or equal to Fraction } S"
\]

\[
\text{MPCSF ==> MPolyCatFunctions2(V, IndexedExponents V, IndexedExponents V, S, F, POLYS, POLYF)}
\]

\[
\text{SUPF2EXPRR(xx: Symbol, p: SUP F): EXPRR ==}
\]

\[
\text{zero? p => 0}
\]

\[
(\text{coerce(leadingCoefficient p)}::EXPRR \times (xx::EXPRR)**degree p + SUPF2EXPRR(xx, reductum p)
\]

\[
\text{FSUPF2EXPRR(xx: Symbol, p: FSUPF): EXPRR ==}
\]

\[
(\text{SUPF2EXPRR(xx, numer p) / (SUPF2EXPRR(xx, denom p))}
\]

\[
\text{POLYS2POLYF(p: POLYS): POLYF ==}
\]

\[
\text{if } F = S \text{ then}
\]

\[
p \text{ pretend POLYF}
\]

\[
\text{else if } F \text{ is Fraction } S \text{ then}
\]
map(coerce(#1)$Fraction(S), p)$MPCSF
else error "Type parameter F should be either equal to S or equal _
to Fraction S"

SUPPOLYS2SUPF(p: SUP POLYS, a1v: F, Av: F): SUP F ==
zero? p => 0
lc: POLYF := POLYS2POLYF leadingCoefficient(p)
monomial(retract(eval(lc, [index(1)$V, index(2)$V]::List V,
    [a1v, Av])),
    degree p) + SUPPOLYS2SUPF(reductum p, a1v, Av)

SUPFPOLYS2FSUPPOLYS(p: SUP FPOLYS): Fraction SUP POLYS ==
cden := splitDenominator(p)
    $UnivariatePolynomialCommonDenominator(POLYS, FPOLYS,SUP FPOLYS)

    pnum: SUP POLYS
    := map(retract(#1 * cden.den)$FPOLYS, p)
        $SparseUnivariatePolynomialFunctions2(FPOLYS, POLYS)
    pden: SUP POLYS := (cden.den)::SUP POLYS

    pnum/pden

POLYF2EXPRR(p: POLYF): EXPRR ==
    map(convert(#1)@Symbol::EXPRR, coerce(#1)@EXPRR, p)
        $PolynomialCategoryLifting(IndexedExponents V, V,
    F, POLYF, EXPRR)

    -- this needs documentation. In particular, why is V appearing here?
    GF ==> GeneralizedMultivariateFactorize(SingletonAsOrderedSet,
        IndexedExponents V, F, F, 
        SUP F)

    -- does not work:
    -- 6
    -- WARNING (genufact): No known algorithm to factor ?, trying square-free.

    -- GF ==> GenUFactorize F
    defaultD: DIFFSPECN
defaultD(expr: EXPRR): EXPRR == expr

    -- applies n->q^n or whatever DN is to i
    DN2DL: (DIFFSPECN, Integer) -> F
    DN2DL(DN, i) == retract(retract(DN(i::EXPRR))@R)

evalResultant(p1: POLYS, p2: POLYS, o: Integer, d: Integer, va1: V, vaA: V)_
    : List S ==
    res: List S := []
d1 := degree(p1, va1)
d2 := degree(p2, va1)
lead: S
for k in 1..d-o+1 repeat
  p1atk := univariate eval(p1, vA, k::S)
p2atk := univariate eval(p2, vA, k::S)

  d1atk := degree p1atk
d2atk := degree p2atk

  -- output("k: " string(k))$OutputPackage
  -- output("d1: " string(d1) " d1atk: " string(d1atk))$OutputPackage
  -- output("d2: " string(d2) " d2atk: " string(d2atk))$OutputPackage

  if d2atk < d2 then
    if d1atk < d1
      then lead := 0$S
      else lead := (leadingCoefficient p1atk)**((d2-d2atk)::NNI)
    else
      if d1atk < d1
        then lead := (-1$S)**d2 * (leadingCoefficient p2atk)**((d1-d1atk)::NNI)
        else lead := 1$S

  if zero? lead
    then res := cons(0, res)
  else res := cons(lead * (resultant(p1atk, p2atk)$SUP(S) exquo _
                      (k::S)**(o::NNI))::S,
                      res)

reverse res

vA::POLYS::FPOLYS + va1::POLYS::FPOLYS _
  * F2FPOLYS(DN2DL(basis, i) - DN2DL(basis, xm))

p2(xm: Integer, i: Symbol, a1v: F, Av: F, basis: DIFFSPECN): EXPRR ==
  coerce(Av) + coerce(a1v)*(basis(i::EXPRR) - basis(xm::EXPRR))

guessExpRatAux(xx: Symbol, list: List F, basis: DIFFSPECN,
xValues: List Integer, options: LGOPT): List EXPRR ==

  a1: V := index(1)$V
  A: V := index(2)$V

  len: NNI := #list
  if len < 4 then return []
    else len := (len-3)::NNI

  xlist := [F2FPOLYS DN2DL(basis, xValues.i) for i in 1..len]
x1 := F2FPOLYS DN2DL(basis, xValues.(len+1))
x2 := F2FPOLYS DN2DL(basis, xValues.(len+2))
x3 := F2FPOLYS DN2DL(basis, xValues.(len+3))

y: NNI -> FPOLYS :=
    F2FPOLYS(list.#1) * _
    p(last xValues, (xValues.#1)::Integer, a1, A, basis)**_
    (-(xValues.#1)::Integer)

ylist: List FPOLYS := [y i for i in 1..len]
y1 := y(len+1)
y2 := y(len+2)
y3 := y(len+3)

res := []::List EXPRR
-- tpd: this is undefined since maxDegree is always nonnegative
-- if maxDegree(options)$GOPT0 = -1
-- then maxDeg := len-1
-- else maxDeg := min(maxDegree(options)$GOPT0, len-1)
-- maxDeg := min(maxDegree(options)$GOPT0, len-1)
-- tpd:Integer := (maxDegree(options)$GOPT0)::NNI::Integer
-- maxDeg:Integer:=min(tpd,len-1)

for i in 0..maxDeg repeat
    if debug(options)$GOPT0 then
        output(hconcat("degree ExpRat ":OutputForm, i::OutputForm))
        $OutputPackage
    if debug(options)$GOPT0 then
        systemCommand("sys date +%s")$MoreSystemCommands
        output("interpolating...":OutputForm)$OutputPackage
    ri: FSUPFPOLYS
        := interpolate(xlist, ylist, (len-1-i)::NNI) _
        $FFFG(FPOLYS, SUP FPOLYS)
-- for experimental fraction free interpolation
--   ri: Fraction SUP POLYS
--      := interpolate(xlist, ylist, (len-1-i)::NNI) _
--      $FFFG(POLYS, SUP POLYS)

if debug(options)$GOPT0 then
    -- output(hconcat("xlist: ", xlist::OutputForm))$OutputPackage
    -- output(hconcat("ylist: ", ylist::OutputForm))$OutputPackage
    -- output(hconcat("ri: ", ri::OutputForm))$OutputPackage
    systemCommand("sys date +%s")$MoreSystemCommands
    output("polynomials...":OutputForm)$OutputPackage

poly1: POLYS := numer(elt(ri, x1)$SUP(FPOLYS) - y1)
poly2: POLYS := numer(elt(ri, x2)$SUP(FPOLYS) - y2)
poly3: POLYS := numer(elt(ri, x3)$SUP(FPOLYS) - y3)

-- for experimental fraction free interpolation
-- ri2: FSUPFPOLYS := map(#1::FPOLYS, numer ri) _
-- $SparseUnivariatePolynomialFunctions2(POLYS, FPOLYS)_
-- /map(#1::FPOLYS, denom ri) _
-- $SparseUnivariatePolynomialFunctions2(POLYS, FPOLYS)
--
-- poly1: POLYS := numer(elt(ri2, x1)$SUP(FPOLYS) - y1)
poly2: POLYS := numer(elt(ri2, x2)$SUP(FPOLYS) - y2)
poly3: POLYS := numer(elt(ri2, x3)$SUP(FPOLYS) - y3)

n:Integer := len - i
o1: Integer := ord1(xValues, i)
d1: Integer := deg1(xValues, i)
o2: Integer := ord2(xValues, i)
d2: Integer := deg2(xValues, i)

-- another compiler bug: using i as iterator here makes the loop break

if debug(options)$GOPT0 then
  systemCommand("sys date +%s")$MoreSystemCommands
  output("interpolating resultants..."::OutputForm)$OutputPackage
  res1: SUP S := newton(evalResultant(poly1, poly3, o1, d1, a1, A))
    $NewtonInterpolation(S)
  res2: SUP S := newton(evalResultant(poly2, poly3, o2, d2, a1, A))
    $NewtonInterpolation(S)

if debug(options)$GOPT0 then
  -- res1: SUP S := univariate(resultant(poly1, poly3, a1))
  -- res2: SUP S := univariate(resultant(poly2, poly3, a1))
  -- if res1 ~= res1res or res2 ~= res2res then
  --   output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
  --   output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
  --   output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage
  --   output(hconcat("res1 ", res1::OutputForm))$OutputPackage
  --   output(hconcat("res2 ", res2::OutputForm))$OutputPackage
  --   output("n/i: " string(n) " " string(i))$OutputPackage
  --   output("res1 ord: " string(o1) " " string(minimumDegree res1))
    $OutputPackage
  --   output("res1 deg: " string(d1) " " string(degree res1))
    $OutputPackage
  --   output("res2 ord: " string(o2) " " string(minimumDegree res2))
    $OutputPackage
  --   output("res2 deg: " string(d2) " " string(degree res2))
    $OutputPackage
if debug(options)$GOPT0 then
    systemCommand("sys date +%s")$MoreSystemCommands
    output("computing gcd..."::OutputForm)$OutputPackage

-- we want to solve over F
-- for polynomial domains S this seems to be very costly!
res3: SUP F := SUPS2SUPF(primitivePart(gcd(res1, res2)))

if debug(options)$GOPT0 then
    systemCommand("sys date +%s")$MoreSystemCommands
    output("solving..."::OutputForm)$OutputPackage

-- res3 is a polynomial in A=a0+(len+3)*a1
-- now we have to find the roots of res3

for f in factors factor(res3)$GF | degree f.factor = 1 repeat
    -- we are only interested in the linear factors
    if debug(options)$GOPT0 then
        output(hconcat("f: ", f::OutputForm))$OutputPackage
    Av: F := -coefficient(f.factor, 0)
    / leadingCoefficient f.factor

-- FIXME: in an earlier version, we disregarded vanishing Av
-- maybe we intended to disregard vanishing a1v? Either doesn't really
-- make sense to me right now.

    evalPoly := eval(POLYS2POLYF poly3, A, Av)
    if zero? evalPoly
    then evalPoly := eval(POLYS2POLYF poly1, A, Av)

-- Note that it really may happen that poly3 vanishes when specializing
-- A. Consider for example guessExpRat([1,1,1,1]).

-- FIXME: We check poly1 below, too. I should work out in what cases poly3
-- vanishes.

    for g in factors factor(univariate evalPoly)$GF | degree g.factor = 1 repeat
        if debug(options)$GOPT0 then
            output(hconcat("g: ", g::OutputForm))$OutputPackage
        alv: F := -coefficient(g.factor, 0)
        / leadingCoefficient g.factor

-- check whether poly1 and poly2 really vanish. Note that we could have found
-- an extraneous solution, since we only computed the gcd of the two
-- resultants.
\[ t_1 := \text{eval}(\text{POLYS2POLYF} \ poly1, [a_1, A]::\text{List} \ V, \ [a_1v, Av]::\text{List} \ F) \]

if zero? t1 then
\[ t_2 := \text{eval}(\text{POLYS2POLYF} \ poly2, [a_1, A]::\text{List} \ V, \ [a_1v, Av]::\text{List} \ F) \]

if zero? t2 then

\[ \text{ri1: Fraction SUP POLYS := \text{SUPFPOLYS2FSUPPOLYS}(\text{numer ri})} \]
\[ \quad / \text{SUPFPOLYS2FSUPPOLYS}(\text{denom ri}) \]

-- for experimental fraction free interpolation

\[ \text{rii: Fraction SUP POLYS := ri} \]

\[ \text{numr: SUP F := \text{SUPPOLYS2SUPF}(\text{numer ri1}, a_1v, Av)} \]
\[ \text{denr: SUP F := \text{SUPPOLYS2SUPF}(\text{denom ri1}, a_1v, Av)} \]

if not zero? denr then
\[ \text{res4: EXPRR := \text{eval}(\text{FSUPF2EXPRR}(\text{xx, numr/denr}),} \]
\[ \quad \text{kernel(\text{xx}),} \]
\[ \quad \text{basis(\text{xx::EXPRR})}} \]
\[ \quad * \text{p2(last xValues, \text{xx, alv, Av, basis})} \]
\[ \quad **\text{xx::EXPRR} \]
\[ \text{res := cons(res4, res)} \]
else if zero? numr and debug(\text{options})$GOPT0 then
\[ \quad \text{output("numerator and denominator vanish!") \}$OutputPackage \]

-- If we are only interested in one solution, we do not try other degrees if we
-- have found already some solutions. I.e., the indentation here is correct.

if not null(res) and one(\text{options})$GOPT0 then return res

\[ \text{res} \]

\[ \text{guessExpRatAux0(list: List F, basis: DIFFSPECN, options: LGOPT)} \]
\[ \quad \text{:GUESSRESULT} \]
if zero? \text{safety(\text{options})$GOPT0 then
\[ \quad \text{error "Guess: guessExpRat does not support zero safety"} \]

-- guesses Functions of the Form \((a_1*n+a_0)^n*rat(n)\)
\[ \text{xx := indexName(\text{options})$GOPT0} \]

-- restrict to safety

\[ \text{len: Integer := \#list} \]
if len\text{-safety(\text{options})$GOPT0+1 < 0 then return []

\[ \text{shortlist: List F := first(list, (len\text{-safety(\text{options})$GOPT0+1)::NNI)} \]

-- remove zeros from list
zeros: EPRR := 1
newlist: List F
xValues: List Integer

i: Integer := -1
for x in shortlist repeat
  i := i+1
  if x = 0 then
    zeros := zeros * (basis(xx::EXPRR) - basis(i::EXPRR))

i := -1
for x in shortlist repeat
  i := i+1
  if x ~= 0 then
    newlist := cons(x/retract(retract(eval(zeros, xx::EXPRR, i::EXPRR))@R), newlist)
    xValues := cons(i, xValues)

newlist := reverse newlist
xValues := reverse xValues

res: List EPRR := [eval(zeros * f, xx::EXPRR, xx::EXPRR) _
  for f in guessExpRatAux(xx, newlist, basis, xValues, options)]

reslist := map([#1, checkResult(#1, xx, len, list, options)], res)
$ListFunctions2(EXPRR, Record(function: EPRR, order: NNI))

select(#1.order < len-safety(options)$GOPT0, reslist)

guessExpRat(list : List F): GUESSRESULT ==
guessExpRatAux0(list, defaultD, [])
guessExpRat(list: List F, options: LGOPT): GUESSRESULT ==
guessExpRatAux0(list, defaultD, options)

if F has RetractableTo Symbol and S has RetractableTo Symbol then
  guessExpRat(q: Symbol): GUESSER ==
guessExpRatAux0(#1, q::EXPRR**#1, #2)

EXT ==> (Integer, V, V) -> FPOLYS
EXTEXPR ==> (Symbol, F, F) -> EPRR

binExt: EXT
binExt(i: Integer, val: V, vA: V): FPOLYS ==
  numl: List POLYS := [(vA::POLYS) + i * (val::POLYS) - (l::POLYS) _
for \( l \) in 0..i-1

\[
\text{num: POLYS := reduce(_*, numl, 1)}
\]

\[
\text{num/(factorial(i)::POLYS)}
\]

\[
\text{binExtEXPR: EXTEXPR}
\]

\[
\text{binExtEXPR}(i: Symbol, a1v: F, Av: F): EXPRR ==}
\]

\[
\text{binomial(coerce Av + coerce a1v * (i::EXPRR), i::EXPRR)}
\]

\[
\text{guessBinRatAux(xx: Symbol, list: List F,}
\]

\[
\text{basis: DIFFSPECN, ext: EXT, extEXPR: EXTEXPR,}
\]

\[
\text{xValues: List Integer, options: LGOPT): List EXPRR ==}
\]

\[
\text{a1: V := index(1)$V}
\]

\[
\text{A: V := index(2)$V}
\]

\[
\text{len: NNI := #list}
\]

\[
\text{if len < 4 then return []}
\]

\[
\text{else len := (len-3)::NNI}
\]

\[
\text{xlist := [F2FPOLYS DN2DL(basis, xValues.i) for i in 1..len]}
\]

\[
x1 := F2FPOLYS DN2DL(basis, xValues.(len+1))
\]

\[
x2 := F2FPOLYS DN2DL(basis, xValues.(len+2))
\]

\[
x3 := F2FPOLYS DN2DL(basis, xValues.(len+3))
\]

\[
y: NNI -> FPOLYS :
\]

\[
\text{F2FPOLYS(list.#1) / _}
\]

\[
\text{ext((xValues.#1)::Integer, a1, A)}
\]

\[
ylist: List FPOLYS := [y i for i in 1..len]
\]

\[
y1 := y(len+1)
\]

\[
y2 := y(len+2)
\]

\[
y3 := y(len+3)
\]

\[
\text{res := []::List EXPRR}
\]

-- tpd: this is undefined since maxDegree is always nonnegative

-- if maxDegree(options)$GOPTO = -1

-- then maxDeg := len-1

-- else maxDeg := min(maxDegree(options)$GOPTO, len-1)

-- maxDeg := min(maxDegree(options)$GOPTO, len-1)

-- tpd:Integer := (maxDegree(options)$GOPTO)::NNI::Integer

-- maxDeg:Integer := (maxDegree(options)$GOPTO)::NNI::Integer

for \( i \) in 0..maxDeg repeat

-- if debug(options)$GOPTO then

-- output(hconcat("degree BinRat "::OutputForm, i::OutputForm))

-- $OutputPackage
ri: FSUPFPOLYS := interpolate(xlist, ylist, (len-1-i)::NNI) $FFFG(FPOLYS, SUP FPOLYS)

poly1: POLYS := numer(elt(ri, x1)$SUP(FPOLYS) - y1)
poly2: POLYS := numer(elt(ri, x2)$SUP(FPOLYS) - y2)
poly3: POLYS := numer(elt(ri, x3)$SUP(FPOLYS) - y3)

n:Integer := len - i
res1: SUP S := univariate(resultant(poly1, poly3, a1))
res2: SUP S := univariate(resultant(poly2, poly3, a1))
if debug(options)$GOPT0 then
  output(hconcat("res1 ", res1::OutputForm))$OutputPackage
  output(hconcat("res2 ", res2::OutputForm))$OutputPackage

if res1 ~= res1res or res2 ~= res2res then
  output(hconcat("poly1 ", poly1::OutputForm))$OutputPackage
  output(hconcat("poly2 ", poly2::OutputForm))$OutputPackage
  output(hconcat("poly3 ", poly3::OutputForm))$OutputPackage
  output(hconcat("res1 ", res1::OutputForm))$OutputPackage
  output(hconcat("res2 ", res2::OutputForm))$OutputPackage
  output(hconcat("n/i: " string(n) " " string(i))$OutputPackage
  output("res1 ord: " string(minimumDegree res1))$OutputPackage
  output("res1 deg: " string(degree res1))$OutputPackage
  output("res2 ord: " string(minimumDegree res2))$OutputPackage
  output("res2 deg: " string(degree res2))$OutputPackage

if debug(options)$GOPT0 then
  output("computing gcd...")$OutputPackage

-- we want to solve over F
res3: SUP F := SUPS2SUPF(primitivePart(gcd(res1, res2)))
-- if debug(options)$GOPT0 then
-- output(hconcat("res3 ", res3::OutputForm))$OutputPackage

-- res3 is a polynomial in A=a0+(len+3)*a1
-- now we have to find the roots of res3

for f in factors factor(res3)$GF | degree f.factor = 1 repeat
-- we are only interested in the linear factors
-- if debug(options)$GOPT0 then
-- output(hconcat("f: ", f::OutputForm))$OutputPackage

Av: F := -coefficient(f.factor, 0)
   / leadingCoefficient f.factor
-- if debug(options)$GOPT0 then
-- output(hconcat("Av: ", Av::OutputForm))$OutputPackage

-- FIXME: in an earlier version, we disregarded vanishing Av
-- maybe we intended to disregard vanishing a1v? Either doesn't really
-- make sense to me right now.

evalPoly := eval(POLYS2POLYF poly3, A, Av)
if zero? evalPoly
then evalPoly := eval(POLYS2POLYF poly1, A, Av)
-- Note that it really may happen that poly3 vanishes when specializing
-- A. Consider for example guessExpRat([[1,1,1,1]]).

-- FIXME: We check poly1 below, too. I should work out in what cases poly3
-- vanishes.

for g in factors factor(univariate evalPoly)$GF
   | degree g.factor = 1 repeat
-- if debug(options)$GOPT0 then
-- output(hconcat("g: ", g::OutputForm))$OutputPackage

a1v: F := -coefficient(g.factor, 0)
   / leadingCoefficient g.factor
-- if debug(options)$GOPT0 then
-- output(hconcat("a1v: ", a1v::OutputForm))$OutputPackage

-- check whether poly1 and poly2 really vanish. Note that we could have found
-- an extraneous solution, since we only computed the gcd of the two
-- resultants.

t1 := eval(POLYS2POLYF poly1, [a1, A]::List V, [a1v, Av]::List F)
-- if debug(options)$GOPT0 then
-- output(hconcat("t1: ", t1::OutputForm))$OutputPackage
if zero? t1 then
    t2 := eval(POLYS2POLYF poly2, [a1, A]::List V, [alv, Av]::List F)

-- if debug(options)$GOPT0 then
-- output(hconcat("t2: ", t2::OutputForm))$OutputPackage

if zero? t2 then

ri1: Fraction SUP POLYS := SUPFPOLYS2FSUPPOLYS(numer ri) / SUPFPOLYS2FSUPPOLYS(denom ri)

-- if debug(options)$GOPT0 then
-- output(hconcat("ri1: ", ri1::OutputForm))$OutputPackage

numr: SUP F := SUPPOLYS2SUPF(numer ri1, alv, Av)
denr: SUP F := SUPPOLYS2SUPF(denom ri1, alv, Av)

-- if debug(options)$GOPT0 then
-- output(hconcat("numr: ", numr::OutputForm))$OutputPackage
-- output(hconcat("denr: ", denr::OutputForm))$OutputPackage

if not zero? denr then
    res4: EXPRR := eval(FSUPF2EXPRR(xx, numr/denr),
                        kernel(xx),
                        basis(xx::EXPRR)) * extEXPR(xx, alv, Av)

-- if debug(options)$GOPT0 then
-- output(hconcat("res4: ", res4::OutputForm))$OutputPackage

res := cons(res4, res)
else if zero? numr and debug(options)$GOPT0 then
    output("numerator and denominator vanish!")$OutputPackage

-- If we are only interested in one solution, we do not try other degrees if we
-- have found already some solutions. I.e., the indentation here is correct.

if not null(res) and one(options)$GOPT0 then return res

res

guessBinRatAux0(list: List F,
    basis: DIFFSPECN, ext: EXT, extEXPR: EXTEXPR,
    options: LGOPT): GUESSRESULT ==

if zero? safety(options)$GOPT0 then
error "Guess: guessBinRat does not support zero safety"
-- guesses Functions of the form binomial(a+b*n, n)*rat(n)
xx := indexName(options)$GOPT0

-- restrict to safety

len: Integer := #list
if len-safety(options)$GOPT0+1 < 0 then return []

shortlist: List F := first(list, (len-safety(options)$GOPT0+1)::NNI)

-- remove zeros from list

zeros: EXPRR := 1
newlist: List F
xValues: List Integer

i: Integer := -1
for x in shortlist repeat
  i := i+1
  if x = 0 then
    zeros := zeros * (basis(xx::EXPRR) - basis(i::EXPRR))
  end if
  j := i
for x in shortlist repeat
  j := j+1
  if x ~= 0 then
    newlist := cons(x/retract(retract(eval(zeros, xx::EXPRR, i::EXPRR))@R),
                     newlist)
    xValues := cons(i, xValues)
  end if
end for
newlist := reverse newlist
xValues := reverse xValues

res: List EXPRR
:= [eval(zeros * f, xx::EXPRR, xx::EXPRR) _
         for f in guessBinRatAux(xx, newlist, basis, ext, extEXPR, xValues, _
                                  options)]
reslist := map([#1, checkResult(#1, xx, len, list, options)], res)
  $ListFunctions2(EXPRR, Record(function: EXPRR, order: NNI))

select(#1.order < len-safety(options)$GOPT0, reslist)
guessBinRat(list : List F): GUESSRESULT ==
guessBinRatAux0(list, defaultD, binExt, binExtEXPR, [])
guessBinRat(list: List F, options: LGOPT): GUESSRESULT ==
guessBinRatAux0(list, defaultD, binExt, binExtEXPR, options)
if F has RetractableTo Symbol and S has RetractableTo Symbol then

\( qD : \text{Symbol} \to \text{DIFFSPECN} \)
\( qD \ q == (q:\text{EXPRR})^{#1} \)

\[ q\text{BinExtAux}(q: \text{Symbol}, i: \text{Integer}, va1: V, vA: V): \text{FPOLYS} == \]
\[ f1: \text{List FPOLYS} \]
\[ := [(1\text{FPOLYS} - va1::\text{POLYS}::\text{FPOLYS} \ast (vA::\text{POLYS}::\text{FPOLYS})^{#(i-1)} \ast \]
\[ \text{F2FPOLYS}(q::F)^{#1}) / (1\text{-F2FPOLYS}(q::F)^{#1}) \]
\[ \text{for } l \text{ in } 1..i] \]
\[ \text{reduce}(\_\ast, f1, 1) \]

\( q\text{BinExt}: \text{Symbol} \to \text{EXT} \)
\( q\text{BinExt} \ q == q\text{BinExtAux}(q, #1, #2, #3) \)

\( q\text{BinExtEXPRAux}(q: \text{Symbol}, i: \text{Symbol}, a1v: F, Av: F): \text{EXPRR} == \]
\[ l: \text{Symbol} := 'l \]
\[ \text{product}((1\text{EXPRR} - \]
\[ \text{coerce } a1v \ast (\text{coerce } Av) \ast (\text{coerce } i - 1\text{EXPRR}) \ast \]
\[ (q::\text{EXPRR}) \ast \text{coerce}(l)) / \]
\[ (1\text{EXPRR} - (q::\text{EXPRR}) \ast \text{coerce}(l)), \]
\[ \text{equation}(l, 1\text{EXPRR}..i::\text{EXPRR})) \]

\( q\text{BinExtEXPR}: \text{Symbol} \to \text{TEXPR} \)
\( q\text{BinExtEXPR} \ q == q\text{BinExtEXPRAux}(q, #1, #2, #3) \)

\( \text{guessBinRat}(q: \text{Symbol}): \text{GUESSER} == \]
\[ \text{guessBinRatAux0}(#1, qD \ q, \text{qBinExt} \ q, \text{qBinExtEXPR} \ q, #2) \]

-- some useful types for Ore operators that work on series

-- the differentiation operator
\( \text{DIFFSPECX} \Rightarrow (\text{EXPRR}, \text{Symbol}, \text{NonNegativeInteger}) \to \text{EXPRR} \)
\[ \text{eg.}: f(x)\Rightarrow f(q\ast x) \]
\[ \text{eg.}: f(x)\Rightarrow D(f, x) \]

\( \text{DIFFSPECS} \Rightarrow (\text{UFPSS}, \text{NonNegativeInteger}) \to \text{UFPSS} \)
\[ \text{eg.}: f(x)\Rightarrow f(q\ast x) \]

\( \text{DIFFSPECSF} \Rightarrow (\text{UFPSSUPF}, \text{NonNegativeInteger}) \to \text{UFPSSUPF} \)
\[ \text{eg.}: f(x)\Rightarrow f(q\ast x) \]

-- the constant term for the inhomogeneous case
\( \text{DIFFSPEC1} \Rightarrow \text{UFPSS} \)
\( \text{DIFFSPECIF} \Rightarrow \text{UFPSSUPF} \)
DIFFSPEC1X ==> Symbol -> EXPRR

termAsEXPRR(f: EXPRR, xx: Symbol, l: List Integer, 
    DX: DIFFSPECX, D1X: DIFFSPEC1X): EXPRR ==
    if empty? l then D1X(xx)
    else
        ll: List List Integer := powers(l)$Partition
        fl: List EXPRR := [DX(f, xx, (first part-1)::NonNegativeInteger) ** second(part)::NNI for part in ll]
        reduce(_, fl)

termAsUFPSF(f: UFPSF, l: List Integer, DS: DIFFSPECS, D1: DIFFSPEC1): UFPSF ==
    if empty? l then D1
    else
        ll: List List Integer := powers(l)$Partition
        -- first of each element of ll is the derivative, second is the power
        fl: List UFPSF := [DS(f, (first part -1)::NonNegativeInteger) ** second(part)::NNI for part in ll]
        reduce(_, fl)

        -- returns \prod f^(l_i), but using the Hadamard product
        termAsUFPSF2(f: UFPSF, l: List Integer, DS: DIFFSPECS, D1: DIFFSPEC1): UFPSF ==
            if empty? l then D1
            else
                ll: List List Integer := powers(l)$Partition
                -- first of each element of ll is the derivative, second is the power
                fl: List UFPSF
                    := [map(#1** second(part)::NNI, DS(f, (first part -1)::NNI)) for part in ll]
                reduce(hadamard$UFPS1(F), fl)

        termAsUFPSSUPF(f: UFPSSUPF, l: List Integer, 
            DSF: DIFFSPECSF, D1F: DIFFSPEC1F): UFPSSUPF ==
            if empty? l then D1F
            else
                ll: List List Integer := powers(l)$Partition
                -- first of each element of ll is the derivative, second is the power
                fl: List UFPSSUPF
package guess

1147

\[DSF(f, (\text{first part} - 1)::\text{NonNegativeInteger})\]

**second(part)::\text{NNI for part in ll}\]

reduce(_*, f1)

-- returns \(\prod f^{(l,i)}\), but using the Hadamard product

termAsUFPSUPF2(f: UFPSUPF, l: List Integer,  
DSF: DIFFSPECSF, D1F: DIFFSPEC1F): UFPSUPF ==

if empty? 1 then D1F

else

ll: List List Integer := powers(l)$Partition

-- first of each element of ll is the derivative, second is the power

f1: List UFPSUPF

:= [map(#1 ** second(part)::\text{NNI}, DSF(f, (\text{first part} - 1)::\text{NNI})) _

for part in ll]

reduce(hadamard$UFPS1(SUP F), f1)

FilteredPartitionStream(options: LGOPT): Stream List Integer ==

-- \text{tpd}: must force types to \text{NNI} and \text{Integer}

maxD: Integer := 1 + maxDerivative(options)$GOPT0::\text{NNI}::\text{Integer}

maxP: Integer := maxPower(options)$GOPT0::\text{NNI}::\text{Integer}

if maxD > 0 and maxP > -1 then

\(s := \text{partitions(maxD, maxP)$PartitionsAndPermutations}\)

else

\(s1: \text{Stream Integer} := \text{generate(inc, 1)$Stream(Integer)}\)

\(s2: \text{Stream Stream List Integer} := \text{map(partitions(#1)$PartitionsAndPermutations, } s1)\)

\($\text{StreamFunctions2(Integer, Stream List Integer)}\)

\(s3: \text{Stream List Integer} := \text{concat(s2)$StreamFunctions1(List Integer)}\)

-- \(s := \text{cons([], select(((maxD = 0) or (first #1 <= maxD)) _

- and ((maxP = -1) or (# #1 <= maxP)), s3))}\)

\(s := \text{cons([], select(((maxD = 0) or (# #1 <= maxD)) _

- and ((maxP = -1) or (first #1 <= maxP)), s3))}\)

\(s := \text{conjugates(s)$PartitionsAndPermutations}\)

-- \text{tpd}: force the Boolean branch

\(\text{tpd2:Boolean} := \text{homogeneous(options)$GOPT0::Boolean}\)

if tpd2 then rest s else s

-- for functions
ADEguessStream(f: UFPSF, partitions: Stream List Integer,
DS: DIFFSPECS, D1: DIFFSPEC1): Stream UFPSF ==
map(termAsUFPSF(f, #1, DS, D1), partitions)
$StreamFunctions2(List Integer, UFPSF)

-- for coefficients, i.e., using the Hadamard product
ADEguessStream2(f: UFPSF, partitions: Stream List Integer,
DS: DIFFSPECS, D1: DIFFSPEC1): Stream UFPSF ==
map(termAsUFPSF2(f, #1, DS, D1), partitions)
$StreamFunctions2(List Integer, UFPSF)

ADEdegreeStream(partitions: Stream List Integer): Stream NNI ==
scan(0, max((if empty? #1 then 0 else (first #1 - 1)::NNI), #2),
partitions)$StreamFunctions2(List Integer, NNI)

ADEtestStream(f: UFPSSUPF, partitions: Stream List Integer,
DSF: DIFFSPECSF, D1F: DIFFSPEC1F): Stream UFPSSUPF ==
map(termAsUFPSSUPF(f, #1, DSF, D1F), partitions)
$StreamFunctions2(List Integer, UFPSSUPF)

ADEtestStream2(f: UFPSSUPF, partitions: Stream List Integer,
DSF: DIFFSPECSF, D1F: DIFFSPEC1F): Stream UFPSSUPF ==
map(termAsUFPSSUPF2(f, #1, DSF, D1F), partitions)
$StreamFunctions2(List Integer, UFPSSUPF)

ADEEXPRRStream(f: EXPRR, xx: Symbol, partitions: Stream List Integer,
DX: DIFFSPECX, D1X: DIFFSPEC1X): Stream EXPRR ==
map(termAsEXPRR(f, xx, #1, DX, D1X), partitions)
$StreamFunctions2(List Integer, EXPRR)

diffDX: DIFFSPECX
diffDX(expr, x, n) == D(expr, x, n)

diffDS: DIFFSPECS
diffDS(s, n) == D(s, n)

diffDSF: DIFFSPECSF
diffDSF(s, n) ==
-- I have to help the compiler here a little to choose the right signature...
  if SUP F has _*: (NonNegativeInteger, SUP F) -> SUP F
  then D(s, n)

diffAX: DIFFSPECAX
diffAX(l: NNI, x: Symbol, f: EXPRR): EXPRR ==
  (x::EXPRR)**l * f

diffA: DIFFSPECA
diffA(k: NNI, l: NNI, f: SUP S): S ==
  DiffAction(k, l, f)$FFFG(S, SUP S)
diffAF: DIFFSPECAF
  diffAF(k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
  DiffAction(k, l, f)$FFFG(SUP F, UFPSSUPF)

diffC: DIFFSPEC
  diffC(total: NNI): List S == DiffC(total)$FFFG(S, SUP S)

diff1X: DIFFSPEC1X
  diff1X(x: Symbol) == 1$EXPRR

diffHP options ==
  if displayAsGF(options)$GOPT0 then
    partitions := FilteredPartitionStream options
    [ADEguessStream(#1, partitions, diffDS, 1$UFPSF), _
      ADEdegreeStream partitions, _
      ADEtestStream(#1, partitions, diffDSF, 1$UFPSSUPF), _
      ADEEXPRRStream(#1, #2, partitions, diffDX, diff1X), _
      diffA, diffAF, diffAX, diffC]$HPSPEC
  else
    error "Guess: guessADE supports only displayAsGF"

  if F has RetractableTo Symbol and S has RetractableTo Symbol then

  qDiffDX(q: Symbol, expr: EXPRR, x: Symbol, n: NonNegativeInteger): EXPRR ==
  eval(expr, x::EXPRR, (q::EXPRR)**n*x::EXPRR)

  qDiffDS(q: Symbol, s: UFPSF, n: NonNegativeInteger): UFPSF ==
  multiplyCoefficients((q::F)**((n*#1)::NonNegativeInteger), s)

  qDiffDSF(q: Symbol, s: UFPSSUPF, n: NonNegativeInteger): UFPSSUPF ==
  multiplyCoefficients((q::F::SUP F)**((n*#1)::NonNegativeInteger), s)

  diffHP(q: Symbol): (LGOPT -> HPSPEC) ==
  if displayAsGF(#1)$GOPT0 then
    partitions := FilteredPartitionStream #1
    [ADEguessStream(#1, partitions, qDiffDS(q, #1, #2), 1$UFPSF), _
      repeating([0$NNI]$Stream(NNI), _
      ADEtestStream(#1, partitions, qDiffDSF(q, #1, #2), 1$UFPSSUPF), _
      ADEEXPRRStream(#1, #2, partitions, qDiffDX(q, #1, #2, #3), diff1X), _
      diffA, diffAF, diffAX, diffC]$HPSPEC
  else
    error "Guess: guessADE supports only displayAsGF"

  ShiftSX(expr: EXPRR, x: Symbol, n: NNI): EXPRR ==
  eval(expr, x::EXPRR, x::EXPRR+n::EXPRR)

  ShiftSXGF(expr: EXPRR, x: Symbol, n: NNI): EXPRR ==
  if zero? n then expr
  else
    l := [eval(D(expr, x, i)/factorial(i)::EXPRR, x::EXPRR, 0$EXPRR)
\[(x::\text{EXPRR})^i \text{ for } i \in 0..n-1\]
\[
\frac{\text{expr-reduce}(+, l)}{(x::\text{EXPRR})^n}
\]

\[\text{ShiftSS}(s:\text{UFPSF}, n:\text{NNI}): \text{UFPSF} ==
\]
\[
((\text{quoByVar} \ #1)^n)\text{MappingPackage1(UFPSF)} \ (s)
\]

\[\text{ShiftSF}(s:\text{UFPSUFF}, n:\text{NNI}): \text{UFPSUFF} ==
\]
\[
((\text{quoByVar} \ #1)^n)\text{MappingPackage1(UFPSUFF)} \ (s)
\]

\[\text{ShiftAX}(l: \text{NNI}, n: \text{Symbol}, f: \text{EXPRR}): \text{EXPRR} ==
\]
\[
n::\text{EXPRR}^l \ast f
\]

\[\text{ShiftAXGF}(l: \text{NNI}, x: \text{Symbol}, f: \text{EXPRR}): \text{EXPRR} ==
\]
\[
\text{-- I need to help the compiler here, unfortunately}
\]
\[
\text{if zero?} \ 1 \text{ then } f
\]
\[
\text{else}
\]
\[
s := \left[(\text{stirling2}(l, i))\text{IntegerCombinatoricFunctions(Integer)}::\text{EXPRR} \ast (x::\text{EXPRR})^i\text{D}(f, x, i) \text{ for } i \in 1..l\right]
\]
\[
\text{reduce}(+, s)
\]

\[\text{ShiftA}(k: \text{NNI}, l: \text{NNI}, f: \text{SUP S}): S ==
\]
\[
\text{ShiftAction}(k, l, f)\text{FFFG}(S, \text{SUP S})
\]

\[\text{ShiftAF}(k: \text{NNI}, l: \text{NNI}, f: \text{UFPSSUPF}): \text{SUP F} ==
\]
\[
\text{ShiftAction}(k, l, f)\text{FFFG}(\text{SUP F}, \text{UFPSSUPF})
\]

\[\text{ShiftC}(\text{total}: \text{NNI}): \text{List} S ==
\]
\[
\text{ShiftC(\text{total})}\text{FFFG}(S, \text{SUP S})
\]

\[\text{shiftHP} \text{ options} ==
\]
\[
\text{partitions} := \text{FilteredPartitionStream} \text{ options}
\]
\[
\text{if displayAsGF(\text{options})} \text{GOPT0} \text{ then}
\]
\[
\text{if maxPower(\text{options})} \text{GOPT0} = 1 \text{ then}
\]
\[
[\text{ADEguessStream}(\#1, \text{partitions}, \text{ShiftSS}, (1-\text{monomial}(1,1))^{*(-1)}), _
\]
\[
\text{ADEdegreeStream} \text{ partitions, _}
\]
\[
\text{ADEtestStream}(\#1, \text{partitions}, \text{ShiftSF}, (1-\text{monomial}(1,1))^{*(-1)}), _
\]
\[
\text{ADEXPRRStream}(\#1, \#2, \text{partitions}, \text{ShiftSXGF}, 1/(1-\#1::\text{EXPRR})), _
\]
\[
\text{ShiftA, ShiftAF, ShiftAXGF, ShiftC}\text{HPSPEC}
\]
\[
\text{else}
\]
\[
\text{error} \text{ "Guess: no support for the Shift operator with displayAsGF _}
\]
\[
\text{and maxPower>1"}
\]
\[
\text{else}
\]
\[
[\text{ADEguessStream2}(\#1, \text{partitions}, \text{ShiftSS}, (1-\text{monomial}(1,1))^{*(-1)}), _
\]
\[
\text{ADEdegreeStream} \text{ partitions, _}
\]
\[
\text{ADEtestStream2}(\#1, \text{partitions}, \text{ShiftSF}, (1-\text{monomial}(1,1))^{*(-1)}), _
\]
\[
\text{ADEXPRRStream}(\#1, \#2, \text{partitions}, \text{ShiftSX, diff1X}), _
\]
\[
\text{ShiftA, ShiftAF, ShiftAX, ShiftC}\text{HPSPEC}
\]

\[\text{if F has RetractableTo Symbol and S has RetractableTo Symbol then}
qShiftAX(q: Symbol, l: NNI, n: Symbol, f: EXPRR): EXPRR ==
(q::EXPRR)**(l*n::EXPRR) * f

qShiftA(q: Symbol, k: NNI, l: NNI, f: SUP S): S ==
qShiftAction(q::S, k, l, f)$FFFG(S, SUP S)

qShiftAF(q: Symbol, k: NNI, l: NNI, f: UFPSSUPF): SUP F ==
qShiftAction(q::F::SUP(F), k, l, f)$FFFG(SUP F, UFPSSUPF)

qShiftC(q: Symbol, total: NNI): List S ==
qShiftC(q::S, total)$FFFG(S, SUP S)

shiftHP(q: Symbol): (LGQPT -> HPSPEC) ==
partitions := FilteredPartitionStream #1
if displayAsGF(#1)$GOPT0 then
  error "Guess: no support for the qShift operator with displayAsGF"
else
  [ADEguessStream2(#1, partitions, ShiftSS, _
   (1-monomial(1,1))**(-1)), _
   ADEdegreeStream partitions, _
   ADEtestStream2(#1, partitions, ShiftSF, _
   (1-monomial(1,1))**(-1)), _
   ADEEXPRRStream(#1, #2, partitions, ShiftSX, diff1X), _
   qShiftA(q, #1, #2, #3), qShiftAF(q, #1, #2, #3), _
   qShiftAX(q, #1, #2, #3), qShiftC(q, #1)]$HPSPEC

makeEXPRR(DAX: DIFFSPECAX, x: Symbol, p: SUP F, expr: EXPRR): EXPRR ==
if zero? p then 0$EXPRR
else
  coerce(leadingCoefficient p)::EXPRR * DAX(degree p, x, expr) +
  makeEXPRR(DAX, x, reductum p, expr)

list2UFPSS(list: List F): UFPSF ==
  series(list::Stream F)$UFPSS

list2UFPSSUPF(list: List F): AFPSSUPF ==
  l := [e::SUP(F) for e in list for i in 0..]:Stream SUP F
  series(l)$UFPSSUPF + monomial(monomial(1,1)$SUP(F), #list)$UFPSSUPF

SUPF2SUPSUPF(p: SUP F): SUP SUP F ==
  map(#1::SUP F, p)$SparseUnivariatePolynomialFunctions2(F, SUP F)

UFPSSUPF(f: UFPSF, deg: NNI): SUP F ==
  makeSUP univariatePolynomial(f, deg)

gbListSUPF(s: Stream UFPSF, o: NNI, deg: NNI): List SUP F ==
  map(UFPSSUPF(#1, deg), entries complete first(s, o))
  $ListFunctions2(UFPSF, SUP F)

S2EXPRR(s: S): EXPRR ==
  if F is S then
coerce(s pretend F)@EXPRR
else if F is Fraction S then
  coerce(s::Fraction(S))@EXPRR
else error "Type parameter F should be either equal to S or equal to Fraction S"

guessInterpolate(guessList: List SUP F, eta: List NNI, D: HPSPEC)
  : Matrix SUP S ==
if F is S then
  vguessList: Vector SUP S := vector(guessList pretend List(SUP(S)))
  generalInterpolation((D.C)(reduce(_+, eta)), D.A, vguessList, eta)$FFFG(S, SUP S)
else if F is Fraction S then
  vguessListF: Vector SUP F := vector(guessList)
  generalInterpolation((D.C)(reduce(_+, eta)), D.A, vguessListF, eta)$FFFGF(S, SUP S, SUP F)
else error "Type parameter F should be either equal to S or equal to Fraction S"

if F is S then
  vguessList: Vector SUP S := vector(guessList pretend List(SUP(S)))
  generalInterpolation((D.C)(sumEta), D.A, vguessList, sumEta, maxEta)$FFFG(S, SUP S)
else if F is Fraction S then
  vguessListF: Vector SUP F := vector(guessList)
  generalInterpolation((D.C)(sumEta), D.A, vguessListF, sumEta, maxEta)$FFFGF(S, SUP S, SUP F)
else error "Type parameter F should be either equal to S or equal to Fraction S"

  : Union("failed", Record(function: EXPRR, order: NNI)) ==
--tpd: maxDegree is defined to be nonnegative
-- (maxDegree(options)$GOPT0 = -1) and
((allDegrees(options)$GOPT0 = false) and zero?(last resi))
=> return "failed"
nonZeroCoefficient: Integer := 0

for i in 1..#resi repeat
    if not zero? resi.i then
        if zero? nonZeroCoefficient then
            nonZeroCoefficient := i
        else
            nonZeroCoefficient := 0
        break
    if not zero? nonZeroCoefficient then
        (freeOf?(exprList.nonZeroCoefficient, name op)) => return "failed"
    for e in list repeat
        if not zero? e then return "failed"
else
    resiSUPF := map(SUPF2SUPSUPF SUPS2SUPF #1, resi)
    $ListFunctions2(SUP S, SUP SUP F)
    iterate? := true;
    for d in guessDegree+1.. repeat
        c: SUP F := generalCoefficient(D.AF, vector testList,
        d, vector resiSUPF)
        $FFG(SUP F, UFPSSUPF)
        if not zero? c then
            iterate? := ground? c
        break
    iterate? => return "failed"
    g: SUP S
    if S has Field
    then g := leadingCoefficient(find(not zero? #1, reverse resi)::SUP(S))::SUP(S)
    else g := gcd resi
    resiF := map(SUPS2SUPF((#1 exquo g)::SUP(S)), resi)
    $ListFunctions2(SUP S, SUP F)

if debug(options)$GOPT0 then
    output(hconcat("trying possible solution ", resiF::OutputForm))
    $OutputPackage

-- transform each term into an expression

ex: List EXPRR := [makeEXPRR(D.AX, dummy, p, e) _
    for p in resiF for e in exprList]

-- transform the list of expressions into a sum of expressions

res: EXPRR
if displayAsGF(options)$GOPT0 then
res := evalADE(op, dummy, variableName(options).GOPT0::EXPRR,
indexName(options).GOPT0::EXPRR,
numerator reduce(_+, ex),
reverse initials)
$RecurrenceOperator(Integer, EXPRR)
ord: NNI := 0
--FIXME: checkResult doesn't really work yet for generating functions
else
res := evalRec(op, dummy, indexName(options).GOPT0::EXPRR,
indexName(options).GOPT0::EXPRR,
numerator reduce(_+, ex),
reverse initials)
$RecurrenceOperator(Integer, EXPRR)
ord: NNI := checkResult(res, indexName(options)GOPT0, _
#list, list, options)

[res, ord]$Record(function: EXPRR, order: NNI)

guessHPaux(list: List F, D: HPSPEC, options: LGOPT): GUESSRESULT ==
reslist: GUESSRESULT := []
listDegree := #list-1-safety(options)GOPT0
if listDegree < 0 then return reslist
a := functionName(options)GOPT0
op := operator a
x := variableName(options)GOPT0
dummy := new$Symbol

initials: List EXPRR := [coerce(e)@EXPRR for e in list]

guessS := (D.guessStream)(list2UFPSF list)
degreeS := D.degreeStream
testS := (D.testStream)(list2UFPSSUPF list)
exprS := (D.exprStream)(op(dummy::EXPRR)::EXPRR, dummy)
iterate?: Boolean := false -- this is necessary because the compiler
-- doesn't understand => "iterate" properly
-- the latter just leaves the current block, it
-- seems
for o in 2.. repeat
empty? rest(guessS, (o-1)::NNI) => break
guessDegree: Integer := listDegree-(degreeS.o)::Integer
if guessDegree < 0 => break
if debug(options)GOPT0 then
output(hconcat("Trying order ", o::OutputForm))$OutputPackage
output(hconcat("guessDegree is ", guessDegree::OutputForm))
$OutputPackage
if allDegrees(options)GOPT0 then
(o > guessDegree+2) => return reslist
--tpd: force NNI and Integer
maxEta: Integer := 1+maxDegree(options)$GOPT0::NNI::Integer
if maxEta = 0
then maxEta := guessDegree+1
else
maxParams := divide(guessDegree::NNI+1, o)
if debug(options)$GOPT0
then output(hconcat("maxParams: ", maxParams::OutputForm))
$OutputPackage
if maxParams.quotient = 0 and maxParams.remainder < o-1
then return reslist
--tpd: maxDegree is defined to be nonnegative
-- if ((maxDegree(options)$GOPT0 =~ -1) and
if ((maxDegree(options)$GOPT0::NNI::Integer < maxParams.quotient)) and
not (empty? rest(guessS, o) or
((newGuessDegree := listDegree-(degreeS.(o+1))::Integer) < 0) or
(((newMaxParams := divide(newGuessDegree::NNI+1, o+1))
 .quotient = 0) and
(newMaxParams.remainder < o)))
then iterate? := true
--tpd:maxDegree is defined to be nonnegative
-- else if ((maxDegree(options)$GOPT0 =~ -1) and
if (maxParams.quotient > maxDegree(options)$GOPT0::NNI::Integer)
then
--tpd:maxDegree is defined to be nonnegative
-- else
guessDegree := o*(1+maxDegree(options)$GOPT0::NNI::Integer)-2
eta: List NNI
:= [(if i < o 
     then maxDegree(options)$GOPT0::NNI + 1 
     else maxDegree(options)$GOPT0::NNI) 
    for i in 1..o]
else eta: List NNI
:= [(if i <= maxParams.remainder 
     then maxParams.quotient + 1 
     else maxParams.quotient)::NNI for i in 1..o]
if iterate?
then
iterate? := false
if debug(options)$GOPT0 then output("iterating")$OutputPackage
else
guessList: List SUP F := getListSUPF(guessS, o, guessDegree::NNI)
testList: List UFPSSUPF := entries complete first(testS, o)
exprList: List EXPRR := entries complete first(exprS, o)
if debug(options)$GOPT0 then
output("The list of expressions is")$OutputPackage
output(exprList::OutputForm)$OutputPackage
if allDegrees(options)$GOPT0 then
MS: Stream Matrix SUP S := guessInterpolate2(guessList,
guessDegree::NNI+1, maxEta::NNI, D)

repeat
  (empty? MS) => break
  M := first MS

  for i in 1..o repeat
    res := testInterpolant(entries column(M, i),
      list, testList, exprList, initials,
      guessDegree::NNI, D, dummy, op, options)
    (res case "failed") => "iterate"
    if not member?(res, reslist)
      then reslist := cons(res, reslist)
    if one(options)$GOPT0 then return reslist

  MS := rest MS
else
  M: Matrix SUP S := guessInterpolate(guessList, eta, D)

  for i in 1..o repeat
    res := testInterpolant(entries column(M, i),
      list, testList, exprList, initials,
      guessDegree::NNI, D, dummy, op, options)
    (res case "failed") => "iterate"
    if not member?(res, reslist)
      then reslist := cons(res, reslist)
    if one(options)$GOPT0 then return reslist

  reslist

guessHP(D: LGOPT -> HPSPEC): GUESSER == guessHPaux(#1, D #2, #2)

--tpd comment out the call to displayAsGF. it won't type check
guessADE(list: List F, options: LGOPT): GUESSRESULT ==
--tpd
  opts: LGOPT := cons(displayAsGF(true)$GuessOption, options)
  opts := options
  guessHPaux(list, diffHP opts, opts)
guessADE(list: List F): GUESSRESULT == guessADE(list, [])

guessAlg(list: List F, options: LGOPT) ==
guessADE(list, cons(maxDerivative(0)$GuessOption, options))

guessHolo(list: List F): GUESSRESULT == guessAlg(list, [])

guessHolo(list: List F, options: LGOPT): GUESSRESULT ==
guessADE(list, cons(maxPower(1)$GuessOption, options))

guessHolo(list: List F): GUESSRESULT == guessHolo(list, [])

guessPade(list: List F, options: LGOPT): GUESSRESULT ==
  opts := append(options, [maxDerivative(0)$GuessOption, maxPower(1)$GuessOption, allDegrees(true)$GuessOption])
guessADE(list, opts)

guessPade(list: List F): GUESSRESULT == guessPade(list, [])

if F has RetractableTo Symbol and S has RetractableTo Symbol then

  --tpd comment out the call to displayAsGF. it won't type check
  guessADE(q: Symbol): GUESSESSE ==
  --tpd
  --opts: LGOPT := cons(displayAsGF(true)$GuessOption, #2)
  opts := #2
  guessHPaux(#1, (diffHP q)(opts), opts)

  --tpd comment out the call to displayAsGF. it won't type check
  guessRec(list: List F, options: LGOPT): GUESSRESULT ==
  --tpd
  --opts: LGOPT := cons(displayAsGF(false)$GuessOption, options)
  opts := options
  guessHPaux(list, shiftHP opts, opts)

guessRec(list: List F): GUESSRESULT == guessRec(list, [])

guessPRec(list: List F, options: LGOPT): GUESSRESULT ==
guessRec(list, cons(maxPower(1)$GuessOption, options))

guessPRec(list: List F): GUESSRESULT == guessPRec(list, [])

guessRat(list: List F, options: LGOPT): GUESSRESULT ==
  opts := append(options, [maxShift(0)$GuessOption, maxPower(1)$GuessOption, allDegrees(true)$GuessOption])
  guessRec(list, opts)

guessRat(list: List F): GUESSRESULT == guessRat(list, [])
if F has RetractableTo Symbol and S has RetractableTo Symbol then

-- tpd comment out the call to displayAsGF. it won’t type check
guessRec(q: Symbol): GUESSER ==
-- opts: LGOPT := cons(displayAsGF(false)$GuessOption, #2)
  opts := #2
  guessHPaux(#1, (shiftHP q)(opts), opts)

-- tpd comment out the call to displayAsGF. it won’t type check
guessPRec(q: Symbol): GUESSER ==
-- opts: LGOPT := append([displayAsGF(false)$GuessOption,
-- maxPower(1)$GuessOption], #2)
  opts := #2
  guessHPaux(#1, (shiftHP q)(opts), opts)

-- tpd comment out the call to displayAsGF. it won’t type check
guessRat(q: Symbol): GUESSER ==
-- opts := append(#2, [displayAsGF(false)$GuessOption,
-- maxShift(0)$GuessOption, maxPower(1)$GuessOption,
-- allDegrees(true)$GuessOption])
  opts := #2
  guessHPaux(#1, (shiftHP q)(opts), opts)

guess(list: List F, guessers: List GUESSER, ops: List Symbol,
options: LGOPT): GUESSRESULT ==
maxLevel := maxLevel(options)$GOPT0
xx := indexName(options)$GOPT0
if debug(options)$GOPT0 then
  output(hconcat("guessing level ", maxLevel::OutputForm))$OutputPackage
  output(hconcat("guessing ", list::OutputForm))$OutputPackage
res: GUESSRESULT := []
len := #list :: PositiveInteger
if len <= 1 then return res
for guesser in guessers repeat
  res := append(guesser(list, options), res)
  if debug(options)$GOPT0 then
    output(hconcat("res ", res::OutputForm))$OutputPackage
  if one(options)$GOPT0 and not empty? res then return res
if (maxLevel = 0) then return res
if member?('guessProduct, ops) and not member?(0$F, list) then
  prodList: List F := [(list.(i+1))/(list.i) for i in 1..(len-1)]
  -- tpd: maxLevel is NNI
if not every?(one?, prodList) then
  var: Symbol := subscript('p, [len::OutputForm])
  prodGuess :=
    [[coerce(list.(guess.order+1))
      * product(guess.function, _
      equation(var, _
        (guess.order)::EXPRR..xx::EXPRR-1)), _
      guess.order] _
    for guess in guess(prodList, guessers, ops, options)$%]
-- tpd: this is broken
-- append([[indexName(var)$GuessOption)::Symbol,_
-- (maxLevel(maxLevel-1)$GuessOption)::NNI],_
-- options))$%]

if debug(options)$GOPT0 then
  output(hconcat("prodGuess "::OutputForm,
    prodGuess::OutputForm))
  $OutputPackage

for guess in prodGuess
  | not any?(guess.function = #1.function, res) repeat
    res := cons(guess, res)

if one(options)$GOPT0 and not empty? res then return res

if member?('guessSum, ops) then
  sumList: List F := [(list.(i+1))-(list.i) for i in 1..(len-1)]
  -- tpd:maxLevel is NNI
  if not every?(#1=sumList.1, sumList) then
    var: Symbol := subscript('s, [len::OutputForm])
    sumGuess :=
      [[coerce(list.(guess.order+1)) _
        + summation(guess.function, _
        equation(var, _
          (guess.order)::EXPRR..xx::EXPRR-1)), _
        guess.order] _
      for guess in guess(sumList, guessers, ops, options)$%]
-- tpd: this is broken
-- for guess in guess(sumList, guessers, ops,_
-- append([[indexName(var)$GuessOption)::Symbol,_
-- (maxLevel(maxLevel-1)$GuessOption)::NNI],_
-- options))$%]

for guess in sumGuess
  | not any?(guess.function = #1.function, res) repeat
    res := cons(guess, res)

res

guess(list: List F): GUESSRESULT ==
guess(list, [guessADE%, guessRec%], ['guessProduct', 'guessSum], [])

guess(list: List F, options: LGOPT): GUESSRESULT ==
  guess(list, [guessADE%, guessRat%], ['guessProduct', 'guessSum],
  options)

guess(list: List F, guessers: List GUESSER, ops: List Symbol)
  : GUESSRESULT ==
  guess(list, guessers, ops, [])

---

--- GUESS.dotabb ---

"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUES"
"RECO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RECO"
"GUESS" -> "RECO"

---

package GUESSAN GuessAlgebraicNumber

--- GuessAlgebraicNumber.input ---

)set break resume
)sys rm -f GuessAlgebraicNumber.output
)spool GuessAlgebraicNumber.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessAlgebraicNumber
--R
--R GuessAlgebraicNumber is a package constructor
--R Abbreviation for GuessAlgebraicNumber is GUESSAN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESSAN
--R
--R-------------------------- Operations --------------------------
--R diffHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(AlgebraicNumber) -> ... -> Expression(Integer)),C: (NonNegativeInteger -> List(AlgebraicNumber))
--R diffHP : Symbol -> (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(AlgebraicNumber) -> ... -> Expression(Integer)),C: (NonNegativeInteger -> List(AlgebraicNumber))) if AlgebraicNumber has RETRACT(SYMBOL)
--R guess : List(AlgebraicNumber) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(AlgebraicNumber),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
This package exports guessing of sequences of rational functions

See Also:
  o )show GuessAlgebraicNumber
GuessAlgebraicNumber (GUESSAN)

Exports:
- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

--- package GUESSAN GuessAlgebraicNumber ---

)abbrev package GUESSAN GuessAlgebraicNumber
++ Description:
++ This package exports guessing of sequences of rational functions

GuessAlgebraicNumber() == Guess(AlgebraicNumber, AlgebraicNumber,
Expression Integer,
AlgebraicNumber,
id$MappingPackage1(AlgebraicNumber),
coerce$Expression(Integer))

---

--- GUESSAN.dotabb ---

"GUESSAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSAN"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSAN" -> "GUESS"

---
package GUESSF GuessFinite

--- GuessFinite.input ---

)set break resume
)sys rm -f GuessFinite.output
)spool GuessFinite.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessFinite
--R
--R GuessFinite(F: Join(FiniteFieldCategory,ConvertibleTo(Integer))) is a package constructor
--R Abbreviation for GuessFinite is GUESSF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESSF
--R
--R--------------------------------- Operations ---------------------------------
--R diffHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(F) -> Stream(UnivariateFormalPowerSeries(F))
--R diffHP : Symbol -> (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(F) -> Stream(UnivariateFormalPowerSeries(F))
--R guess : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessAlg : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessAlg : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHP : (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(F) -> Stream(UnivariateFormalPowerSeries(F))
--R guessHolo : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHolo : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : List(F) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : (List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : Symbol -> ((List(F),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
GuessFinite (GUESSF)

This package exports guessing of sequences of numbers in a finite field.

See Also:
- `)show GuessFinite`

Exports:
- `diffHP`  
- `guess`  
- `guessADE`  
- `guessAlg`  
- `guessBinRat`  
- `guessExpRat`  
- `guessHP`  
- `guessHolo`  
- `guessPRec`  
- `guessPade`  
- `guessRec`  
- `shiftHP`

package GUESSF GuessFinite
package GUESSF1 GuessFiniteFunctions

--R GuessFiniteFunctions(F: Join(FiniteFieldCategory,ConvertibleTo(Integer))) is a package constructor
--R Abbreviation for GuessFiniteFunctions is GUESSF1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESSF1
--R
--R---------------------------------------- Operations ----------------------------------------
--R F2EXPRR : F -> Expression(Integer)
--R
--E 1

)spool
)lisp (bye)
This package exports guessing of sequences of numbers in a finite field

See Also:
- )show GuessFiniteFunctions

---

**GuessFiniteFunctions** (GUESSF1)

Exports:
- F2EXPRR

---

**package GUESSF1 GuessFiniteFunctions**

)abbrev package GUESSF1 GuessFiniteFunctions
++ Description:
++ This package exports guessing of sequences of numbers in a finite field

GuessFiniteFunctions(F:Join(FiniteFieldCategory, ConvertibleTo Integer)):

Exports == Implementation where

EXPRR => Expression Integer

Exports == with
F2EXPRR: F -> EXPRR

Implementation == add

F2EXPRR(p: F): EXPRR == convert(p)@Integer::EXPRR

---

package GUESSINT GuessInteger

--- GuessInteger.input ---

)set break resume
)sys rm -f GuessInteger.output
)spool GuessInteger.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessInteger

--R
--R GuessInteger is a package constructor
--R Abbreviation for GuessInteger is GUESSINT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESSINT

--R----------------------------- Operations -----------------------------
--R diffHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer)) -> String)
--R diffHP : Symbol -> (List(GuessOption)) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer)) -> String)
--R guess : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Integer)),List(((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Integer)),List(((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))

--R---------------------------- Operations --------------------------------
--R diffHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer)) -> String
--R diffHP : Symbol -> (List(GuessOption)) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer)) -> String
--R guess : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Integer)),List(((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Integer)),List(((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))

---

"GUESSF1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSF1"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSF1" -> "GUESS"
CHAPTER 8. CHAPTER G

--R guessADE : Symbol -> ((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R guessAlg : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessAlg : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : Symbol -> ((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R guessExpRat : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : Symbol -> ((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R guessHP : (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer)) -> ...))
--R guessHolo : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHolo : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : Symbol -> ((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R guessPade : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : Symbol -> ((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R guessRat : (List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : List(Fraction(Integer)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : Symbol -> ((List(Fraction(Integer)),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R shiftHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer)) -> ...)
--R shiftHP : Symbol -> (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Integer))

--E 1

)spool
)lisp (bye)

---------

GuessInteger.help

--------------------------------------

GuessInteger examples

--------------------------------------

This package exports guessing of sequences of rational numbers

See Also:
     o )show GuessInteger

---------
GuessInteger (GUESSINT)

Exports:
- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

-- concerning algebraic functions, it may make sense to look at A.J. van der Poorten, Power Series Representing Algebraic Functions, Section 6.

)abbrev package GUESSINT GuessInteger
++ Description:
++ This package exports guessing of sequences of rational numbers

GuessInteger() == Guess(Fraction Integer, Integer, Expression Integer,
Fraction Integer,
id$MappingPackage1(Fraction Integer),
coerce$Expression(Integer))

— GUESSINT.dotabb —

"GUESSINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESSINT"]
"GUESS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=GUESS"]
"GUESSINT" -> "GUESS"

— package GUESSP GuessPolynomial —
--- GuessPolynomial.input ---

)set break resume
)sys rm -f GuessPolynomial.output
)spool GuessPolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessPolynomial

--R GuessPolynomial is a package constructor
--R Abbreviation for GuessPolynomial is GUESSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESSP

--R ----------------------------- Operations -----------------------------
--R
--R diffHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Polynomial(Integer))) -> ... -> Expression(Integer)),C: (NonNegativeInteger -> List(Polynomial(Integer))))
--R diffHP : Symbol -> (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Polynomial(Integer))) -> ... -> Expression(Integer)),C: (NonNegativeInteger -> List(Polynomial(Integer))))
--R guess : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guess : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessADE : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessAlg : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessAlg : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessAlg : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessBinRat : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessExpRat : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHP : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHP : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHP : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHolo : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHolo : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessHolo : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPRec : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessPade : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRat : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRec : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRec : (List(Fraction(Polynomial(Integer))),List(GuessOption)) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R guessRec : List(Fraction(Polynomial(Integer))) -> List(Record(function: Expression(Integer),order: NonNegativeInteger))
--R shiftHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Polynomial(Integer)))) -> ... -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
--R shiftHP : Symbol -> (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Polynomial(Integer)))) -> ... -> List(Record(function: Expression(Integer),order: NonNegativeInteger))))
--R shiftHP : List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Polynomial=Integer)))) -> ... -> List(Record(function: Expression(Integer),order: NonNegativeInteger)))
GuessPolynomial (GUESSP)

Exports:

- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

--R shiftHP : Symbol -> (List(GuessOption) -> Record(guessStream: (UnivariateFormalPowerSeries(Fraction(Polynomial)) has RETRACT(SYMBOL) and Polynomial(Integer) has RETRACT(SYMBOL) --R
--E 1

)spool
)lisp (bye)

---

-- GuessPolynomial.help --

====================================================================
GuessPolynomial examples
====================================================================

This package exports guessing of sequences of rational functions

See Also:
o )show GuessPolynomial

---
This package exports guessing of sequences of rational functions

```plaintext
GuessPolynomial() == Guess(Fraction Polynomial Integer, Polynomial Integer,
   Expression Integer,
   Fraction Polynomial Integer,
   id$MappingPackage1(Fraction Polynomial Integer),
   coerce$Expression(Integer))
```

---

package GUESSUP GuessUnivariatePolynomial

---

```plaintext
)set break resume
)sys rm -f GuessUnivariatePolynomial.output
)spool GuessUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show GuessUnivariatePolynomial
--R
--R GuessUnivariatePolynomial(q: Symbol) is a package constructor
--R Abbreviation for GuessUnivariatePolynomial is GUESSUP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for GUESSUP
--R
--R----------------------------- Operations ----------------------------
--R diffHP : List(GuessOption) -> HPSPEC
--R diffHP : Symbol -> (List(GuessOption) -> HPSPEC)
--R guess : List(Fraction(MyUnivariatePolynomial(q,Integer))) -> List(Record(function: MyExpression(q,Integer),order: NonNegativeInteger))
--R guess : (List(Fraction(MyUnivariatePolynomial(q,Integer))),List(GuessOption)) -> List(Record(function: MyExpression(q,Integer),order: NonNegativeInteger))
--R guess : ... NonNegativeInteger)))),List(Symbol)) -> List(Record(function: MyExpression(q,Integer),order: NonNegativeInteger))
```
This package exports guessing of sequences of univariate rational functions.

See Also:
- \( \text{show GuessUnivariatePolynomial} \)
GuessUnivariatePolynomial (GUESSUP)

Exports:
- diffHP
- guess
- guessADE
- guessAlg
- guessBinRat
- guessExpRat
- guessHP
- guessHolo
- guessPRec
- guessPade
- guessRat
- guessRec
- shiftHP

— package GUESSUP GuessUnivariatePolynomial —

)abbrev package GUESSUP GuessUnivariatePolynomial
++ Description:
++ This package exports guessing of sequences of univariate rational functions

GuessUnivariatePolynomial(q: Symbol): Exports == Implementation where

| package GUESSUP GuessUnivariatePolynomial |
| (List F, LGOPT) -> GUESSRESULT |

Exports == with

- guess: List F -> GUESSRESULT
  ++ \spad{guess l} applies recursively \spadfun{guessRec} and
  ++ \spadfun{guessADE} to the successive differences and quotients of
  ++ the list. Default options as described in
  ++ \spadtype{GuessOptionFunctions0} are used.

- guess: (List F, LGOPT) -> GUESSRESULT
  ++ \spad{guess(l, options)} applies recursively \spadfun{guessRec}
  ++ and \spadfun{guessADE} to the successive differences and quotients
  ++ of the list. The given options are used.
guess: (List F, List GUESSER, List Symbol) -> GUESSRESULT
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ guessSum and quotients if ops contains the symbol guessProduct to
++ the list. Default options as described in
++ \spadtype{GuessOptionFunctions0} are used.

guess: (List F, List GUESSER, List Symbol, LGOPT) -> GUESSRESULT
++ \spad{guess(l, guessers, ops)} applies recursively the given
++ guessers to the successive differences if ops contains the symbol
++ \spad{guessSum} and quotients if ops contains the symbol
++ \spad{guessProduct} to the list. The given options are used.

guessExpRat: List F -> GUESSRESULT
++ \spad{guessExpRat l} tries to find a function of the form
++ \(n\to(a+b \cdot n)^n r(n)\), where \(r(n)\) is a rational function, that fits
++ \(l\).

guessExpRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessExpRat(l, options)} tries to find a function of the
++ form \(n\to(a+b \cdot n)^n r(n)\), where \(r(n)\) is a rational function, that
++ fits \(l\).

guessBinRat: List F -> GUESSRESULT
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form \(n\to\text{binomial}(a+b \cdot n, n) r(n)\), where \(r(n)\) is a rational
++ function, that fits \(l\).

guessBinRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessBinRat(l, options)} tries to find a function of the
++ form \(n\to\text{binomial}(a+b \cdot n, n) r(n)\), where \(r(n)\) is a rational
++ function, that fits \(l\).

-- if \(F\) has RetractableTo \text{Symbol} and \(S\) has RetractableTo \text{Symbol} then

guessExpRat: Symbol -> GUESSER
++ \spad{guessExpRat q} returns a guesser that tries to find a
++ function of the form \(n\to(a+b \cdot q^n)^n r(q^n)\), where \(r(q^n)\) is a
++ \(q\)-rational function, that fits \(l\).

guessBinRat: Symbol -> GUESSER
++ \spad{guessBinRat q} returns a guesser that tries to find a
++ function of the form \(n\to\text{qbinomial}(a+b \cdot n, n) r(n)\), where \(r(q^n)\) is a
++ \(q\)-rational function, that fits \(l\).

guessHP: (LGOPT -> HPSPEC) -> GUESSER
++ \spad{guessHP f} constructs an operation that applies Hermite-Pade
++ approximation to the series generated by the given function \(f\).
guessADE: List F -> GUESSRESULT
++ \spad{guessADE l} tries to find an algebraic differential equation
++ for a generating function whose first Taylor coefficients are
++ given by \( l \), using the default options described in
++ \spadtype{GuessOptionFunctions0}.

guessADE: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessADE(l, options)} tries to find an algebraic
differential equation for a generating function whose first Taylor
coefficients are given by \( l \), using the given options.

guessAlg: List F -> GUESSRESULT
++ \spad{guessAlg l} tries to find an algebraic equation for a
generating function whose first Taylor coefficients are given by
\( l \), using the default options described in
++ \spadtype{GuessOptionFunctions0}. It is equivalent to
++ \spadfun{guessADE}(1, \maxDerivative == 0).

guessAlg: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessAlg(l, options)} tries to find an algebraic equation
++ for a generating function whose first Taylor coefficients are
++ given by \( l \), using the given options. It is equivalent to
++ \spadfun{guessADE}(1, options) with \spad{maxDerivative == 0}.

guessHolo: List F -> GUESSRESULT
++ \spad{guessHolo l} tries to find an ordinary linear differential
++ equation for a generating function whose first Taylor coefficients
++ are given by \( l \), using the default options described in
++ \spadtype{GuessOptionFunctions0}. It is equivalent to
++ \spadfun{guessADE}\spad{(1, \maxPower == 1)}.

guessHolo: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessHolo(l, options)} tries to find an ordinary linear
differential equation for a generating function whose first Taylor
coefficients are given by \( l \), using the given options. It is
++ equivalent to \spadfun{guessADE}\spad{(1, options)} with
++ \spad{maxPower == 1}.

guessPade: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessPade(l, options)} tries to find a rational function
++ whose first Taylor coefficients are given by \( l \), using the given
++ options. It is equivalent to \spadfun{guessADE}\spad{(1,
++ \maxDerivative == 0, \maxPower == 1, allDegrees == true)}.

guessPade: List F -> GUESSRESULT
++ \spad{guessPade(l, options)} tries to find a rational function
++ whose first Taylor coefficients are given by \( l \), using the default
++ options described in \spadtype{GuessOptionFunctions0}. It is
++ equivalent to \spadfun{guessADE}\spad{(1, options)} with
guessRec: List F -> GUESSRESULT
++ \spad{guessRec 1} tries to find an ordinary difference equation
++ whose first values are given by 1, using the default options
++ described in \spadtype{GuessOptionFunctions0}.

guessRec: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessRec(1, options)} tries to find an ordinary difference equation whose first values are given by 1, using the given options.

guessPRec: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessPRec(1, options)} tries to find a linear recurrence with polynomial coefficients whose first values are given by 1, using the given options. It is equivalent to
++ \spadfun{guessRec}\spad{(1, options)} with \spad{maxPower == 1}.

guessPRec: List F -> GUESSRESULT
++ \spad{guessPRec 1} tries to find a linear recurrence with polynomial coefficients whose first values are given by 1, using the default options described in
++ \spadtype{GuessOptionFunctions0}. It is equivalent to
++ \spadfun{guessRec}\spad{(1, maxPower == 1)}.

guessRat: (List F, LGOPT) -> GUESSRESULT
++ \spad{guessRat(1, options)} tries to find a rational function whose first values are given by 1, using the given options. It is equivalent to \spadfun{guessRec}\spad{(1, maxShift == 0, maxPower == 1, allDegrees == true)}.

guessRat: List F -> GUESSRESULT
++ \spad{guessRat 1} tries to find a rational function whose first values are given by 1, using the default options described in
++ \spadtype{GuessOptionFunctions0}. It is equivalent to
++ \spadfun{guessRec}\spad{(1, maxShift == 0, maxPower == 1, allDegrees == true)}.

diffHP: LGOPT -> HPSPEC
++ \spad{diffHP options} returns a specification for Hermite-Pade approximation with the differential operator

shiftHP: LGOPT -> HPSPEC
++ \spad{shiftHP options} returns a specification for Hermite-Pade approximation with the shift operator

-- if F has RetractableTo Symbol and S has RetractableTo Symbol then

shiftHP: Symbol -> (LGOPT -> HPSPEC)
++ \spad{shiftHP options} returns a specification for
++ Hermite-Pade approximation with the $q$-shift operator

diffHP: Symbol -> (LGOPT -> HPSPEC)
++ \spad{diffHP options} returns a specification for Hermite-Pade
++ approximation with the $q$-dilation operator

guessRec: Symbol -> GUESSER
++ \spad{guessRec q} returns a guesser that finds an ordinary
++ $q$-difference equation whose first values are given by l, using
++ the given options.

guessPRec: Symbol -> GUESSER
++ \spad{guessPRec q} returns a guesser that tries to find
++ a linear $q$-recurrence with polynomial coefficients whose first
++ values are given by l, using the given options. It is
++ equivalent to \spad{fun{guessRec}\spad{(q)}} with
++ \spad{maxPower == 1}.

guessRat: Symbol -> GUESSER
++ \spad{guessRat q} returns a guesser that tries to find a
++ $q$-rational function whose first values are given by l, using
++ the given options. It is equivalent to \spad{fun{guessRec}\spad{(q)}} with
++ \spad{(l, maxShift == 0, maxPower == 1, allDegrees == true)}.

guessADE: Symbol -> GUESSER
++ \spad{guessADE q} returns a guesser that tries to find an
++ algebraic differential equation for a generating function whose
++ first Taylor coefficients are given by l, using the given
++ options.

Implementation == Guess(Fraction UP, UP,
       MyExpression(q, Integer),
       Fraction UP,
       id$MappingPackage1(Fraction UP),
       coerce$MyExpression(q, Integer))
Chapter 9

Chapter H

package HB HallBasis

— HallBasis.input —

)set break resume
)sys rm -f HallBasis.output
)spool HallBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show HallBasis
--R
--R HallBasis is a package constructor
--R Abbreviation for HallBasis is HB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for HB
--R
--R---------------------------------------------------------- Operations -------------------------------
--R lfunc : (Integer,Integer) -> Integer
--R generate : (NonNegativeInteger,NonNegativeInteger) -> Vector(List(Integer))
--R inHallBasis? : (Integer,Integer,Integer,Integer) -> Boolean
--R
--E 1

)spool
)lisp (bye)
HallBasis (HB)

Exports:
generate  inHallBasis?  lfunc

— package HB HallBasis —

)abbrev package HB HallBasis
++ Author : Larry Lambe
++ Date Created : August 1988
++ Date Last Updated : March 9 1990
++ Description:
++ Generate a basis for the free Lie algebra on n
genators over a ring R with identity up to basic commutators
++ of length c using the algorithm of P. Hall as given in Serre’s book Lie Groups -- Lie Algebras
HallBasis() : Export == Implement where

B ==> Boolean
I ==> Integer
NNI ==> NonNegativeInteger
VI ==> Vector Integer
VLI ==> Vector List Integer

Export ==> with

lfunc : (I,I) -> I
++ lfunc(d,n) computes the rank of the nth factor in the
++ lower central series of the free d-generated free Lie
++ algebra; This rank is d if n = 1 and binom(d,2) if
++ n = 2

inHallBasis? : (I,I,I,I) -> B
++ inHallBasis?(numberOfGens, leftCandidate, rightCandidate, left)
++ tests to see if a new element should be added to the P. Hall
++ basis being constructed.
++ The list \spad{[leftCandidate,wt,rightCandidate]}
++ is included in the basis if in the unique factorization of
++ rightCandidate, we have left factor leftOfRight, and
++ leftOfRight <= leftCandidate

generate : (NNI,NNI) -> VLI
++ generate(numberOfGens, maximalWeight) generates a vector of
++ elements of the form \spad{[left,weight,right]} which represents a
++ P. Hall basis element for the free lie algebra on numberOfGens
++ generators. We only generate those basis elements of weight
++ less than or equal to maximalWeight

Implement ==> add

lfunc(d,n) ==
  n < 0 => 0
  n = 0 => 1
  n = 1 => d
  sum:I := 0
  m:I
  for m in 1..(n-1) repeat
    if n rem m = 0 then
      sum := sum + m * lfunc(d,m)
  res := (d**(n::NNI) - sum) quo n

inHallBasis?(n,i,j,l) ==
  i >= j => false
  j <= n => true
  l <= i => true
  false

generate(n:NNI,c:NNI) ==
  gens:=n
  maxweight:=c
siz:I := 0
for i in 1 .. maxweight repeat siz := siz + lfunc(gens,i)
v:VLI:= new(siz::NNI,[])
for i in 1 .. gens repeat v(i) := [0, 1, i]
firstindex:VI := new(maxweight::NNI,0)
wt:I := 1
firstindex(1) := 1
numComms:I := gens
newNumComms:I := numComms
done:B := false
while not done repeat
    wt := wt + 1
    if wt > maxweight then done := true
    else
        firstindex(wt) := newNumComms + 1
        leftIndex := 1
        -- cW == complimentaryWeight
        cW:I := wt - 1
        while (leftIndex <= numComms) and (v(leftIndex).2 <= cW) repeat
            for rightIndex in firstindex(cW)..(firstindex(cW+1) - 1) repeat
                if inHallBasis?(gens,leftIndex,rightIndex,v(rightIndex).1) then
                    newNumComms := newNumComms + 1
                    v(newNumComms) := [leftIndex,wt,rightIndex]
                leftIndex := leftIndex + 1
            cW := wt - v(leftIndex).2
        numComms := newNumComms
    v

——-

— HB.dotabb —

"HB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=HB"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"HB" -> "IVECTOR"

——-

package HEUGCD HeuGcd

— HeuGcd.input —

)set break resume
)sys rm -f HeuGcd.output
This package provides the functions for the heuristic integer gcd.
Geddes's algorithm, for univariate polynomials with integer coefficients

See Also:
\( \text{HeuGcd} \)
HeuGcd (HEUGCD)

Exports:
content gcd gcdcofact gcdcofactprim gcdprim lintgcd

— package HEUGCD HeuGcd —

)abbrev package HEUGCD HeuGcd
++ Author: P.Gianni
++ Date Last Updated: 13 September 94
++ Description:
++ This package provides the functions for the heuristic integer gcd.
++ Geddes's algorithm, for univariate polynomials with integer coefficients

HeuGcd (BP):C == T
where
BP : UnivariatePolynomialCategory Integer
Z ==> Integer
ContPrim ==> Record(cont:Z,prim:BP)

C == with
  gcd : List BP -> BP
  ++ gcd([f1,..,fk]) = gcd of the polynomials fi.
  ++
  ++ gcd([671*671*x^2-1,671*671*x^2+2*671*x+1])
  ++ gcd([7*x^2+1,(7*x^2+1)^2])
  
gcdprim : List BP -> BP
  ++ gcdprim([f1,..,fk]) = gcd of k PRIMITIVE univariate polynomials
  
gcdcofact : List BP -> List BP
  ++ gcdcofact([f1,..fk]) = gcd and cofactors of k univariate polynomials.
  
gcdcofactprim : List BP -> List BP
  ++ gcdcofactprim([f1,..fk]) = gcd and cofactors of k
  ++ primitive polynomials.
  
content : List BP -> List Z
  ++ content([f1,..,fk]) = content of a list of univariate polynomials
lintgcd : List Z -> Z
++ lintgcd([a1,..,ak]) = gcd of a list of integers

T == add

PI ==> PositiveInteger
NNI ==> NonNegativeInteger
Cases ==> Union("gcdprim","gcd","gcdfactprim","gcdfact")
import ModularDistinctDegreeFactorizer BP

--local functions
localgcd : List BP -> List BP
constNotZero : BP -> Boolean
height : BP -> PI
genpoly : (Z,PI) -> BP
negShiftz : (Z,PI) -> Z
internal : (Cases,List BP) -> List BP
constcase : (List NNI,List BP) -> List BP
lincase : (List NNI,List BP) -> List BP
myNextPrime : (Z,NNI) -> Z

bigPrime:= prevPrime(2**26)$IntegerPrimesPackage(Integer)

myNextPrime(val:Z,bound:NNI) : Z == nextPrime(val)$IntegerPrimesPackage(Z)

constNotZero(f : BP) : Boolean == (degree f = 0) and ~ (zero? f)

negShiftz(n:Z,Modulus:PI):Z ==
n < 0 => n:= n+Modulus
n > (Modulus quo 2) => n-Modulus
n
--compute the height of a polynomial
height(f:BP):PI ==
k:PI:=1
while f^=0 repeat
  k:=max(k,abs(leadingCoefficient(f)$Z)::PI)
  f:=reductum f
k

--reconstruct the polynomial from the value-adic representation of --dval.
genpoly(dval:Z,value:PI):BP ==
d:=0$BP
val:=dval
for i in 0.. while (val^=0) repeat
  val1:=negShiftz(val rem value,value)
  d:= d+monomial(val1,i)
  val:=(val-val1) quo value
d
--gcd of a list of integers

\[
\text{lntgcd}(lval: \text{List(Z)}): \text{Z} =\\
\text{empty? } lval \Rightarrow 0 \text{Z}\\
\text{member?(1,lval) } \Rightarrow 1 \text{Z}\\
\text{lval:=sort((z1,z2) +-> z1<z2,lval)}\\
\text{val:=lval.first}\\
\text{for val1 in lval.rest while \(~(val=1)\) repeat val:=gcd(val,val1)}\\
\text{val}
\]

--content for a list of univariate polynomials

\[
\text{content}(\text{listf: List BP }): \text{List(Z)} =\\
[\text{lntgcd coefficients } f \text{ for } f \text{ in listf}]
\]

--content of a list of polynomials with the relative primitive parts

\[
\text{conptrim}(\text{listf: List BP }): \text{List(ContPrim)} =\\
[[c:=\text{lntgcd coefficients } f,(f \text{ exquo } c)::BP]$\text{ContPrim} \text{ for } f \text{ in listf}]
\]

-- one polynomial is constant, remark that they are primitive

\[
\text{constcase}(\text{listdeg: List NNI }, \text{listf: List BP }): \text{List BP} =\\
\text{lind:=select(constNotZero,listf)}\\
\text{empty? } \text{lind} \Rightarrow\\
\text{member?(1,listdeg) } \Rightarrow \text{lincase(listdeg,listf)}\\
\text{localgcd listf}\\
or/[n>0 \text{ for } n \text{ in listdeg} ] \Rightarrow \text{cons(1$BP,listf)}\\
\text{lclistf:List(Z):=} \text{[leadingCoefficient } f \text{ for } f \text{ in listf]}\\
d=\text{lntgcd(lclistf)}\\
d=1 \Rightarrow \text{cons(1$BP,listf)}\\
\text{cons(d::BP,[(lcf quo d)::BP for lcf in lclistf]})
\]

\[
\text{testDivide}(\text{listf: List BP}, g:BP): \text{Union(List BP, "failed") =\\}
\text{result:List BP := []}\\
\text{for f in listf repeat }\\
\text{if (f1:=f exquo g) case "failed" then return "failed" }\\
\text{result := cons(f1::BP,result) }\\
\text{reverse!(result)}
\]

--one polynomial is linear, remark that they are primitive

\[
\text{lincase}(\text{listdeg: List NNI }, \text{listf: List BP }): \text{List BP} =\\
n:= \text{position(1,listdeg)}\\
g:=\text{listf.n}\\
\text{result:=[g]}\\
\text{for f in listf repeat }\\
\text{if (f1:=f exquo g) case "failed" then return cons(1$BP,listf)}\\
\text{result := cons(f1::BP,result) }\\
\text{reverse(result)}
\]

\[
\text{IMG := InnerModularGcd(Z,BP,67108859,myNextPrime)}
\]
mindegpol(f:BP, g:BP):BP ==
  degree(g) < degree (f) => g
  f

--local function for the gcd among n PRIMITIVE univariate polynomials
localgcd(listf:List BP ):List BP ==
  hgt:="min"/[height(f) for f in listf|~zero? f]
  answr:=2+2*hgt
  minf := "mindegpol"/[f for f in listf|~zero? f]
  (result := testDivide(listf, minf)) case List(BP) =>
    cons(minf, result::List BP)
  if degree minf < 100 then for k in 1..10 repeat
    listval:=[f answr for f in listf]
    dval:=lintgcd(listval)
    dd:=genpoly(dval,answr)
    contd:=content(dd)
    d:=(dd exquo contd)::BP
    result:List BP :=[d]
    flag : Boolean := true
    for f in listf while flag repeat
      (f1:=f exquo d) case "failed" => flag:=false
      result := cons (f1::BP,result)
    if flag then return reverse(result)
  nvalue:= answr*832040 quo 317811
  if ((nvalue + answr) rem 2) = 0 then nvalue:=nvalue+1
  answr:=nvalue::PI
  gg:=modularGcdPrimitive(listf)$IMG
  cons(gg,[(f exquo gg) :: BP for f in listf])

--internal function:it evaluates the gcd and avoids duplication of
--code.
internal(flag:Cases,listf:List BP ):List BP ==
  --special cases
  listf=[] => [1$BP]
  (nlf:=#listf)=1 => [first listf,1$BP]
  minpol:=1$BP
  -- extract a monomial gcd
  mdeg:= "min"/[minimumDegree f for f in listf]
  if mdeg>0 then
    minpol1:= monomial(1,mdeg)
    listf:=[(f exquo minpol1)::BP for f in listf]
    minpol:=minpol*minpol1
  -- make the polynomials primitive
  Cgcd:List(Z):=[]
  contgcd : Z := 1
  if (flag case "gcd") or (flag case "gcdcofact") then
    contlistf:List(ContPrim):=contprim(listf)
    Cgcd:=[term.cont for term in contlistf]
    contgcd:=lintgcd(Cgcd)
  listf:List BP :=[term.prim for term in contlistf]
\begin{verbatim}
minpol := contgcd*minpol
listdeg := [degree f for f in listf ]
f := first listf
if positiveRemainder(leadingCoefficient(f), bigPrime) ~= 0 then
  for g in rest listf repeat
    lcg := leadingCoefficient(g)
    if positiveRemainder(lcg, bigPrime) = 0 then
      leave
    f := gcd(f, g, bigPrime)
  if degree f = 0 then return cons(minpol, listf)
ans := List BP :=
--one polynomial is constant
member?(0, listdeg) => constcase(listdeg, listf)
--one polynomial is linear
member?(1, listdeg) => lincase(listdeg, listf)
localgcd(listf)
(result, ans) := (first ans*minpol, rest ans)
if (flag case "gcdcofact") then
  ans := [(p quo contgcd)*q for p in Gcd for q in ans]
cons(result, ans)

--gcd among n PRIMITIVE univariate polynomials
gcdprim (listf: List BP): BP == first internal("gcdprim", listf)

--gcd and cofactors for n PRIMITIVE univariate polynomials
gdcfactprim (listf: List BP): List BP == internal("gdcfctprim", listf)

--gcd for n generic univariate polynomials.
gcd (listf: List BP): BP == first internal("gcd", listf)

--gcd and cofactors for n generic univariate polynomials.
gdcfact (listf: List BP): List BP == internal("gdcfact", listf)
\end{verbatim}
package IDECOMP IdealDecompositionPackage

— IdealDecompositionPackage.input —

)set break resume
)sys rm -f IdealDecompositionPackage.output
)spool IdealDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IdealDecompositionPackage
--R
--R IdealDecompositionPackage(vl: List(Symbol),nv: NonNegativeInteger) is a package constructor
--R Abbreviation for IdealDecompositionPackage is IDECOMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IDECOMP
--R
--R------------------------------- Operations --------------------------------
--R contract : (PolynomialIdeals(Fraction(Integer),DirectProduct(nv,NonNegativeInteger),OrderedVariableList(vl)),OrderedVariableList(vl),DistributedMultivariatePolynomial(vl,Fraction(Integer)))
--R primaryDecomp : PolynomialIdeals(Fraction(Integer),DirectProduct(nv,NonNegativeInteger),OrderedVariableList(vl)),OrderedVariableList(vl),DistributedMultivariatePolynomial(vl,Fraction(Integer)))
--R prime? : PolynomialIdeals(Fraction(Integer),DirectProduct(nv,NonNegativeInteger),OrderedVariableList(vl))
--R radical : PolynomialIdeals(Fraction(Integer),DirectProduct(nv,NonNegativeInteger),OrderedVariableList(vl))
--R zeroDimPrimary? : PolynomialIdeals(Fraction(Integer),DirectProduct(nv,NonNegativeInteger),OrderedVariableList(vl))
--R zeroDimPrime? : PolynomialIdeals(Fraction(Integer),DirectProduct(nv,NonNegativeInteger),OrderedVariableList(vl))
--R
--E 1

)spool
)lisp (bye)
This package provides functions for the primary decomposition of polynomial ideals over the rational numbers. The ideals are members of the PolynomialIdeals domain, and the polynomial generators are required to be from the DistributedMultivariatePolynomial domain.

See Also:
- )show IdealDecompositionPackage

Exports:
- contract primaryDecomp prime?
- radical zeroDimPrimary?
- zeroDimPrime?

)abbrev package IDECOMP IdealDecompositionPackage
++ Author: P. Gianni
++ Date Created: summer 1986
++ Description:
++ This package provides functions for the primary decomposition of polynomial ideals over the rational numbers. The ideals are members of the PolynomialIdeals domain, and the polynomial generators are required to be from the DistributedMultivariatePolynomial domain.
IdealDecompositionPackage(vl, nv) : C == T -- take away nv, now doesn't
-- compile if it isn't there

where

vl : List Symbol
nv : NonNegativeInteger
Z ==> Integer -- substitute with PFE cat
Q ==> Fraction Z
F ==> Fraction P
P ==> Polynomial Z
UP ==> SparseUnivariatePolynomial P
Expon ==> DirectProduct(nv,NNI)
OV ==> OrderedVariableList(vl)
SE ==> Symbol
SUP ==> SparseUnivariatePolynomial(DPoly)

DPoly1 ==> DistributedMultivariatePolynomial(vl,Q)
DPoly ==> DistributedMultivariatePolynomial(vl,F)
NNI ==> NonNegativeInteger

Ideal == PolynomialIdeals(Q,Expon,OV,DPoly1)
FIdeal == PolynomialIdeals(F,Expon,OV,DPoly)
Fun0 == Union("zeroPrimDecomp","zeroRadComp")
GenPos == Record(changeval:List Z,genideal:FIdeal)

C == with

zeroDimPrime? : Ideal -> Boolean
++ zeroDimPrime?(I) tests if the ideal I is a 0-dimensional prime.
zeroDimPrimary? : Ideal -> Boolean
++ zeroDimPrimary?(I) tests if the ideal I is 0-dimensional primary.
prime?: Ideal -> Boolean
++ prime?(I) tests if the ideal I is prime.
radical : Ideal -> Ideal
++ radical(I) returns the radical of the ideal I.
primaryDecomp : Ideal -> List(Ideal)
++ primaryDecomp(I) returns a list of primary ideals such that their
++ intersection is the ideal I.
contract : (Ideal, List OV ) -> Ideal
++ contract(I,lvar) contracts the ideal I to the polynomial ring
++ \spad{F[lvar]}.

T == add

import MPolyCatRationalFunctionFactorizer(Expon,OV,Z,DPoly)
import GroebnerPackage(F,Expon,OV,DPoly)
import GroebnerPackage(Q,Expon,OV,DPoly1)
--- Local Functions ---

\[
\begin{align*}
genPosLastVar &: (\text{FIdeal}, \text{List OV}) \rightarrow \text{GenPos} \\
zeroPrimDecomp &: (\text{FIdeal}, \text{List OV}) \rightarrow \text{List(FIdeal)} \\
zeroRadComp &: (\text{FIdeal}, \text{List OV}) \rightarrow \text{FIdeal} \\
zerodimcase &: (\text{FIdeal}, \text{List OV}) \rightarrow \text{Boolean} \\
is0dimprimary &: (\text{FIdeal}, \text{List OV}) \rightarrow \text{Boolean} \\
backGenPos &: (\text{FIdeal}, \text{List Z}, \text{List OV}) \rightarrow \text{FIdeal} \\
reduceDim &: (\text{Fun0}, \text{FIdeal}, \text{List OV}) \rightarrow \text{List FIdeal} \\
findvar &: (\text{FIdeal}, \text{List OV}) \rightarrow \text{OV} \\
testPower &: (\text{SUP}, \text{OV}, \text{FIdeal}) \rightarrow \text{Boolean} \\
goodPower &: (\text{DPoly}, \text{FIdeal}) \rightarrow \text{Record(spol:DPoly, id:FIdeal)} \\
pushdown &: (\text{DPoly, FIdeal}) \rightarrow \text{Record(spol:DPoly, id:FIdeal)} \\
pushdterm &: (\text{DPoly, OV, Z}) \rightarrow \text{DPoly} \\
pushup &: (\text{DPoly, OV}) \rightarrow \text{DPoly} \\
pushuterm &: (\text{DPoly, SE, OV}) \rightarrow \text{DPoly} \\
pushucoef &: (\text{UP, OV}) \rightarrow \text{DPoly} \\
trueden &: (\text{P, SE}) \rightarrow \text{P} \\
rearrange &: (\text{List OV}) \rightarrow \text{List OV} \\
deleteunit &: (\text{List FIdeal}) \rightarrow \text{List FIdeal} \\
ismonic &: (\text{DPoly, OV}) \rightarrow \text{Boolean} \\
\end{align*}
\]

\[\text{MPCFQF} \Rightarrow \text{MPolyCatFunctions2(OV, Expon, Expon, Q, F, DPoly1, DPoly)}\]
\[\text{MPCFFQ} \Rightarrow \text{MPolyCatFunctions2(OV, Expon, Expon, F, Q, DPoly, DPoly1)}\]

\[\text{convertQF(a:Q)} : F = ((\text{numer a}):: F)/(((\text{denom a})::F)\]
\[\text{convertFQ(a:F)} : Q = (\text{ground numer a})/(\text{ground denom a})\]

\[\text{internalForm(I:Ideal)} : \text{FIdeal} ==\]
\[\text{Id:=generators I} \]
\[\text{nId:=[map(convertQF,poly)$MPCFQF for poly in Id]} \]
\[\text{groebner? I \Rightarrow groebnerIdeal nId} \]
\[\text{ideal nId}\]

\[\text{externalForm(I:FIdeal)} : \text{Ideal} ==\]
\[\text{Id:=generators I} \]
\[\text{nId:=[map(convertFQ,poly)$MPCFFQ for poly in Id]} \]
\[\text{groebner? I \Rightarrow groebnerIdeal nId} \]
\[\text{ideal nId}\]

\[\text{lvint:=[variable(xx)::OV for xx in vl]} \]
\[\text{nvint1:=(#lvint-1)::NNI}\]

\[\text{deleteunit(I1: List FIdeal)} : \text{List FIdeal} ==\]
\[\text{[I for I in I1 | _^ element?(I$DPoly, I)]}\]

\[\text{rearrange(vlist:List OV)} : \text{List OV} == \]
\[\text{vlist=[] \Rightarrow vlist} \]
\[\text{sort((z1,z2)+->z1>z2, setDifference(lvint, setDifference(lvint, vlist)))}\]
---- radical of a 0-dimensional ideal ----
zeroRadComp(I:FIdeal, trueList: List OV): FIdeal ==
  trueList=[] => I
  Id:=generators I
  x:=trueList.last
  #Id=1 =>
    f:=Id.first
    g:= (f exquo (gcd (f,differentiate(f,x))))::DPoly
groebnerIdeal([g])
y:=trueList.first
px:=DPoly=x::DPoly
py:=DPoly=y::DPoly
f:=Id.last
  g:= (f exquo (gcd (f,differentiate(f,x))))::DPoly
  Id:=groebner(cons(g,remove(f,Id)))
  lf:=Id.first
  pv:=DPoly:=0
  pw:=DPoly:=0
  while degree(lf,y)^=1 repeat
    val:=random()$Z rem 23
    pv:=px+val*py
    pw:=px-val*py
    Id:=groebner(cons((univariate(h,x)).pv for h in Id))
    lf:=Id.first
  ris:= generators(zeroRadComp(groebnerIdeal(Id.rest),trueList.rest))
  ris:=cons(lf,ris)
  if pv^=0 then
    ris:=[(univariate(h,x)).pw for h in ris]
groebnerIdeal(groebner ris)

---- find the power that stabilizes (I:s) ----
goodPower(s:DPoly, I:FIdeal): Record(spoly:DPoly, id:FIdeal) ==
  f:=DPoly:=s
  I:=groebner I
  J:=generators(JJ:= (saturate(I,s)))
  while _^ in?(ideal([f*g for g in J]), I) repeat f:=s*f
  [f, JJ]

---- is the ideal zerodimensional? ----
zerodimCase(J: FIdeal, trueList: List OV): Boolean ==
  element?(1, I, J) => true
  trueList=[] => true
  n:=#trueList
  Jd:=groebner generators J
  for x in trueList while Jd^=[] repeat
    f := Jd.first
    Jd:=Jd.rest
    if ((y:=mainVariable f) case "failed") or (y::OV ^=x)
or \(_\) (ismonic \((f,x)\)) then return false

while \(Jd=[\]\) and \((\text{mainVariable } Jd.\text{first})::\text{OV}=x\) repeat \(Jd:=Jd.\text{rest}\)

if \(Jd=[\]\) and \(\text{position}(x,\text{truelist})<\text{n}\) then return false

true

---- choose the variable for the reduction step ----

--- J groebnerner in gen pos ---

findvar(J:FIdeal,\text{truelist}:List \text{OV}) : \text{OV} ==

monicvar:List \text{OV} :=[\]

for \(f\) in \(\text{generators } J\) repeat

t:=\(f - \text{reductum } f\)

vt:List \text{OV} :=\text{variables } t

if \#vt=1 then \text{monicvar}:=setUnion(vt,\text{monicvar})

badvar:=setDifference(\text{truelist},\text{monicvar})

badvar.first

---- function for the "reduction step ----

reduceDim(flag:Fun0,J:FIdeal,\text{truelist}:List \text{OV}) : List(FIdeal) ==

\text{element?}(1,J) => [J]

zerodimcase(J,\text{truelist}) =>

(flag case "zeroPrimDecomp") => zeroPrimDecomp(J,\text{truelist})

(flag case "zeroRadComp") => [zeroRadComp(J,\text{truelist})]

x:OV:=\text{findvar}(J,\text{truelist})

Jnew:=[\text{pushdown}(f,x) for \(f\) in \(\text{generators } J\)]

Jc: List FIdeal :=[\]

Jc:=reduceDim(flag,\text{groebnerIdeal } Jnew,\text{remove}(x,\text{truelist}))

res1:=[\text{ideal}(\text{pushup}(f,x) for \(f\) in \text{generators } idp) for \text{idp} in \text{Jc}]\n
s:=\text{pushup}(_/\{\text{leadingCoefficient } f \text{ for } \text{f in Jnew}\}::\text{DPoly},x)

degree(s,x)=0 => res1

res1:=[\text{saturate}(\text{II},s) for \(\text{II}\) in \text{res1}]\n
\text{good}:=\text{goodPower}(s,J)

sideal := \text{groebnerIdeal}(\text{groebner}(\text{cons}(\text{good}.\text{spol},\text{generators } J)))\n
\text{in?}(\text{good}.\text{id}, \text{sideal}) => res1

sresult:=\text{reduceDim}(\text{flag},\text{sideal},\text{truelist})

for JJ in \text{sresult} repeat

if not(\text{in?}(\text{good}.\text{id},JJ)) then res1:=\text{cons}(JJ,res1)

res1

---- Primary Decomposition for 0-dimensional ideals ----

zeroPrimDecomp(I:FIdeal,\text{truelist}:List \text{OV}): List(FIdeal) ==

\text{truelist}=[\] \Rightarrow \text{list } I

\text{newJ}:=[\text{genPosLastVar}(I,\text{truelist})];lval:=\text{newJ}.\text{changeval};

J:=\text{groebner } \text{newJ}.\text{genideal}

x:=\text{trueList}.\text{last}

Jd:=\text{generators } J

g:=\text{Jd}.\text{last}

lfact:=factors factor(g)

ris:List FIdeal:=[\]

for ef in lfact repeat

g:DPoly:=(ef.factor)**(ef.exponent::\text{NNI})
J1 := groebnerIdeal(groebner cons(g,Jd))
if _^ (is0dimprimary (J1,truelist)) then
    return zeroPrimDecomp(I,truelist)
ris := cons(groebner backGenPos(J1,lval,truelist),ris)
ris

---- radical of an Ideal -----
radical(I:Ideal) : Ideal ==
J := groebner(internalForm I)
truelist := rearrange("setUnion"/[variables f for f in generators J])
truelist = [] => externalForm J
externalForm("intersect"/reduceDim("zeroRadComp",J,truelist))

-- the following functions are used to "push" x in the coefficient ring -

---- push x in the coefficient domain for a polynomial ----
pushdown(g:DPoly,x:OV) : DPoly ==
rf : DPoly := 0$DPoly
i := position(x,lvint)
while g^=0 repeat
    g1 := reductum g
    rf := rf + pushdterm(g-g1,x,i)
    g := g1
rf

---- push x in the coefficient domain for a term ----
pushdterm(t:DPoly,x:OV,i:Z):DPoly ==
n := degree(t,x)
xp := convert(x)@SE
cf := monomial(1,xp,n)$P :: F
newt := t exquo monomial(1,x,n)$DPoly
cf * newt :: DPoly

---- push back the variable ----
pushup(f:DPoly,x:OV) : DPoly ==
h := 1$P
rf : DPoly := 0$DPoly
g := f
xp := convert(x)@SE
while g^=0 repeat
    h := lcm(trueden(denom leadingCoefficient g,xp),h)
g := reductum g
f := (h :: F) * f
while f^=0 repeat
    g := reductum f
    rf := rf + pushuterm(f-g,xp,x)
f := g
rf
trueden(c:P,x:SE) : P ==
    degree(c,x) = 0 => 1
    c

---- push x back from the coefficient domain for a term ----
pushuterm(t:DPoly,xp:SE,x:OV):DPoly ==
    pushucoef((univariate(numer leadingCoefficient t,xp)$P), x)*
    monomial(inv((denom leadingCoefficient t)::F),degree t)$DPoly

pushucoef(c:UP,x:OV):DPoly ==
    c = 0 => 0
    monomial((leadingCoefficient c)::F::DPoly,x,degree c) +
    pushucoef(reductum c,x)

-- is the 0-dimensional ideal I primary ? --
----- internal function ----
is0dimprimary(J:FIdeal,truelist:List OV) : Boolean ==
    element?(1,J) => true
    Jd:=generators(groebner J)
    #(factors factor Jd.last)^=1 => return false
    i:=subtractIfCan(#truelist,1)
    (i case "failed") => return true
    JR:=(reverse Jd);JM:=groebnerIdeal([JR.first]);JP:List(DPoly):=[
    for f in JR.rest repeat
    if _^ ismonic(f,truelist.i) then
    if _^ inRadical?(f,JM) then return false
    JP:=cons(f,JP)
    else
    x:=truelist.i
    i:=(i-1)::NNI
    if _^ testPower(univariate(f,x),x,JM) then return false
    JM :=groebnerIdeal(append(cons(f,JP),generators JM))
    true

---- Functions for the General Position step ----

---- put the ideal in general position ----
genPosLastVar(J:FIdeal,truelist:List OV):GenPos ==
    x := last truelist ;lv1:List OV :=remove(x,truelist)
    ranvals:List(Z):=[(random()$Z rem 23) for vv in lv1]
    val:=._+/[rv*(vv::DPoly) for vv in lv1 for rv in ranvals]
    val:=val+(x::DPoly)
    [ranvals,groebnerIdeal(groebner([[univariate(p,x)].val
    for p in generators J]))]$GenPos

---- convert back the ideal ----
backGenPos(I:FIdeal,lval:List Z,truelist:List OV) : FIdeal ==
lval=[] => I
\begin{verbatim}
x := last truelist ; lv1:List OV:= remove(x, truelist)
val:=-(_+/[rv*(vv::DPoly) for vv in lv1 for rv in lval])
val:= val+(x::DPoly)
groebnerIdeal
 (groebner([univariate(p,x)).val for p in generators I ]))

ismonic(f:DPoly, x:OV) : Boolean ==
ground? leadingCoefficient(univariate(f,x))

---- test if f is power of a linear mod (rad J) ----
---- f is monic ----
testPower(uf:SUP, x:OV, J:FIdeal) : Boolean ==
df:=degree(uf)
trailp:DPoly := inv(df:Z ::F) *coefficient(uf,(df-1)::NNI)
linp:SUP:=(monomial(1$DPoly,1$NNI)$SUP +
 monomial(trailp,0$NNI)$SUP)**df
g:DPoly:=multivariate(uf-linp, x)
inRadical?(g,J)

---- Exported Functions ----
-- is the 0-dimensional ideal I prime ? --
zeroDimPrime?(I:Ideal) : Boolean ==
J:=groebner((genPosLastVar(internalForm I,lvint)).genideal)
element?(1,J) => true
n:NNI:=#vl;i:NNI:=1
Jd:=generators J
#Jd"n => false
for f in Jd repeat
  if _~ ismonic(f,lvint.i) then return false
  if i<n and (degree univariate(f,lvint.i))^=1 then return false
  i:=i+1
g:=Jd.n
#(lfact:=factors(factor g)) >1 => false
lfact.1.exponent =1

-- is the 0-dimensional ideal I primary ? --
zeroDimPrimary?(J:Ideal):Boolean ==
is0dimprimary(internalForm J,lvint)

---- Primary Decomposition of I ------
primaryDecomp(I:Ideal) : List(Ideal) ==
J:=groebner(internalForm I)
truelist:=rearrange("setUnion"/[variables f for f in generators J])
truelist=[] => [externalForm J]
[externalForm II for II in reduceDim("zeroPrimDecomp",J,truelist)]
\end{verbatim}
--- contract I to the ring with lvar variables ---

contract(I:Ideal,lvar: List OV) : Ideal ==
    Id:= generators(groebner I)
    empty?(Id) => I
    fullVars:= "setUnion"/[variables g for g in Id]
    fullVars = lvar => I
    n:= # lvar
    #fullVars < n => error "wrong vars"
    n=0 => I
    newVars:=
        append([vv for vv in fullVars| ^member?(vv,lvar)]$List(OV),lvar)
    subsVars := [monomial(1,vv,1)$DPoly1 for vv in newVars]
    lJ:= [eval(g,fullVars,subsVars) for g in Id]
    J := groebner(lJ)
    J=[1] => groebnerIdeal J
    J=[0] => groebnerIdeal empty()
    J:=[f for f in J| member?(mainVariable(f)::OV,newVars)]
    fullPol :=[monomial(1,vv,1)$DPoly1 for vv in fullVars]
    groebnerIdeal([eval(gg,newVars,fullPol) for gg in J])

———

— IDECOMP.dotabb ——

"IDECOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IDECOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"IDECOMP" -> "PFECAT"
"IDECOMP" -> "DIRPCAT"

———

package INCRMAPS IncrementingMaps

— IncrementingMaps.input —

)set break resume
)sys rm -f IncrementingMaps.output
)spool IncrementingMaps.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show IncrementingMaps
--R
--R IncrementingMaps(R: Join(Monoid,AbelianSemiGroup)) is a package constructor
--R Abbreviation for IncrementingMaps is INCRMAPS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INCRMAPS
--R
--R----------------------------------- Operations -----------------------------------
--R increment : () -> (R -> R)  incrementBy : R -> (R -> R)
--R
--E 1

)spool
)lisp (bye)

---

--- IncrementingMaps.help ---

=================================================================================
IncrementingMaps examples
=================================================================================

This package provides operations to create incrementing functions.

See Also:
  o )show IncrementingMaps

---

IncrementingMaps (INCRMAPS)

Exports:
  increment  incrementBy
package INCRMAPS IncrementingMaps

)abbrev package INCRMAPS IncrementingMaps
++ Date Last Updated: June 4, 1991
++ Description:
++ This package provides operations to create incrementing functions.

IncrementingMaps(R:Join(Monoid, AbelianSemiGroup)): with
   increment: () -> (R -> R)
      ++ increment() produces a function which adds \spad{1} to whatever
      ++ argument it is given. For example, if \{f := increment()\} then
      ++ \spad{f x} is \spad{x+1}.
   incrementBy: R -> (R -> R)
      ++ incrementBy(n) produces a function which adds \spad{n} to whatever
      ++ argument it is given. For example, if \{f := increment(n)\} then
      ++ \spad{f x} is \spad{x+n}.
   == add
      increment() == x +-> 1 + x
      incrementBy n == x +-> n + x

— INCRMAPS.dotabb —
"INCRMAPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INCRMAPS"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"INCRMAPS" -> "SGROUP"

package INFPROD0 InfiniteProductCharacteristicZero

— InfiniteProductCharacteristicZero.input —

)set break resume
)sys rm -f InfiniteProductCharacteristicZero.output
)spool InfiniteProductCharacteristicZero.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteProductCharacteristicZero
This package computes infinite products of univariate Taylor series over an integral domain of characteristic 0.

See Also:
  o )show InfiniteProductCharacteristicZero
Exports:
  evenInfiniteProduct infiniteProduct oddInfiniteProduct generalInfiniteProduct

— package INFPROD0 InfiniteProductCharacteristicZero —

)abbrev package INFPROD0 InfiniteProductCharacteristicZero
++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Description:
++ This package computes infinite products of univariate Taylor series
++ over an integral domain of characteristic 0.

InfiniteProductCharacteristicZero(Coef,UTS):_
Exports == Implementation where
  Coef : Join(IntegralDomain,CharacteristicZero)
  UTS : UnivariateTaylorSeriesCategory Coef
  I  ==> Integer

Exports ==> with

  infiniteProduct: UTS -> UTS
  ++ infiniteProduct(f(x)) computes \( \prod_{n=1,2,3,...} f(x^n) \).
  ++ The series \( f(x) \) should have constant coefficient 1.
  evenInfiniteProduct: UTS -> UTS
  ++ evenInfiniteProduct(f(x)) computes \( \prod_{n=2,4,6,...} f(x^n) \).
  ++ The series \( f(x) \) should have constant coefficient 1.
  oddInfiniteProduct: UTS -> UTS
  ++ oddInfiniteProduct(f(x)) computes \( \prod_{n=1,3,5,...} f(x^n) \).
  ++ The series \( f(x) \) should have constant coefficient 1.
  generalInfiniteProduct: (UTS,I,I) -> UTS
  ++ generalInfiniteProduct(f(x),a,d) computes
  ++ \( \prod_{n=a,a+d,a+2d,...} f(x^n) \).
  ++ The series \( f(x) \) should have constant coefficient 1.

Implementation ==> add

  import StreamInfiniteProduct Coef

  infiniteProduct x  == series infiniteProduct coefficients x
  evenInfiniteProduct x == series evenInfiniteProduct coefficients x
  oddInfiniteProduct x == series oddInfiniteProduct coefficients x

  generalInfiniteProduct(x,a,d) ==
    series generalInfiniteProduct(coefficients x,a,d)

________
package INPRODFF InfiniteProductFiniteField

--- InfiniteProductFiniteField.input ---

)set break resume
)sys rm -f InfiniteProductFiniteField.output
)spool InfiniteProductFiniteField.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show InfiniteProductFiniteField
--R
--R InfiniteProductFiniteField(K: Join(Field,Finite,ConvertibleTo(Integer)),UP: UnivariatePolynomialCategory(K),Coef: MonogenicAlgebra(K,UP),UTS: UnivariateTaylorSeriesCategory(Coef)) is a package constructor
--R Abbreviation for InfiniteProductFiniteField is INPRODFF
--R
--R------------------------------- Operations --------------------------------
--R evenInfiniteProduct : UTS -> UTS infiniteProduct : UTS -> UTS
--R oddInfiniteProduct : UTS -> UTS
generalInfiniteProduct : (UTS,Integer,Integer) -> UTS
--R
--E 1

)spool
)lisp (bye)

---

--- InfiniteProductFiniteField.help ---

====================================================================
InfiniteProductFiniteField examples
====================================================================
This package computes infinite products of univariate Taylor series over an arbitrary finite field.

See Also:
- )show InfiniteProductFiniteField
STF ==> StreamTranscendentalFunctions
STT ==> StreamTaylorSeriesOperations
ST2 ==> StreamFunctions2
SUP ==> SparseUnivariatePolynomial
Exports ==> with

infiniteProduct: UTS -> UTS
  ++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3...,f(x**n))}.
  ++ The series \spad{f(x)} should have constant coefficient 1.

evenInfiniteProduct: UTS -> UTS
  ++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
  ++ The series \spad{f(x)} should have constant coefficient 1.

oddInfiniteProduct: UTS -> UTS
  ++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
  ++ The series \spad{f(x)} should have constant coefficient 1.

generalInfiniteProduct: (UTS,I,I) -> UTS
  ++ generalInfiniteProduct(f(x),a,d) computes
  ++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
  ++ The series \spad{f(x)} should have constant coefficient 1.

Implementation ==> add

liftPoly: UP -> SUP RN
liftPoly poly ==
  -- lift coefficients of 'poly' to integers
  ans : SUP RN := 0
  while not zero? poly repeat
    coef := convert(leadingCoefficient poly)@I :: RN
    ans := ans + monomial(coef,degree poly)
    poly := reductum poly
  ans

reducePoly: SUP RN -> UP
reducePoly poly ==
  -- reduce coefficients of 'poly' to elements of K
  ans : UP := 0
  while not zero? poly repeat
    coef := numer(leadingCoefficient(poly)) :: K
    ans := ans + monomial(coef,degree poly)
    poly := reductum poly
  ans

POLY := liftPoly definingPolynomial()$Coef
ALG := SAE(RN,SUP RN,POLY)
infiniteProduct x ==
  stUP := map(lift,coefficients x)$ST2(Coef,UP)
  stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
  stALG := map(reduce,stSUP)$ST2(SUP RN,ALG)
stALG := exp(lambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
series map(reduce,stUP)$ST2(UP,Coeff)

evenInfiniteProduct x ==
stUP := map(lift,coefficients x)$ST2(Coeff,UP)
stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
stALG := exp(evenlambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
series map(reduce,stUP)$ST2(UP,Coeff)

oddInfiniteProduct x ==
stUP := map(lift,coefficients x)$ST2(Coeff,UP)
stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
stALG := exp(oddlambert(log(stALG)$STF(ALG))$STT(ALG))$STF(ALG)
stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
series map(reduce,stUP)$ST2(UP,Coeff)

generalInfiniteProduct(x,a,d) ==
stUP := map(lift,coefficients x)$ST2(Coeff,UP)
stSUP := map(liftPoly,stUP)$ST2(UP,SUP RN)
stALG := exp(generalLambert(log(stALG)$STF(ALG),a,d)$STT(ALG))$STF(ALG)
stSUP := map(lift,stALG)$ST2(ALG,SUP RN)
stUP := map(reducePoly,stSUP)$ST2(SUP RN,UP)
series map(reduce,stUP)$ST2(UP,Coeff)

package INPRODPF InfiniteProductPrimeField
--- InfiniteProductPrimeField.input ---

)set break resume
)sys rm -f InfiniteProductPrimeField.output
)spool InfiniteProductPrimeField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteProductPrimeField

--R InfiniteProductPrimeField(Coef: Join(Field,Finite,ConvertibleTo(Integer)),UTS: UnivariateTaylorSeriesCategory(Coef)) is a package constructor

--R Abbreviation for InfiniteProductPrimeField is INPRODPF

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INPRODPF

--R------------------------------- Operations --------------------------------

--R evenInfiniteProduct : UTS -> UTS  infiniteProduct : UTS -> UTS

--R oddInfiniteProduct : UTS -> UTS

--R generalInfiniteProduct : (UTS,Integer,Integer) -> UTS

--E 1

)spool
)lisp (bye)

---

--- InfiniteProductPrimeField.help ---

====================================================================
InfiniteProductPrimeField examples
====================================================================

This package computes infinite products of univariate Taylor series
over a field of prime order.

See Also:
o )show InfiniteProductPrimeField

---
InfiniteProductPrimeField (INPRODPF)

Exports:
   evenInfiniteProduct  generalInfiniteProduct  infiniteProduct  oddInfiniteProduct

package INPRODPF InfiniteProductPrimeField

)abbrev package INPRODPF InfiniteProductPrimeField
++ Author: Clifton J. Williamson
++ Date Created: 22 February 1990
++ Date Last Updated: 23 February 1990
++ Description:
++ This package computes infinite products of univariate Taylor series
++ over a field of prime order.

InfiniteProductPrimeField(Coef,UTS): Exports == Implementation where
   Coef : Join(Field,Finite,ConvertibleTo Integer)
   UTS : UnivariateTaylorSeriesCategory Coef
   I  ==> Integer
   ST ==> Stream

Exports ==>

infiniteProduct: UTS -> UTS
   ++ infiniteProduct(f(x)) computes \spad{product(n=1,2,3,...,f(x**n))}.
   ++ The series \spad{f(x)} should have constant coefficient 1.

   evenInfiniteProduct: UTS -> UTS
   ++ evenInfiniteProduct(f(x)) computes \spad{product(n=2,4,6...,f(x**n))}.
   ++ The series \spad{f(x)} should have constant coefficient 1.

   oddInfiniteProduct: UTS -> UTS
   ++ oddInfiniteProduct(f(x)) computes \spad{product(n=1,3,5...,f(x**n))}.
   ++ The series \spad{f(x)} should have constant coefficient 1.

   generalInfiniteProduct: (UTS,I,I) -> UTS
   ++ generalInfiniteProduct(f(x),a,d) computes
   ++ \spad{product(n=a,a+d,a+2*d,...,f(x**n))}.
   ++ The series \spad{f(x)} should have constant coefficient 1.
import StreamInfiniteProduct Integer

applyOverZ:(ST I -> ST I, ST Coef) -> ST Coef
applyOverZ(f, st) ==
  stZ := map(z1 +-> convert(z1)@Integer, st)$StreamFunctions2(Coef, I)
  map(z1 +-> z1 :: Coef, f stZ)$StreamFunctions2(I, Coef)

infiniteProduct x ==
  series applyOverZ(infiniteProduct, coefficients x)
evenInfiniteProduct x ==
  series applyOverZ(evenInfiniteProduct, coefficients x)
oddInfiniteProduct x ==
  series applyOverZ(oddInfiniteProduct, coefficients x)
generalInfiniteProduct(x, a, d) ==
  series
    applyOverZ(
      (z1:ST(I)):ST(I) +-> generalInfiniteProduct(z1, a, d), coefficients x)
--R InfiniteTupleFunctions2(A: Type,B: Type) is a package constructor
--R Abbreviation for InfiniteTupleFunctions2 is ITFUN2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ITFUN2
--R
--R------------------------------- Operations --------------------------------
--R map : ((A -> B),InfiniteTuple(A)) -> InfiniteTuple(B)
--R
--E 1

)spool
)lisp (bye)

---

--- InfiniteTupleFunctions2.help ---

====================================================================
InfiniteTupleFunctions2 examples
====================================================================

Functions defined on streams with entries in two sets.

See Also:
  o )show InfiniteTupleFunctions2

---

InfiniteTupleFunctions2 (ITFUN2)

Exports:
map
— package ITFUN2 InfiniteTupleFunctions2 —

)abbrev package ITFUN2 InfiniteTupleFunctions2
++ Description:
++ Functions defined on streams with entries in two sets.

InfiniteTupleFunctions2(A:Type,B:Type): Exports == Implementation where
    IT  ==> InfiniteTuple

Exports ==> with
    map: ((A -> B),IT A) -> IT B
    ++ \spad{map(f,[x0,x1,x2,...])} returns \spad{[f(x0),f(x1),f(x2),..]}. 

Implementation ==> add

    map(f,x) ==
    map(f,x pretend Stream(A))$StreamFunctions2(A,B) pretend IT(B)

— ITFUN2.dotabb —

"ITFUN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ITFUN2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ITFUN2" => "TYPE"

package ITFUN3 InfiniteTupleFunctions3

— InfiniteTupleFunctions3.input —

)set break resume
)sys rm -f InfiniteTupleFunctions3.output
)spool InfiniteTupleFunctions3.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InfiniteTupleFunctions3
--R
--R InfiniteTupleFunctions3(A: Type,B: Type,C: Type) is a package constructor
--- R Abbreviation for InfiniteTupleFunctions3 is ITFUN3
--- R This constructor is exposed in this frame.
--- R Issue )edit bookvol10.4.pamphlet to see algebra source code for ITFUN3
--- R
--- R------------------------------- Operations --------------------------------
--- R map : (((A,B) -> C),InfiniteTuple(A),InfiniteTuple(B)) -> InfiniteTuple(C)
--- R map : (((A,B) -> C),Stream(A),InfiniteTuple(B)) -> Stream(C)
--- R map : (((A,B) -> C),InfiniteTuple(A),Stream(B)) -> Stream(C)
--- R
--- E 1

)spool
)lisp (bye)

--- InfiniteTupleFunctions3.help ---

====================================================================
InfiniteTupleFunctions3 examples
====================================================================

Functions defined on streams with entries in two sets.

See Also:
o )show InfiniteTupleFunctions3

InfiniteTupleFunctions3 (ITFUN3)

Exports:
map
package ITFUN3 InfiniteTupleFunctions3

)abbrev package ITFUN3 InfiniteTupleFunctions3
++ Description:
++ Functions defined on streams with entries in two sets.

InfiniteTupleFunctions3(A:Type, B:Type,C:Type): Exports
  == Implementation where
  IT  ==> InfiniteTuple
  ST  ==> Stream
  SF3  ==> StreamFunctions3(A,B,C)
  FUN  ==> ((A,B)->C)
  Exports ==> with
    map: (((A,B)->C), IT A, IT B) -> IT C
      ++ undocumented
    map: (((A,B)->C), ST A, IT B) -> ST C
      ++ undocumented
    map: (((A,B)->C), IT A, ST B) -> ST C
      ++ undocumented

Implementation ==> add

map(f:FUN, s1:IT A, s2:IT B):IT C ==
  map(f, s1 pretend Stream(A), s2 pretend Stream(B))$SF3 pretend IT(C)
map(f:FUN, s1:ST A, s2:IT B):ST C ==
  map(f, s1, s2 pretend Stream(B))$SF3
map(f:FUN, s1:IT A, s2:ST B):ST C ==
  map(f, s1 pretend Stream(A), s2)$SF3

———

— ITFUN3.dotabb —

"ITFUN3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ITFUN3"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"ITFUN3" -> "TYPE"

———

package INFINITY Infinity

— Infinity.input —
"Infinity is a package constructor
Abbreviation for Infinity is INFINITY
This constructor is exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for INFINITY

Operations

infinity : () -> OnePointCompletion(Integer)
minusInfinity : () -> OrderedCompletion(Integer)
plusInfinity : () -> OrderedCompletion(Integer)

Infinity examples

Default infinity signatures for the interpreter;

See Also:

\( )\)show Infinity
**Infinity (INFINITY)**

Exports:
infinitiy minusInfinity plusInfinity

--- package INFINITY Infinity ---

```lisp
)abbrev package INFINITY Infinity
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Top-level infinity
++ Default infinity signatures for the interpreter;

Infinity(): with
    infinity : () -> OnePointCompletion Integer
        ++ infinity() returns infinity.
    plusInfinity : () -> OrderedCompletion Integer
        ++ plusInfinity() returns plusInfinity.
    minusInfinity: () -> OrderedCompletion Integer
        ++ minusInfinity() returns minusInfinity.
    == add
        infinity() == infinity()$OnePointCompletion(Integer)
        plusInfinity() == plusInfinity()$OrderedCompletion(Integer)
        minusInfinity() == minusInfinity()$OrderedCompletion(Integer)
```

--- INFINITY.dotabb ---

"INFINITY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFINITY"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"INFINITY" -> "PID"
package IALGFACT InnerAlgFactor

— InnerAlgFactor.input —

)set break resume
)sys rm -f InnerAlgFactor.output
)spool InnerAlgFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerAlgFactor

--R
--R InnerAlgFactor(F: Field,UP: UnivariatePolynomialCategory(F),AlExt: Join(Field,CharacteristicZero,MonogenicAlgebra(F,UP)),AlPol: UnivariatePolynomialCategory(AlExt)) is a package constructor
--R Abbreviation for InnerAlgFactor is IALGFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IALGFACT
--R
--R----------------------------------- Operations -----------------------------------
--R factor : (AlPol,(UP -> Factored(UP))) -> Factored(AlPol)
--R
--E 1

)spool
)lisp (bye

— InnerAlgFactor.help —

===============================================
InnerAlgFactor examples
===============================================

Factorisation in a simple algebraic extension. Factorization of
univariate polynomials with coefficients in an algebraic extension of
a field over which we can factor UP’s;

See Also:
o )show InnerAlgFactor
InnerAlgFactor (IALGFACT)

Exports:
factor

--- package IALGFACT InnerAlgFactor ---

)abbrev package IALGFACT InnerAlgFactor
++ Author: Patrizia Gianni
++ Date Last Updated: 20 Jul 1988
++ Description:
++ Factorisation in a simple algebraic extension
++ Factorization of univariate polynomials with coefficients in an
++ algebraic extension of a field over which we can factor UP's;

InnerAlgFactor(F, UP, AlExt, AlPol): Exports == Implementation where
  F : Field
  UP : UnivariatePolynomialCategory F
  AlPol: UnivariatePolynomialCategory AlExt
  AlExt : Join(Field, CharacteristicZero, MonogenicAlgebra(F,UP))
  NUP ===> SparseUnivariatePolynomial UP
  N ===> NonNegativeInteger
  Z ===> Integer
  FR ===> Factored UP
  UPCF2 ===> UnivariatePolynomialCategoryFunctions2

Exports ==> with
  factor: (AlPol, UP -> FR) -> Factored AlPol
    ++ factor(p, f) returns a prime factorisation of p;
    ++ f is a factorisation map for elements of UP;
Implementation $$\Rightarrow$$ add

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pnorm</code></td>
<td><code>A1Pol -&gt; UP</code></td>
</tr>
<tr>
<td><code>convert</code></td>
<td><code>A1Pol -&gt; NUP</code></td>
</tr>
<tr>
<td><code>change</code></td>
<td><code>UP -&gt; A1Pol</code></td>
</tr>
<tr>
<td><code>perturbfactor : (A1Pol, Z, UP -&gt; FR) -&gt; List A1Pol</code></td>
<td></td>
</tr>
<tr>
<td><code>irrfactor : (A1Pol, Z, UP -&gt; FR) -&gt; List A1Pol</code></td>
<td></td>
</tr>
</tbody>
</table>

```plaintext
perturbfactor(f, k, fact) ==
  pol := monomial(1$A1Ext,1) -
    monomial(reduce monomial(k::F,1)$UP,0)
  newf := elt(f, pol)
  lsols := irrfactor(newf, k, fact)
  pol := monomial(1, 1) +
    monomial(reduce monomial(k::F,1)$UP,0)
  [elt(pp, pol) for pp in lsols]

--- factorize the square-free parts of f ---
irrfactor(f, k, fact) ==
  degree(f) =\$N 1 => [f]
  newf := f
  nn := pnorm f
  --newval:RN:=1
  --pert:=false
  --if ^ SqFr? nn then
  --  pert:=true
  --  newterm:=perturb(f)
  --  newf:=newterm.ppol
  --  newval:=newterm.pval
  --  nn:=newterm.nnorm
  listfact := factors fact nn
  #listfact =\$N 1 =>
    first(listfact).exponent =\$Z 1 => [f]
    perturbfactor(f, k + 1, fact)
  listerterm(List(A1Pol) := []
  for pelt in listfact repeat
    g := gcd(change(pelt.factor), newf)
    newf := (newf exquo g)::A1Pol
    listerterm :=
      pelt.exponent =\$Z 1 => cons(g, listerterm)
      append(perturbfactor(g, k + 1, fact), listerterm)
    listerterm

factor(f, fact) ==
  sqf := squareFree f
  unit(sqf) * _* /[primeFactor(pol, sqterm.exponent)
    for pol in irrfactor(sqterm.factor, 0, fact)]
    for sqterm in factors sqf]

  p := definingPolynomial()$A1Ext```
newp := map(x +-> x::UP, p)$UPCF2(F, UP, UP, NUP)

pnorm q == resultant(convrt q, newp)
change q == map(coerce, q)$UPCF2(F,UP,A1Ext,A1Pol)

convrt q ==
  swap(map(lift, q)$UPCF2(A1Ext, A1Pol,
    UP, NUP))$CommuteUnivariatePolynomialCategory(F, UP, NUP)

package ICDEN InnerCommonDenominator

— IALGFACT.dotabb —
"IALGFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IALGFACT"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"IALGFACT" -> "MONOGEN"

package ICDEN InnerCommonDenominator

— InnerCommonDenominator.input —

)set break resume
)sys rm -f InnerCommonDenominator.output
)spool InnerCommonDenominator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerCommonDenominator
--R
--R InnerCommonDenominator(R: IntegralDomain,Q: QuotientFieldCategory(R),A: FiniteLinearAggregate(R),B: FiniteLinearAggregate(Q)) is a package constructor
--R Abbreviation for InnerCommonDenominator is ICDEN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ICDEN
--R
--R----------------------------- Operations -----------------------------
--R clearDenominator : B -> A
commonDenominator : B -> R
--R splitDenominator : B -> Record(num: A,den: R)
--R
--E 1
InnerCommonDenominator examples
====================================================================

InnerCommonDenominator provides functions to compute the common
denominator of a finite linear aggregate of elements of the quotient
field of an integral domain.

See Also:
o )show InnerCommonDenominator

---

InnerCommonDenominator (ICDEN)

Exports:
clearDenominator commonDenominator splitDenominator

---

)abbrev package ICDEN InnerCommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Nov 1989
++ Description:
InnerCommonDenominator provides functions to compute the common denominator of a finite linear aggregate of elements of the quotient field of an integral domain.

InnerCommonDenominator(R, Q, A, B): Exports == Implementation where
  R: IntegralDomain
  Q: QuotientFieldCategory R
  A: FiniteLinearAggregate R
  B: FiniteLinearAggregate Q

Exports ==> with
  commonDenominator: B -> R
  clearDenominator : B -> A
  splitDenominator : B -> Record(num: A, den: R)

Implementation ==> add
  import FiniteLinearAggregateFunctions2(Q, B, R, A)

  clearDenominator l ==
    d := commonDenominator l
    map(x +-> numer(d*x), l)

  splitDenominator l ==
    d := commonDenominator l
    [map(x +-> numer(d*x), l), d]

  if R has GcdDomain then
    commonDenominator l == reduce(lcm, map(denom, l), 1)
  else
    commonDenominator l == reduce("*", map(denom, l), 1)
package IMATLIN InnerMatrixLinearAlgebraFunctions

— InnerMatrixLinearAlgebraFunctions.input —

)set break resume
)sys rm -f InnerMatrixLinearAlgebraFunctions.output
)spool InnerMatrixLinearAlgebraFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerMatrixLinearAlgebraFunctions
--R
--R InnerMatrixLinearAlgebraFunctions(R: Field,Row: FiniteLinearAggregate(R),Col: FiniteLinearAggregate(R),M: MatrixCategory(R,Row,Col)) is a package constructor
--R Abbreviation for InnerMatrixLinearAlgebraFunctions is IMATLIN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IMATLIN
--R
--R------------------------------- Operations --------------------------------
--R determinant : M -> R generalizedInverse : M -> M
--R inverse : M -> Union(M,"failed") nullity : M -> NonNegativeInteger
--R rank : M -> NonNegativeInteger rowEchelon : M -> M
--R nullSpace : M -> List(Col) if Col has shallowlyMutable
--R
--E 1

)spool
)lisp (bye)

— InnerMatrixLinearAlgebraFunctions.help —

===============================================
InnerMatrixLinearAlgebraFunctions examples
===============================================

InnerMatrixLinearAlgebraFunctions is an internal package which provides standard linear algebra functions on domains in MatrixCategory.

See Also:
o )show InnerMatrixLinearAlgebraFunctions

———
Exports:

--- package IMATLIN InnerMatrixLinearAlgebraFunctions ---

)abbrev package IMATLIN InnerMatrixLinearAlgebraFunctions
++ Author: Clifton J. Williamson, P.Gianni
++ Date Created: 13 November 1989
++ Date Last Updated: September 1993
++ Description:
++ \spadtype{InnerMatrixLinearAlgebraFunctions} is an internal package
++ which provides standard linear algebra functions on domains in
++ \spad{MatrixCategory}

InnerMatrixLinearAlgebraFunctions(R,Row,Col,M):_
    Exports == Implementation where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R,Row,Col)
I ==> Integer

Exports ==> with
rowEchelon: M -> M
    ++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.
rank: M -> NonNegativeInteger
    ++ \spad{rank(m)} returns the rank of the matrix m.
nullity: M -> NonNegativeInteger

---
nullity(m) returns the nullity of the matrix m. This is the dimension of the null space of the matrix m.

if Col has shallowlyMutable then
  nullSpace: M -> List Col
  nullSpace(m) returns a basis for the null space of the matrix m.

determinant: M -> R
  determinant(m) returns the determinant of the matrix m.
  an error message is returned if the matrix is not square.

generalizedInverse: M -> M
  generalizedInverse(m) returns the generalized (Moore--Penrose) inverse of the matrix m, i.e. the matrix h such that m*h*m = h, h*m*h = m, m*h and h*m are both symmetric matrices.

inverse: M -> Union(M,"failed")
  inverse(m) returns the inverse of the matrix m.
  If the matrix is not invertible, "failed" is returned.
  Error: if the matrix is not square.

Implementation => add

rowAllZeroes?: (M,I) -> Boolean
rowAllZeroes?(x,i) ==
  -- determines if the ith row of x consists only of zeroes
  -- internal function: no check on index i
  for j in minColIndex(x)..maxColIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

colAllZeroes?: (M,I) -> Boolean
colAllZeroes?(x,j) ==
  -- determines if the ith column of x consists only of zeroes
  -- internal function: no check on index j
  for i in minRowIndex(x)..maxRowIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

rowEchelon y ==
  -- row echelon form via Gaussian elimination
  x := copy y
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  i := minR
  n: I := minR - 1
  for j in minC..maxC repeat
    i > maxR => return x
    n := minR - 1
    -- n = smallest k such that k >= i and qelt(x,k,j) ^= 0
    for k in i..maxR repeat
      if qelt(x,k,j) ^= 0 then leave (n := k)
    n = minR - 1 => "no non-zeroes"
-- put nth row in ith position
if i ^= n then swapRows!(x,i,n)
-- divide ith row by its first non-zero entry
b := inv qelt(x,i,j)
qsetelt_!(x,i,j,1)
for k in (j+1) .. maxC repeat qsetelt_!(x,i,k, b * qelt(x,i,k))
-- perform row operations so that jth column has only one 1
for k in minR .. maxR repeat
  if k ^= i and qelt(x,k,j) ^= 0 then
    for k1 in (j+1) .. maxC repeat
      qsetelt_!(x,k,k1, qelt(x,k,k1) - qelt(x,k,j) * qelt(x,i,k1))
      qsetelt_!(x,k,j,0)
  -- increment i
  i := i + 1
x

rank x ==
y :=
  (rk := nrows x) > (rh := ncols x) =>
  rk := rh
  transpose x
  copy x
y := rowEchelon y; i := maxRowIndex y
while rk > 0 and rowAllZeroes?(y,i) repeat
  i := i - 1
  rk := (rk - 1) :: NonNegativeInteger
rk :: NonNegativeInteger

nullity x == (ncols x - rank x) :: NonNegativeInteger

if Col has shallowlyMutable then

nullSpace y ==
x := rowEchelon y
minR := minRowIndex x; maxR := maxRowIndex x
minC := minColIndex x; maxC := maxColIndex x
nrow := nrows x; ncol := ncols x
basis : List Col := nil()
rk := nrow; row := maxR
  -- compute rank = # rows - # rows of all zeroes
while rk > 0 and rowAllZeroes?(x,row) repeat
  rk := (rk - 1) :: NonNegativeInteger
  row := (row - 1) :: NonNegativeInteger
  -- if maximal rank, return zero vector
  ncol <= nrow and rk = ncol => [new(ncol,0)]
  -- if rank = 0, return standard basis vectors
  rk = 0 =>
    for j in minC .. maxC repeat
      w : Col := new(ncol,0)
      qsetelt_!(w,j,1)
basis := cons(w,basis)
basis
-- v contains information about initial 1's in the rows of x
-- if the ith row has an initial 1 in the jth column, then
-- v.j = i; v.j = minR - 1, otherwise
v : IndexedOneDimensionalArray(I,minC) := new(ncol,minR - 1)
for i in minR..(minR + rk - 1) repeat
  for j in minC.. while qelt(x,i,j) = 0 repeat j
    qsetelt_!(v,j,i)
  j := maxC; l := minR + ncol - 1
while j >= minC repeat
  w : Col := new(ncol,0)
  -- if there is no row with an initial 1 in the jth column,
  -- create a basis vector with a 1 in the jth row
  if qelt(v,j) = minR - 1 then
    colAllZeroes?(x,j) =>
      qsetelt_!(w,l,1)
  basis := cons(w,basis)
  for k in minC..(j-1) for ll in minR..(l-1) repeat
    if qelt(v,k) ^= minR - 1 then
      qsetelt_!(w,ll,-qelt(x,qelt(v,k),j))
    qsetelt_!(w,l,1)
  basis := cons(w,basis)
  j := j - 1; l := l - 1

basis
determinant y ==
  (ndim := nrows y) ^= (ncols y) =>
    error "determinant: matrix must be square"
  -- Gaussian Elimination
  ndim = 1 => qelt(y,minRowIndex y,minColIndex y)
  x := copy y
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  ans := 1
  for i in minR..(maxR - 1) for j in minC..(maxC - 1) repeat
    if qelt(x,i,j) = 0 then
      rown := minR - 1
      for k in (i+1)..maxR repeat
        qelt(x,k,j) ^= 0 => leave (rown := k)
      if rown = minR - 1 then return 0
      swapRows_!(x,i,rown); ans := -ans
    ans := qelt(x,i,j) * ans; b := -inv qelt(x,i,j)
    for l in (j+1)..maxC repeat qsetelt_!(x,i,l,b * qelt(x,i,l))
    for k in (i+1)..maxR repeat
      if (b := qelt(x,k,j)) ^= 0 then
        for l in (j+1)..maxC repeat
          qsetelt_!(x,k,l,qelt(x,k,l) + b * qelt(x,i,l))
    qelt(x,maxR,maxC) * ans
generalizedInverse(x) ==
SUP:=SparseUnivariatePolynomial R
FSUP := Fraction SUP
VFSUP := Vector FSUP
MATCAT2 := MatrixCategoryFunctions2(R, Row, Col, M,
FSUP, VFSUP, VFSUP, Matrix FSUP)
MATCAT22 := MatrixCategoryFunctions2(FSUP, VFSUP, VFSUP, Matrix FSUP,
R, Row, Col, M)
y:= map((r1:R):FSUP +-> coerce(coerce(r1)$SUP)$(Fraction SUP),x)$MATCAT2
ty:=transpose y
yy:=ty*y
nc:=ncols yy
var:=monomial(1,1)$SUP ::(Fraction SUP)
yy:=inverse(yy+scalarMatrix(ncols yy, var))::Matrix(FSUP)*ty
map((z1:FSUP):R +-> elt(z1,0),yy)$MATCAT22

inverse x ==
(ndim := nrows x) ^= (ncols x) =>
   error "inverse: matrix must be square"
ndim = 2 =>
   ans2 : M := zero(ndim, ndim)
   zero?(det := x(1,1)*x(2,2)-x(1,2)*x(2,1)) => "failed"
   detinv := inv det
   ans2(1,1) := x(2,2)*detinv
   ans2(1,2) := -x(1,2)*detinv
   ans2(2,1) := -x(2,1)*detinv
   ans2(2,2) := x(1,1)*detinv
   ans2
   AB : M := zero(ndim,ndim + ndim)
   minR := minRowIndex x; maxR := maxRowIndex x
   minC := minColIndex x; maxC := maxColIndex x
   kmin := minRowIndex AB; kmax := kmin + ndim - 1
   lmin := minColIndex AB; lmax := lmin + ndim - 1
   for i in minR..maxR for k in kmin..kmax repeat
      for j in minC..maxC for l in lmin..lmax repeat
         qsetelt_!(AB,k,l,qelt(x,i,j))
         qsetelt_!(AB,k,lmin + ndim + k - kmin,1)
   AB := rowEchelon AB
   elt(AB,kmax,lmax) = 0 => "failed"
   subMatrix(AB,kmin,kmax,lmin + ndim,lmax + ndim)
package IMATQF InnerMatrixQuotientFieldFunctions

---

InnerMatrixQuotientFieldFunctions examples

InnerMatrixQuotientFieldFunctions provides functions on matrices over an integral domain which involve the quotient field of that integral domain. The functions rowEchelon and inverse return matrices with entries in the quotient field.

See Also:
InnerMatrixQuotientFieldFunctions (IMATQF)

Exports:
  inverse   rowEchelon   nullSpace

— package IMATQF InnerMatrixQuotientFieldFunctions —

)abbrev package IMATQF InnerMatrixQuotientFieldFunctions
++ Author: Clifton J. Williamson
++ Date Created: 22 November 1989
++ Date Last Updated: 22 November 1989
++ Description:
++ \spadtype{InnerMatrixQuotientFieldFunctions} provides functions on matrices
++ over an integral domain which involve the quotient field of that integral
++ domain. The functions rowEchelon and inverse return matrices with
++ entries in the quotient field.

InnerMatrixQuotientFieldFunctions(R,Row,Col,M,QF,Row2,Col2,M2):_
  Exports == Implementation where
  R : IntegralDomain
  Row : FiniteLinearAggregate R
  Co1 : FiniteLinearAggregate R
  M  : MatrixCategory(R,Row,Col)
  QF : QuotientFieldCategory R
  Row2 : FiniteLinearAggregate QF
  Co12 : FiniteLinearAggregate QF
  M2 : MatrixCategory(QF,Row2,Co12)
  IMATLIN ==> InnerMatrixLinearAlgebraFunctions(QF,Row2,Co12,M2)
  MATCAT2 ==> MatrixCategoryFunctions2(R,Row,Col,M,QF,Row2,Co12,M2)
CDEN ==> InnerCommonDenominator(R,QF,Col,Col2)

Exports ==> with
   rowEchelon: M -> M2
      +\spad{rowEchelon(m)} returns the row echelon form of the matrix \( m \).
      +the result will have entries in the quotient field.
   inverse: M -> Union(M2,"failed")
      +\spad{inverse(m)} returns the inverse of the matrix \( m \).
      +# If the matrix is not invertible, "failed" is returned.
      +# Error: if the matrix is not square.
      +# Note that the result will have entries in the quotient field.
if Col2 has shallowlyMutable then
   nullSpace : M -> List Col
      +\spad{nullSpace(m)} returns a basis for the null space of the
      +matrix \( m \).
Implementation ==> add

qfMat: M -> M2
qfMat m == map((r1:R):QF +-> r1::QF,m)$MATCAT2

rowEchelon m == rowEchelon(qfMat m)$IMATLIN
inverse m ==
   (inv := inverse(qfMat m)$IMATLIN) case "failed" => "failed"
   inv :: M2

if Col2 has shallowlyMutable then
   nullSpace m ==
      [clearDenominator(v)$CDEN for v in nullSpace(qfMat m)$IMATLIN]

---

--- IMATQF.dotabb ---

"IMATQF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IMATQF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"IMATQF" -> "PFECAT"

---

package INMODGCD InnerModularGcd

--- InnerModularGcd.input ---

)set break resume
--S 1 of 1
)show InnerModularGcd
--R
--R InnerModularGcd(R: EuclideanDomain,BP: UnivariatePolynomialCategory(R),pMod: R,nextMod: (R,NonNegativeInteger) -> R)
--R Abbreviation for InnerModularGcd is INMODGCD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INMODGCD
--R
--R------------------------------- Operations --------------------------------
--R modularGcd : List(BP) -> BP
--R modularGcdPrimitive : List(BP) -> BP
--R reduction : (BP,R) -> BP
--R
--E 1

)spool
)lisp (bye)

— InnerModularGcd.help —

====================================================================
InnerModularGcd examples
====================================================================

This file contains the functions for modular gcd algorithm for
univariate polynomials with coefficients in a non-trivial euclidean
domain (i.e. not a field). The package parametrised by the
coefficient domain, the polynomial domain, a prime, and a function for
choosing the next prime

See Also:
o )show InnerModularGcd
CHAPTER 10. CHAPTER I

InnerModularGcd (INMODGCD)

Exports:

modularGcd modularGcdPrimitive reduction

--- package INMODGCD InnerModularGcd ---

)abbrev package INMODGCD InnerModularGcd
++ Author: J.H. Davenport and P. Gianni
++ Date Created: July 1990
++ Date Last Updated: November 1991
++ Description:
++ This file contains the functions for modular gcd algorithm
++ for univariate polynomials with coefficients in a
++ non-trivial euclidean domain (i.e. not a field).
++ The package parametrised by the coefficient domain,
++ the polynomial domain, a prime, and a function for choosing the next prime

InnerModularGcd(R,BP,pMod,nextMod):C == T

where

Z  ==> Integer
NNI ==> NonNegativeInteger
R  : EuclideanDomain
BP  : UnivariatePolynomialCategory(R)
pMod : R
nextMod : (R,NNI) -> R

C == with

modularGcdPrimitive : List BP -> BP
++ modularGcdPrimitive(f1,f2) computes the gcd of the two polynomials
++ f1 and f2 by modular methods.

modularGcd : List BP -> BP
++ modularGcd(listf) computes the gcd of the list of polynomials
++ listf by modular methods.

reduction : (BP,R) -> BP
++ reduction(f,p) reduces the coefficients of the polynomial f
++ modulo the prime p.

T == add

-- local functions --
height : BP -> NNI
mbound : (BP,BP) -> NNI
modGcdPrimitive : (BP,BP) -> BP
test : (BP,BP,BP) -> Boolean
merge : (R,R) -> Union(R,"failed")
modInverse : (R,R) -> R
exactquo : (BP,BP,R) -> Union(BP,"failed")
constNotZero : BP -> Boolean
constcase : (List NNI ,List BP ) -> BP
lincase : (List NNI ,List BP ) -> BP

if R has IntegerNumberSystem then
  reduction(u:BP,p:R):BP ==
    p = 0 => u
    map((r1:R):R +-> symmetricRemainder(r1,p),u)
else
  reduction(u:BP,p:R):BP ==
    p = 0 => u
    map((r1:R):R +-> r1 rem p,u)

FP:=EuclideanModularRing(R,BP,R,reduction,merge,exactquo)
zeroChar : Boolean := R has CharacteristicZero

-- exported functions --
modularGcdPrimitive(listf : List BP) :BP ==
  empty? listf => 0$BP
  g := first listf
  for f in rest listf | ^zero? f while degree g > 0 repeat
    g:=modGcdPrimitive(g,f)
  g

-- gcd for univariate polynomials
modularGcd(listf : List BP): BP ==
  listf:=remove!(0$BP,listf)
  empty? listf => 0$BP
  # listf = 1 => first listf
  minpol:=1$BP
  -- extract a monomial gcd
  mdeg:= "min"/[minimumDegree f for f in listf]
  if mdeg>0 then
    minpol1:= monomial(1,mdeg)
    listf:= [(f exquo minpol1)::BP for f in listf]
minpol := minpol * minpol
listdeg := [degree f for f in listf]
-- make the polynomials primitive
listCont := [content f for f in listf]
contgcd := gcd listCont
-- make the polynomials primitive
listf := [(f exquo cf)::BP for f in listf for cf in listCont]
minpol := contgcd * minpol
ans :=
-- one polynomial is constant
member?(1, listf) => 1
-- one polynomial is linear
member?(1, listdeg) => lincase(listdeg, listf)
modularGcdPrimitive listf
minpol * ans

-- local functions --

-- one polynomial is linear, remark that they are primitive
lincase(listdeg: List NNI, listf: List BP): BP ==
n := position(1, listdeg)
g := listf.n
for f in listf repeat
  if (f1 := f exquo g) case "failed" then return 1$BP
  g
-- test if d is the gcd
test(f: BP, g: BP, d: BP): Boolean ==
d0 := coefficient(d, 0)
coefficient(f, 0) exquo d0 case "failed" => false
coefficient(g, 0) exquo d0 case "failed" => false
f exquo d case "failed" => false
g exquo d case "failed" => false
true
-- gcd and cofactors for PRIMITIVE univariate polynomials
-- also assumes that constant terms are non zero
modGcdPrimitive(f: BP, g: BP): BP ==
  df := degree f
dg := degree g
dp := FP
lcf := leadingCoefficient f
lcg := leadingCoefficient g
testdeg := NNI
lcd := gcd(lcf, lcg)
prime := pMod
bound := mbound(f, g)
while zero? (lcd rem prime) repeat
  prime := nextMod(prime, bound)
  soFar := gcd(reduce(f, prime), reduce(g, prime))::BP
testdeg:=degree soFar
zero? testdeg => return 1$BP
ldp:FP:=
   ((lcdp:=leadingCoefficient(soFar::BP)) = 1) =>
       reduce(lcd::BP,prime)
   reduce((modInverse(lcdp,prime)*lcd)::BP,prime)
soFar:=reduce(ldp::BP *soFar,prime)::BP
soFarModulus:=prime
-- choose the prime
while true repeat
    prime := nextMod(prime,bound)
    lcd rem prime =0 => "next prime"
    fp:=reduce(f,prime)
    gp:=reduce(g,prime)
    dp:=gcd(fp,gp)
    dgp :=euclideanSize dp
if dgp =0 then return 1$BP
if dgp=dg and ~(f exquo g case "failed") then return g
if dgp=df and ~(g exquo f case "failed") then return f
if dgp > testdeg => "next prime"
ldp:FP:=
   ((lcdp:=leadingCoefficient(dp::BP)) = 1) =>
       reduce(lcd::BP,prime)
   reduce((modInverse(lcdp,prime)*lcd)::BP,prime)
dp:=ldp *dp
dgp=testdeg =>
correction:=reduce(dp::BP-soFar,prime)::BP
zero? correction =>
   ans:=reduce(lcd::BP*soFar,soFarModulus)::BP
   cont:=content ans
   ans:=(ans exquo cont)::BP
   test(f,g,ans) => return ans
   soFarModulus:=soFarModulus*prime
   correctionFactor:=modInverse(soFarModulus rem prime,prime)
   -- the initial rem is just for efficiency
   soFar:=soFar+soFarModulus*(correctionFactor*correction)
   soFarModulus:=soFarModulus*prime
   soFar:=reduce(soFar,soFarModulus)::BP
dgp<testdeg =>
   soFarModulus:=prime
   soFar:=dp::BP
   testdeg:=dgp
if ~zeroChar and euclideanSize(prime)>1 then
    result:=dp::BP
    test(f,g,result) => return result
    -- this is based on the assumption that the caller of this package,
    -- in non-zero characteristic, will use primes of the form
    -- x-alpha as long as possible, but, if these are exhausted,
    -- will switch to a prime of degree larger than the answer
    -- so the result can be used directly.
merge(p:R,q:R):Union(R,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

modInverse(c:R,p:R):R ==
  (extendedEuclidean(c,p,1)::Record(coef1:R,coef2:R)).coef1

exactQuo(u:BP,v:BP,p:R):Union(BP,"failed") ==
  invlcv:=modInverse(leadingCoefficient v,p)
  r:=monicDivide(u,reduction(invlcv*v,p))
  reduction(r.remainder,p) ^=0 => "failed"
  reduction(invlcv*r.quotient,p)

-- compute the height of a polynomial --
height(f:BP):NNI ==
  degf:=degree f
  "max"/[euclideanSize cc for cc in coefficients f]

-- compute the bound
mbound(f:BP,g:BP):NNI ==
  hf:=height f
  hg:=height g
  2*min(hf,hg)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
\section{package FOMOGCD ForModularGcd}

-- ForModularGcd(R,BP) : C == T
-- where
--  R : EuclideanDomain  -- characteristic 0
--  BP : UnivariatePolynomialCategory(R)
--
-- C == with
--  nextMod : (R,NNI) -> R
--
-- T == add
--  nextMod(val:R,bound:NNI) : R ==
--    ival:=val pretend Z
--    (nextPrime(ival)$IntegerPrimesPackage(Z))::R
--
-- ForTwoGcd(F) : C == T
-- where
--  F : Join(Finite,Field)
--  SUP == SparseUnivariatePolynomial
package INNMFACT InnerMultFact

-- R ==> SUP F
-- P ==> SUP R
-- UPCF2 ==> UnivariatePolynomialCategoryFunctions2
--
-- C == with
-- nextMod : (R,NNI) -> R
--
-- T == add
-- nextMod(val:R,bound:NNI) : R ==
-- ris:R:= nextItem(val) :: R
-- euclideanSize ris < 2 => ris
-- generateIrredPoly(
-- (bound+1)::PositiveInteger)$IrredPolyOverFiniteField(F)
--
--
-- ModularGcd(R,BP) == T
-- where
-- R : EuclideanDomain -- characteristic 0
-- BP : UnivariatePolynomialCategory(R)
-- T ==> InnerModularGcd(R,BP,67108859::R,nextMod$ForModularGcd(R,BP))
--
-- TwoGcd(F) : C == T
-- where
-- F : Join(Finite,Field)
-- SUP ==> SparseUnivariatePolynomial
-- R ==> SUP F
-- P ==> SUP R
--
-- T ==> InnerModularGcd(R,P,nextMod(monomial(1,1)$R)$ForTwoGcd(F),
-- nextMod$ForTwoGcd(F))
CHAPTER 10. CHAPTER I

---S 1 of 1
)show InnerMultFact
--R
--R InnerMultFact(DV: OrderedSet,E: OrderedAbelianMonoidSup,R: Join(EuclideanDomain,CharacteristicZero),P: PolynomialCategory(R,E,OV)) is a package constructor
--R Abbreviation for InnerMultFact is INNMFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INNMFACT
--R
--R-------------------------------------------------------------- Operations --------------------------------
--R factor : (P,(SparseUnivariatePolynomial(R) -> Factored(SparseUnivariatePolynomial(R)))) -> Factored(P)
--R factor : (SparseUnivariatePolynomial(P),(SparseUnivariatePolynomial(R) -> Factored(SparseUnivariatePolynomial(R)))) -> Factored(SparseUnivariatePolynomial(P))
--R
--E 1

)spool
)lisp (bye)

---

--- InnerMultFact.help ---

====================================================================
InnerMultFact examples
====================================================================

This is an inner package for factoring multivariate polynomials over various coefficient domains in characteristic 0. The univariate factor operation is passed as a parameter. Multivariate hensel lifting is used to lift the univariate factorization

See Also:
o )show InnerMultFact

---
InnerMultFact (INNMFACT)

Exports:

factor

— package INNMFACT InnerMultFact —

)abbrev package INNMFACT InnerMultFact
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Description:
++ This is an inner package for factoring multivariate polynomials
++ over various coefficient domains in characteristic 0.
++ The univariate factor operation is passed as a parameter.
++ Multivariate hensel lifting is used to lift the univariate
++ factorization

-- Both exposed functions call mFactor. This deals with issues such as
-- monomial factors, contents, square-freeness etc., then calls intfact.
-- This uses intChoose to find a "good" evaluation and factorise the
-- corresponding univariate, and then uses MultivariateLifting to find
-- the multivariate factors.

InnerMultFact(OV,E,R,P) : C == T

where

R : Join(EuclideanDomain, CharacteristicZero)
    -- with factor on R[x]
OV : OrderedSet
E : OrderedAbelianMonoidSup
P : PolynomialCategory(R,E,OV)
BP ==> SparseUnivariatePolynomial R
UFactor ==> (BP -> Factored BP)
Z ==> Integer
MParFact ==> Record(irr:P,pow:Z)
USP ==> SparseUnivariatePolynomial P
SUParFact ==> Record(irr:USP,pow:Z)
SUPFinalFact ==> Record(contp:R,factors:List SUParFact)
MFinalFact ==> Record(contp:R,factors:List MParFact)

-- contp = content,
-- factors = List of irreducible factors with exponent
L ==> List

C == with
  factor : (P,UFactor) -> Factored P
    ++ factor(p,ufact) factors the multivariate polynomial p
    ++ by specializing variables and calling the univariate
    ++ factorizer ufact.
factor : (USP,UFactor) -> Factored USP
  ++ factor(p,ufact) factors the multivariate polynomial p
  ++ by specializing variables and calling the univariate
  ++ factorizer ufact. p is represented as a univariate
  ++ polynomial with multivariate coefficients.

T == add

NNI ==> NonNegativeInteger

LeadFact ==> Record(polfac:L P,correct:R,corrfact:L BP)
ContPrim ==> Record(cont:P,prim:P)
ParFact ==> Record(irr:BP,pow:Z)
FinalFact ==> Record(contp:R,factors:L ParFact)
NewOrd ==> Record(npol:USP,nvar:L OV,newdeg:L NNI)
pmod:R := (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

import GenExEuclid(R,BP)
import MultivariateLifting(E,OV,R,P)
import FactoringUtilities(E,OV,R,P)
import LeadingCoefDetermination(OV,E,R,P)
Valuf ==> Record(inval:L L R,unvfact:L BP,lu:R,complead:L R)
UPCF2 ==> UnivariatePolynomialCategoryFunctions2

---- Local Functions ----
mFactor : (P,UFactor) -> MFinalFact
supFactor : (USP,UFactor) -> SUPFinalFact
mfconst : (USP,L OV,L NNI,UFactor) -> L USP
mfpol : (USP,L OV,L NNI,UFactor) -> L USP
monicMfpol: (USP,L OV,L NNI,UFactor) -> L USP
varChoose : (P,L OV,L NNI) -> NewOrd
simplify : (P,L OV,L NNI,UFactor) -> MFinalFact
intChoose : (USP,L OV,R,L P,L L R,UFactor) -> Union(Valuf,"failed")
intfact : (USP,L OV,L NNI,MFinalFact,L L R,UFactor) -> L USP
pretest : (P,NNI,L OV,L R) -> FinalFact
checkzero : (USP,BP) -> Boolean
localNorm : L BP -> Z
convertPUP(lfg:MFinalFact): SUPFinalFact ==
   [lfg.contp,[[lff.irr :: USP, lff.pow]$SUParFact
            for lff in lfg.factors]]$SUPFinalFact

-- intermediate routine if an SUP was passed in.
supFactor(um:USP, ufactor: UFactor) : SUPFinalFact ==
ground?(um) => convertPUP(mFactor(ground um,ufactor))
lvar:L OV := "setUnion"/[variables cf for cf in coefficients um]
empty? lvar => -- the polynomial is univariate
   umv := map(ground, um)$UPCF2(P, USP, R, BP)
   lfact := ufactor umv
   [retract unit lfact, [[map(coerce, ff.factor)$UPCF2(R, BP, P, USP),
                            ff.exponent] for ff in factors lfact]]$SUPFinalFact
lcont:P
lf:L USP
flead : SUPFinalFact:=[0,empty()]
factorlist:L SUParFact :=empty()

mdeg := minimumDegree um ---- is the Mindeg > 0? ----
if mdeg>0 then
   f1: USP := monomial(1, mdeg)
   um := (um exquo f1) :: USP
   factorlist := cons([[monomial(1,1), mdeg], factorlist]
   if degree um=0 then return
   lfg := convertPUP mFactor(ground um, ufactor)
      [lfg.contp, append(factorlist, lfg.factors)]
   uum := unitNormal um
   um := uum.canonical
   sqfacs := squareFree(um)$MultivariateSquareFree(E, OV, R, P)
lcont := ground(uum.unit * unit sqfacs)
    ---- Factorize the content ----
flead := convertPUP mFactor(lcont, ufactor)
factorlist := append(flead.factors, factorlist)

    ---- Make the polynomial square-free ----
sqqfact := factors sqfacs
    --- Factorize the primitive square-free terms ---
for fact in sqqfact repeat
   ffactor: USP := fact.factor
   ffexp := fact.exponent
   zero? degree ffactor =>
      lfg := mFactor(ground ffactor, ufactor)
      lcont := lfg.contp * lcont
      factorlist := append(factorlist, [[lff.irr :: USP, lff.pow * ffexp]$SUParFact
                                     for lff in lfg.factors])
   coefs := coefficients ffactor
   ldeg := ["max"/[degree(fc, xx) for fc in coefs] for xx in lvar]
   if :=
      ground?(leadingCoefficient ffactor) =>
      mfconst(ffactor, lvar, ldeg, ufactor)
mfpol(ffactor,lvar,ldeg,ufactor)
auxfl:=[lfp,ffexp]$SUParFact for lfp in lf]
factorlist:=append(factorlist,auxfl)
lcfacs := */[leadingCoefficient leadingCoefficient(f.irr)**((f.pow)::NNI)
for f in factorlist]
[(leadingCoefficient leadingCoefficient(um) exquo lcfacs)::R,
factorlist]$SUPFinalFact

factor(um:USP,ufactor:UFactor):Factored USP ==
flist := supFactor(um,ufactor)
(flist.contp):: P :: USP *
(*/[primeFactor(u.irr,u.pow) for u in flist.factors])

checkzero(u:USP,um:BP) : Boolean ==
u=0 => um =0
um = 0 => false
degree u = degree um => checkzero(reductum u, reductum um)
false

--- Choose the variable of less degree ---
varChoose(m:P,lvar:LOV,ldeg:NNI) : NewOrd ==
k:="min"/[d for d in ldeg]
k=degree(m,first lvar) =>
[univariate(m,first lvar),lvar,ldeg]$NewOrd
i:=position(k,ldeg)
x:OV:=lvar.i
ldeg:=cons(k,delete(ldeg,i))
lvar:=cons(x,delete(lvar,i))
[univariate(m,x),lvar,ldeg]$NewOrd

localNorm(lum: L BP): Z ==
R is AlgebraicNumber =>
"max"/[numberOfMonomials ff for ff in lum]
"max"/[+/[euclideanSize cc for i in 0..degree ff|
(cc:= coefficient(ff,i))^=0] for ff in lum]

--- Choose the integer to reduce to univariate case ---
intChoose(um:USP,lvar:LOV,clc:R,plist:L P,ltry:L L R,
ufactor:UFactor) : Union(Valuf,"failed") ==

-- declarations
degum:NNI := degree um
nvar1:=#1var
range:NNI:=5
unifact:L BP
ctf1 : R := 1
testp:Boolean :=
empty? plist => false
true
leadcomp,leadcomp1 : L R
leadcomp:=leadcomp1:=empty()
nfatt:NNI := degum+1
lffc:R:=1
lffc1:=lffc
newunifact : L BP:=empty()
leadtest:=true --- the lc test with polCase has to be performed
int:L R:=empty()

-- New sets of integers are chosen to reduce the multivariate problem to
-- a univariate one, until we find twice the
-- same (and minimal) number of "univariate" factors:
-- the set smaller in modulo is chosen.
-- Note that there is no guarantee that this is the truth:
-- merely the closest approximation we have found!

while true repeat
   testp and #ltry>10 => return "failed"
   lval := [ ran(range) for i in 1..nvar1]
   member?(lval,ltry) => range:=2*range
   ltry := cons(lval,ltry)
   leadcomp1:=[retract eval(pol,lvar,lval) for pol in plist]
   testp and or/[unit? epl for epl in leadcomp1] => range:=2*range
   newm:BP:=completeEval(um,lvar,lval)
   degum ^= degree newm or minimumDegree newm ^=0 => range:=2*range
   lffc1:=content newm
   newm:=(newm exquo lffc1)::BP
   testp and leadtest and ^ polCase(lffc1*clc,#plist,leadcomp1)
   => range:=2*range
   degree(gcd [newm,differentiate(newm)])^=0 => range:=2*range
   luniv:=ufactor(newm)
   lunivf:= factors luniv
   lffc1:R:=retract(unit luniv)@R * lffc1
   nf:= #lunivf
   nf=0 or nf>nfatt => "next values" --- pretest failed ---
   --- the univariate polynomial is irreducible ---
   if nf=1 then leave (unifact:=[newm])

-- the new integer give the same number of factors
nfatt = nf =>
-- if this is the first univariate factorization with polCase=true
-- or if the last factorization has smaller norm and satisfies
-- polCase
if leadtest or
   ((localNorm unifact > localNorm [ff.factor for ff in lunivf])
    and (^testp or polCase(lffc1*clc,#plist,leadcomp1))) then
   unifact:=[uf.factor for uf in lunivf]
   int:=lval
   lffc:=lffc1
if testp then leadcomp:=leadcomp1
leave "foundit"

-- the first univariate factorization, initialize
nfatt > degum =>
    unifact:=[uf.factor for uf in lunivf]
lffc:=lffc1
    if testp then leadcomp:=leadcomp1
    int:=lval
    leadtest := false
    nfatt := nf

nfatt>nf => -- for the previous values there were more factors
    if testp then leadtest:="polCase(lffc*clc,#plist,leadcomp)
    else leadtest:= false
    -- if polCase=true we can consider the univariate decomposition
    if `leadtest then
        unifact:=[uf.factor for uf in lunivf]
lffc:=lffc1
        if testp then leadcomp:=leadcomp1
        int:=lval
        nfatt := nf
    [cons(int,ltry),unifact,lffc,leadcomp]$Valuf

---- The polynomial has mindeg>0 ----
simplify(m:P,lvar:LOV,lmdeg:NNI,ufactor:UFactor):MFinalFact ==
    factorlist:LMParFact:=[]
poll1:P:= 1$P
    for x in lvar repeat
        i := lmdeg.(position(x,lvar))
i=0 => "next value"
poll1:=poll1*monomial(1$P,x,i)
factorlist:=cons([x::P,i]$MParFact,factorlist)
m := (m exquo poll1)::P
ground? m => [retract m,factorlist]$MFinalFact
flead:=mFactor(m,ufactor)
flead.factors:=append(factorlist,flead.factors)
flead

-- This is the key internal function
-- We now know that the polynomial is square-free etc.,
-- We use intChoose to find a set of integer values to reduce the
-- problem to univariate (and for efficiency, intChoose returns
-- the univariate factors).
-- In the case of a polynomial leading coefficient, we check that this
-- is consistent with leading coefficient determination (else try again)
-- We then lift the univariate factors to multivariate factors, and
-- return the result
intfact(um:USP,lvar: LOV,ldeg:NNI,tleadpol:MFinalFact,
ltry:L L R, ufactor: UFactor) : L USP ==

polcase:Boolean:=(not empty? tleadpol.factors)

vfchoo:Valuf:=
polcase =>
  leadpol:L P:=[ff irr for ff in tleadpol.factors]
  check:=intChoose(um, lvar, tleadpol.contp, leadpol, ltry, ufactor)
  check case "failed" => return monicMfpol(um, lvar, ldeg, ufactor)
  check::Valuf
  intChoose(um, lvar, 1, empty(), empty(), ufactor)::Valuf

unifact:List BP := vfchoo.unvfact

nfact:NNI := #unifact

nfact=1 => [um]
ltry:L L R := vfchoo.inval
lval:L R := first ltry
dd:=vfchoo.lu
leadval:L R := empty()
lpol:List P := empty()

if polcase then
  leadval := vfchoo.complead
  distf := distFact(vfchoo.lu, unifact, tleadpol, leadval, lvar, lval)
  distf case "failed" => return intfact(um, lvar, ldeg, tleadpol, ltry, ufactor)
  dist := distf :: LeadFact
  -- check the factorization of leading coefficient
  lpol := dist.polfac
  dd := dist.correct
  unifact := dist.corrfact
  if dd\^=1 then
  -- if polcase then lpol := [unitCanonical lp for lp in lpol]
  -- dd:=unitCanonical(dd)
    unifact := [dd * unif for unif in unifact]
    umd := unitNormal(dd).unit * ((dd**((nfact-1)::NNI)::P)*um
  else umd := um
    (ffin:=lifting(umd, lvar, unifact, lval, lpol, ldeg, pmod))
    case "failed" => intfact(um, lvar, ldeg, tleadpol, ltry, ufactor)
  factfin: L USP := ffin :: L USP
  if dd\^=1 then
    factfin:=[primitivePart ff for ff in factfin]
factfin

mfconst(um:USP, lvar:L OV, ldeg:L NNI, ufactor: UFactor): L USP ==

monicize(um:USP, c:P): USP ==
n:=degree(um)
\begin{verbatim}
ans:USP := monomial(1,n)
n:=(n-1)::NonNegativeInteger
prod:P:=1
while (um:=reductum(um)) ^= 0 repeat
  i := degree um
  lc := leadingCoefficient um
  prod := prod * c ** (n-(n:=i))::NonNegativeInteger
  ans := ans + monomial(prod*lc, i)
ans

unmonicize(m:USP,c:P):USP == primitivePart m(monomial(c,1))

--- m is square-free,primitive,lc is a polynomial ---
monicMfpol(um:USP,lvar:L OV,ldeg:L NNI,ufactor:UFactor):L USP ==
  l := leadingCoefficient um
  monpol := monicize(um,l)
  nldeg := degree(monpol,lvar)
  map((z1:USP):USP +-> unmonicize(z1,l),
      mconst(monpol,lvar,nldeg,ufactor))

mfpol(um:USP,lvar:L OV,ldeg:L NNI,ufactor:UFactor):L USP ==
  R has Field =>
  monicMfpol(um,lvar,ldeg,ufactor)
  tleadpol:=mFactor(leadingCoefficient um,ufactor)
  intfact(um,lvar,ldeg,tleadpol,[],ufactor)

mFactor(m:P,ufactor:UFactor) : MFinalFact ==
  ground?(m) => [retract(m),empty()]$MFinalFact
  lvar:L OV := variables m
  lcont:P
  lf:L USP
  flead : MFinalFact:=[0,empty()]$MFinalFact
  factorlist:L MParFact :=empty()
  lmdeg :=minimumDegree(m,lvar) ---- is the Mindeg > 0? ----
or/[n>0 for n in lmdeg] => simplify(m,lvar,lmdeg,ufactor)

  sqfacs := squareFree m
  lcont := unit sqfacs
   ---- Factorize the content ----
  if ground? lcont then flead.contp:=retract lcont
  else flead:=mFactor(lcont,ufactor)
  factorlist:=flead.factors
   ---- Make the polynomial square-free ----
  sqqfact:=factors sqfacs
\end{verbatim}
--- Factorize the primitive square-free terms ---
for fact in sqqfact repeat
  ffactor := fact.factor
  ffexp := fact.exponent
  lvar := variables ffactor
  x:OV := lvar.first
  ldeg := degree(ffactor, lvar)
  --- Is the polynomial linear in one of the variables? ---
  if member?(1, ldeg) =>
    x:OV := lvar.position(1, ldeg)
    lcont := gcd coefficients (univariate(ffactor, x))
    ffactor := (ffactor exquo lcont)::P
    factorlist := cons([ffactor, ffexp]$MParFact, factorlist)
  for lcterm in mFactor(lcont, ufactor).factors repeat
    factorlist := cons([lcterm.irr, lcterm.pow * ffexp], factorlist)
  varch := varChoose(ffactor, lvar, ldeg)
  um := varch.npol
  x := lvar.first
  ldeg := ldeg.rest
  lvar := lvar.rest
  if varch.nvar.first ^= x then
    lvar := varch.nvar
    x := lvar.first
    lvar := lvar.rest
  pc := gcd coefficients um
  if pc ^= 1 then
    um := (um exquo pc)::USP
    ffactor := multivariate(um, x)
    for lcterm in mFactor(pc, ufactor).factors repeat
      factorlist := cons([lcterm.irr, lcterm.pow * ffexp], factorlist)
  ldeg := degree(ffactor, lvar)
  um := unitCanonical um
  if ground?((leadingCoefficient um)) then
    iff := mconst(um, lvar, ldeg, ufactor)
  else iff := mpol(um, lvar, ldeg, ufactor)
  auxfl := [[unitCanonical multivariate(lfp, x), ffexp]$MParFact for lfp in lf]
  factorlist := append(factorlist, auxfl)
lcfacs := */[leadingCoefficient(f.irr)**((f.pow)::NNI) for f in factorlist]
[(leadingCoefficient(m) exquo lcfacs)::R, factorlist]$_MPFinalFact$

factor(m:P, ufactor: UFactor): Factored P ==
  flist := mFactor(m, ufactor)
  (flist.contp)::P *
  (*/[primeFactor(u.irr, u.pow) for u in flist.factors])
package INBFF InnerNormalBasisFieldFunctions

-- InnerNormalBasisFieldFunctions.input --

)set break resume
)sys rm -f InnerNormalBasisFieldFunctions.output
)spool InnerNormalBasisFieldFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerNormalBasisFieldFunctions

--R
--R InnerNormalBasisFieldFunctions(GF: FiniteFieldCategory) is a package constructor
--R Abbreviation for InnerNormalBasisFieldFunctions is INBFF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INBFF
--R
--R------------------------------- Operations --------------------------------
--R inv : Vector(GF) -> Vector(GF) normal? : Vector(GF) -> Boolean
--R ?? : (Vector(GF),Vector(GF)) -> Vector(GF)
--R ?? : (Vector(GF),Integer) -> Vector(GF)
--R ?? : (Vector(GF),Vector(GF)) -> Vector(GF)
--R basis : PositiveInteger -> Vector(Vector(GF))
--R dAndcExp : (Vector(GF),NonNegativeInteger,SingleInteger) -> Vector(GF)
--R expPot : (Vector(GF),Integer,SingleInteger) -> Vector(GF)
--R index : (PositiveInteger,PositiveInteger) -> Vector(GF)
--R lookup : Vector(GF) -> PositiveInteger
--R minimalPolynomial : Vector(GF) -> SparseUnivariatePolynomial(GF)
--R norm : (Vector(GF),PositiveInteger) -> Vector(GF)
--R normalElement : PositiveInteger -> Vector(GF)
--R pol : Vector(GF) -> SparseUnivariatePolynomial(GF)
--R qPot : (Vector(GF),Integer) -> Vector(GF)
--R random : PositiveInteger -> Vector(GF)
--R repSq : (Vector(GF),NonNegativeInteger) -> Vector(GF)
--R setFieldInfo : (Vector(List(Record(value: GF,index: SingleInteger)))),GF) -> Void
---R trace : (Vector(GF),PositiveInteger) -> Vector(GF)
---R xn : NonNegativeInteger -> SparseUnivariatePolynomial(GF)
---R
---E 1

)spool
)lisp (bye)

---

— InnerNormalBasisFieldFunctions.help —

====================================================================
InnerNormalBasisFieldFunctions examples
====================================================================

InnerNormalBasisFieldFunctions(GF) (unexposed): This package has
functions used by every normal basis finite field extension domain.

See Also:
o )show InnerNormalBasisFieldFunctions

---

InnerNormalBasisFieldFunctions (INBFF)

Exports:
basis dAndcExp expPot index inv
lookup minimalPolynomial norm normalElement normal?
pol qPot random repSq setFieldInfo
trace xn ?*? ?**? ?/?

— package INBFF InnerNormalBasisFieldFunctions —
\texttt{InnerNormalBasisFieldFunctions}(GF): Exports == Implementation where
\begin{align*}
\text{GF} & : \text{FiniteFieldCategory} \quad \text{-- the ground field} \\
\text{PI} & \Rightarrow \text{PositiveInteger} \\
\text{NNI} & \Rightarrow \text{NonNegativeInteger} \\
\text{I} & \Rightarrow \text{Integer} \\
\text{SI} & \Rightarrow \text{SingleInteger} \\
\text{SUP} & \Rightarrow \text{SparseUnivariatePolynomial} \\
\text{VGF} & \Rightarrow \text{Vector GF} \\
\text{M} & \Rightarrow \text{Matrix} \\
\text{V} & \Rightarrow \text{Vector} \\
\text{L} & \Rightarrow \text{List} \\
\text{OUT} & \Rightarrow \text{OutputForm} \\
\text{TERM} & \Rightarrow \text{Record(value:GF,index:SI)} \\
\text{MM} & \Rightarrow \text{ModMonic(GF,SUP GF)} \\
\end{align*}

Exports == with
\begin{align*}
\text{setFieldInfo}: (\text{V L TERM},\text{GF}) & \rightarrow \text{Void} \\
\text{random : PI} & \rightarrow \text{VGF} \\
\text{index : (PI,PI)} & \rightarrow \text{VGF} \\
\text{pol : VGF} & \rightarrow \text{SUP GF} \\
\text{xn : NNI} & \rightarrow \text{SUP GF}
\end{align*}
**xn(n)** returns the polynomial \(x^{n-1}\).

dAndcExp : (VGF,NNI,SI) -> VGF
++ dAndcExp(v,n,k) computes \(v^{e}\) interpreting v as an element of
++ normal basis field. A divide and conquer algorithm similar to the
++ one from D.R.Stinson,
++ "Some observations on parallel Algorithms for fast exponentiation in
++ GF(2^m)". Siam J. Computation, Vol.19, No.4, pp.711-717, August 1990
++ is used. Argument k is a parameter of this algorithm.

repSq : (VGF,NNI) -> VGF
++ repSq(v,e) computes \(v^{e}\) by repeated squaring,
++ interpreting v as an element of a normal basis field.

expPot : (VGF,SI,SI) -> VGF
++ expPot(v,e,d) returns the sum from \(i = 0\) to
++ \(v^{(q^i*d)}\), interpreting
++ v as an element of a normal basis field and where q is
++ the size of the ground field.
++ Note that for a description of the algorithm, see
++ T.Itoh and S.Tsujii,
++ "A fast algorithm for computing multiplicative inverses in GF(2^m)
++ using normal bases",

qPot : (VGF,SI) -> VGF
++ qPot(v,e) computes \(v^{(q^e)}\), interpreting v as an element of
++ normal basis field, q the size of the ground field.
++ This is done by a cyclic e-shift of the vector v.

-- the semantic of the following functions is obvious from the finite field
-- context, for description see category FAXF

"**" : (VGF,I) -> VGF
++ x**n \ undocumented{}
++ See \axiomFunFrom{**}{DivisionRing}

"*" : (VGF,VGF) -> VGF
++ x*y \ undocumented{}
++ See \axiomFunFrom{*}{SemiGroup}

="/" : (VGF,VGF) -> VGF
++ x/y \ undocumented{}
++ See \axiomFunFrom{/}{Field}

norm : (VGF,PI) -> VGF
++ norm(x,n) \ undocumented{}
++ See \axiomFunFrom{norm}{FiniteAlgebraicExtensionField}

trace : (VGF,PI) -> VGF
++ trace(x,n) \ undocumented{}
++ See \axiomFunFrom{trace}{FiniteAlgebraicExtensionField}

inv : VGF -> VGF
++ inv x \ undocumented{}
++ See \axiomFunFrom{inv}{DivisionRing}

lookup : VGF -> PI
++ lookup(x) \ undocumented{}
++ See \axiomFunFrom{lookup}{Finite}

normal? : VGF -> Boolean
++ normal?(x) \ undocumented{} 
++ See \axiomFunFrom{normal?}{FiniteAlgebraicExtensionField}

basis : PI \rightarrow V \text{VGF}
++ basis(n) \ undocumented{}
++ See \axiomFunFrom{basis}{FiniteAlgebraicExtensionField}

normalElement : PI \rightarrow \text{VGF}
++ normalElement(n) \ undocumented{}
++ See \axiomFunFrom{normalElement}{FiniteAlgebraicExtensionField}

minimalPolynomial : \text{VGF} \rightarrow \text{SUP GF}
++ minimalPolynomial(x) \ undocumented{}
++ See \axiomFunFrom{minimalPolynomial}{FiniteAlgebraicExtensionField}

Implementation == add

-- global variables

sizeGF: \text{NNI} := \text{size}\()$\text{GF}$
-- the size of the ground field

multTable: \text{V L TERM} := \text{new}(1,\text{nil}\()$(\text{L TERM})\)$(\text{V L TERM})$
-- global variable containing the multiplication table

trGen: \text{GF} := 1$\text{GF}$
-- controls the imbedding of the ground field

logq: \text{List SI} := \[0,10::\text{SI},16::\text{SI},20::\text{SI},23::\text{SI},0,28::\text{SI},-
\quad 30::\text{SI},32::\text{SI},0,35::\text{SI}\]
-- logq.i is about $10\log_2(i)$ for the values <12 which
-- can match sizeGF. It's used by "**"

expTable: \text{L L SI} := \[
\quad [4::\text{SI},12::\text{SI},48::\text{SI},160::\text{SI},480::\text{SI},0],-
\quad [8::\text{SI},72::\text{SI},432::\text{SI},0],-
\quad [18::\text{SI},216::\text{SI},0],-
\quad [32::\text{SI},480::\text{SI},0],[],-
\quad [72::\text{SI},0],[98::\text{SI},0],[128::\text{SI},0],[200::\text{SI},0]]
-- expT is used by "**" to optimize the parameter k
-- before calling dAndcExp(......,k)

-- functions

-- computes a**(-1) = a**((q**extDeg)-2)
-- see reference of function expPot
inv(a) ==
  b:VGF := qPot(expPot(a, (#a-1)::NNI::SI,1::SI)$$.1)$$.1$$
  erg:VGF := inv((a $$b).1 *$\text{GF trGen}$\text{GF} *$VGF b

-- "**" decides which exponentiation algorithm will be used, in order to
-- get the fastest computation. If dAndcExp is used, it chooses the
-- optimal parameter k for that algorithm.
a ** ex ==
    e:NNI:=positiveRemainder(ex,sizeGF**((#a)::PI)-1)$I :: NNI
    zero?(e)$NNI => new(#a,0)$VGF
    one?(e)$NNI => copy(a)$VGF
    inGroundField?(a) => new(#a,((a.1*trGen) **$GF e))$VGF
    e1:SI:=(length(e)$I)::SI
    sizeGF >$I 11 =>
        q1:SI:=(length(sizeGF)$I)::SI
        logqe:SI:=(e1 quo$SI q1) +$SI 1$SI
        10::SI * (logqe + sizeGF-2) > 15::SI * e1 =>
            print("repeatedSquaring":"::OUT)
            repSq(a,e)
        print("divAndConquer(a,e,1)"::OUT)
        dAndcExp(a,e,1)
    logqe:SI:=((10::SI *$SI e1) quo$SI (logq.sizeGF)) +$SI 1$SI
    k:SI:=1$SI
    expT:List SI:=expTable.sizeGF
    while (logqe >= expT.k) and not zero? expT.k repeat k:=k +$SI 1$SI
    mult:I:=(sizeGF-1) *$I sizeGF **$I ((k-1)pretend NNI) +$I_
        ((logqe +$SI k -$SI 1$SI) quo$SI k)::$I -$I 2
        (10*mult) >= (15 * (e1::I)) =>
            print("repeatedSquaring(a,e)"::OUT)
            repSq(a,e)
        print(hconcat(["divAndConquer(a,e,:"::OUT,k::OUT,")"::OUT])$OUT)
        dAndcExp(a,e,k)

    -- computes a**e by repeated squaring
    repSq(b,e) ==
        a:=copy(b)$VGF
    one? (e) => a
    odd?(e)$I => a * repSq(a*a,(e quo 2) @ NNI)
    repSq(a*a,(e quo 2) @ NNI)

    -- computes a**e using the divide and conquer algorithm similar to the
    -- one from D.R.Stinson,
    -- "Some observations on parallel Algorithms for fast exponentiation in
    -- GF(2^n)", Siam J. Computation, Vol.19, No.4, pp.711-717, August 1990
    dAndcExp(a,e,k) ==
        plist:List VGF:=[copy(a)$VGF]
        qk:1:=sizeGF**k pretend NNI
        for j in 2..(qk-1) repeat
            if positiveRemainder(j,sizeGF)=0 then b:=qPot(plist.(j quo sizeGF),1)$VGF
            else b:=a *$$ last(plist)$List VGF
        plist:=concat(plist,b)
        l:List NNI:=nil()
        ex:I:=e
        while not(ex = 0) repeat
            l:=concat(l,positiveRemainder(ex,qk) pretend NNI)
ex := ex quo qk
if first(l) = 0 then
  erg := VGF := new(#a, trGen)$VGF
else
  erg := VGF := plist.(first(l))
i := SI := k
for j in rest(l) repeat
  if j^=0 then
    erg := erg *$$ qPot(plist.j, i)$$
i := i + k
end for

a * b ==
e := SI := (#a)::SI
erg := zero(#a)$VGF
for t in multTable.1 repeat
  for j in 1..e repeat
    y := t.value -- didn’t work without defining x and y
    x := t.index
    k := addmod(x, j::SI, e)$SI +$SI 1$SI
    erg.k := erg.k +$$ GF a.j *$$ GF b.j *$$ GF y$$
  end for
end for

lookup(x) ==
e := I := 0
for j in (#x)..1 by -1 repeat
  erg := (erg * sizeGF) + (lookup(x.j)$GF rem sizeGF)
end for
erg := 0 => (sizeGF**(#x)) :: PI
erg :: PI

-- computes the norm of a over GF**d, d must devide extdeg
-- see reference of function expPot below
norm(a, d) ==
d := d::SI
r := divide((#a)::SI, d)$SI
not(r.remainder = 0) => error "norm: 2.arg must divide extdeg"
expPot(a, r.quotient, d)$SI

-- computes expPot(a, e, d) = sum form i = 0 to e-1 over a**(q**id))
-- see T. Itoh and S. Tsujii,
-- "A fast algorithm for computing multiplicative inverses in GF(2**m)
-- using normal bases",
-- Information and Computation 78, pp. 171-177, 1988
expPot(a, e, d) ==
d := (#a)::SI
e := 1 => copy(a)$VGF
k2:SI:=d
y:=copy(a)
if bit?(e,0) then
   erg:=copy(y)
   qpot:SI:=k2
else
   erg:=new(#a,inv(trGen)$GF)$VGF
   qpot:SI:=0
for k in 1..length(e) repeat
   y:= y **$ qPot(y,k2)
   k2:=addmod(k2,k2,deg)$SI
   if bit?(e,k) then
      erg:=erg **$ qPot(y,qpot)
      qpot:=addmod(qpot,k2,deg)$SI
   end if
end for

trace(a,d) ==
dSI:=d::SI
r:=divide((#a)::SI,dSI)$SI
not(r.remainder = 0) => error "trace: 2.arg must divide extdeg"
v:=copy(a.(1..dSI))$VGF
sSI:SI:=r.quotient
for i in 1..dSI repeat
   for j in 1..sSI-1 repeat
      v.i:=v.i+a.(i+j::SI*dSI)
   end for
   v
end for

random(n) ==
v:=zero(n)$VGF
for i in 1..n repeat v.i:=random()$GF
v

xn(m) == monomial(1,m)$(SUP GF) - 1$(SUP GF)

normal?(x) ==
gcd(xn(#x),pol(x))$(SUP GF) = 1 => true
false

x:VGF / y:VGF == x **$ inv(y)$$
setFieldInfo(m,n) ==
    multTable:=m
    trGen:=n
    void()$Void

minimalPolynomial(x) ==
    dx:=#x
    y:=new(#x,inv(trGen)$GF)$VGF
    m:=zero(dx,dx+1)$(M GF)
    for i in 1..dx+1 repeat
        dy:=#y
        for j in 1..dy repeat
            for k in 0..((dx quo dy)-1) repeat
                qsetelt_!(m,j+k*dy,i,y.j)$(M GF)
            end for
        end for
        y:=y *$$ x
    end for
    v:=first nullSpace(m)$(M GF)
    pol(v)$$

basis(n) ==
    bas:=(V VGF):=new(n,zero(n)$VGF)$(V VGF)
    for i in 1..n repeat
        uniti:=zero(n)$VGF
        qsetelt_!(uniti,i,1$GF)$VGF
        qsetelt_!(bas,i,uniti)$(V VGF)
    end for
    bas

normalElement(n) ==
    v:=zero(n)$VGF
    qsetelt_!(v,1,1$GF)
    v
    -- normalElement(n) == index(n,1)$$

index(degm,n) ==
    m:I:=n rem$I (sizeGF ** degm)
    erg:=zero(degm)$VGF
    for j in 1..degm repeat
        erg.j:=index((sizeGF+(m rem sizeGF)) pretend PI)$GF
    end for
    m:=m quo sizeGF
    erg

pol(x) ==
    +/[monomial(x.i,(i-1)::NNI)$(SUP GF) for i in 1..(#x)::I]


— INBFF.dotabb —
package INEP InnerNumericEigenPackage

— InnerNumericEigenPackage.input —

)set break resume
)sys rm -f InnerNumericEigenPackage.output
)spool InnerNumericEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerNumericEigenPackage
--R
--R InnerNumericEigenPackage(K: Field,F: Field,Par: Join(Field,OrderedRing)) is a package constructor
--R Abbreviation for InnerNumericEigenPackage is INEP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INEP
--R
--R------------------------------- Operations --------------------------------
--R charpol : Matrix(K) -> SparseUnivariatePolynomial(K)
--R innerEigenvectors : (Matrix(K),Par,(SparseUnivariatePolynomial(K) -> Factored(SparseUnivariatePolynomial(K)) ...
--R solve1 : (SparseUnivariatePolynomial(K),Par) -> List(F)
--R
--E 1

)spool
)lisp (bye)

— InnerNumericEigenPackage.help —

====================================================================
InnerNumericEigenPackage examples
====================================================================

This package is the inner package to be used by NumericRealEigenPackage and NumericComplexEigenPackage for the computation of numeric
CHAPTER 10. CHAPTER I

eigenvalues and eigenvectors.

See Also:
o )show InnerNumericEigenPackage

InnerNumericEigenPackage (INEP)

Exports:
charpol innerEigenvectors solve1

--- package INEP InnerNumericEigenPackage ---

)abbrev package INEP InnerNumericEigenPackage
++ Author:P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated:Spring 1991
++ Description:
++ This package is the inner package to be used by NumericRealEigenPackage
++ and NumericComplexEigenPackage for the computation of numeric
++ eigenvalues and eigenvectors.

InnerNumericEigenPackage(K,F,Par) : C == T

where
F : Field -- this is the field where the answer will be
    -- for dealing with the complex case
K : Field -- type of the input
Par : Join(Field,OrderedRing) -- it will be NF or RN

SE == Symbol()
RN == Fraction Integer
I == Integer
NF ==> Float
CF ==> Complex Float
GRN ==> Complex RN
GI ==> Complex Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
MRN ==> Matrix RN

MK ==> Matrix K
PK ==> Polynomial K
MF ==> Matrix F
SUK ==> SparseUnivariatePolynomial K
SUF ==> SparseUnivariatePolynomial F
SUP ==> SparseUnivariatePolynomial
MSUK ==> Matrix SUK

PEigenForm ==> Record(algpol:SUK,almult:Integer,poleigen:List(MSUK))

outForm ==> Record(outval:F,outmult:Integer,outvect:List MF)

IntForm ==> Union(outForm,PEigenForm)

UFactor ==> (SUK -> Factored SUK)

C == with

charpol : MK -> SUK
  ++ charpol(m) computes the characteristic polynomial of a matrix
  ++ m with entries in K.
  ++ This function returns a polynomial
  ++ over K, while the general one (that is in EiegenPackage) returns
  ++ Fraction P K

solve1 : (SUK, Par) -> List F
  ++ solve1(pol, eps) finds the roots of the univariate polynomial
  ++ polynomial pol to precision eps. If K is \texttt{Fraction Integer}
  ++ then only the real roots are returned, if K is
  ++ \texttt{Complex Fraction Integer} then all roots are found.

innerEigenvectors : (MK,Par,UFactor) -> List(outForm)
  ++ innerEigenvectors(m,eps,factor) computes explicitly
  ++ the eigenvalues and the correspondent eigenvectors
  ++ of the matrix m. The parameter eps determines the type of
  ++ the output, factor is the univariate factorizer to be used
  ++ to reduce the characteristic polynomial into irreducible factors.

T == add

numeric(r:K):F ==
  K is RN =>
  F is NF => convert(r)$RN
  F is RN => r
F is CF => r :: RN :: CF
F is GRN => r :: RN :: GRN
K is GRN =>
  F is GRN => r
  F is CF => convert(convert r)
error "unsupported coefficient type"

---- next functions needed for defining ModularField ----

monicize(f:SUK) : SUK ==
  (a:=leadingCoefficient f) =1 => f
  inv(a)*f

reduction(u:SUK,p:SUK):SUK == u rem p

merge(p:SUK,q:SUK):Union(SUK,"failed") ==
  p = q => p
  p = 0 => q
  q = 0 => p
  "failed"

  val:=extendedEuclidean(v,p,u)
  val case "failed" => "failed"
  val.coef1

---- eval a vector of F in a radical expression ----

evalvect(vect:MSUK,alg:F) : MF ==
  n:=nrows vect
  w:MF:=zero(n,1)$MF
  for i in 1..n repeat
    polf:=map(numeric,
        vect(i,1))$UnivariatePolynomialCategoryFunctions2(K,SUK,F,SUF)
    v:F:=elt(polf,alg)
    setelt(w,i,1,v)
  w

---- internal function for the computation of eigenvectors ----

eigen(A:MK,p:SUK,fact:UFactor) : List(IntForm) ==
  dimA:NNI:= nrows A
  MM:=ModularField(SUK,SUK,reduction,merge,exactquo)
  AM:=Matrix(MM)
  lff:=factors fact(p)
  res: List IntForm :=[]
  lr : List MF:=[]
  for ff in lff repeat
    pol:SUK:= ff.factor
    if (degree pol)=1 then
      alpha:K:=coefficient(pol,0)/leadingCoefficient pol
      -- compute the eigenvectors, rational case
B1:MK := zero(dimA,dimA)$MK
for i in 1..dimA repeat
  for j in 1..dimA repeat B1(i,j):=A(i,j)
  B1(i,i):= B1(i,i) - alpha
lr:=[ ]
for vecr in nullSpace B1 repeat
  wf:MF:=zero(dimA,1)
  for i in 1..dimA repeat wf(i,1):=numeric vecr.i
  lr:=cons(wf,lr)
res:=cons([numeric alpha,ff.exponent,lr]$outForm,res)
else
  ppol:=monicize pol
  alg:MM:= reduce(monomial(1,1),ppol)
  B:AM:= zero(dimA,dimA)$AM
  for i in 1..dimA repeat
    for j in 1..dimA repeat B(i,j):=reduce(A(i,j) ::SUK,ppol)
    B(i,i):=B(i,i) - alg
  sln2:=nullSpace B
  soln:List MSUK := [ ]
  for vec in sln2 repeat
    wk:MSUK:=zero(dimA,1)
    for i in 1..dimA repeat wk(i,1):=(vec.i)::SUK
    soln:=cons(wk,soln)
  res:=cons([ff.factor,ff.exponent,soln]$PEigenForm,
             res)
res

if K is RN then
  solve1(up:SUK, eps:Par) : List(F) ==
  denom := "lcm"/[denom(c::RN) for c in coefficients up]
  up:=denom*up
  upi:=map(numer,up)_$UnivariatePolynomialCategoryFunctions2(RN,SUP RN,I,SUP I)
  innerSolve1(upi, eps)$InnerNumericFloatSolvePackage(I,F,Par)
else if K is GRN then
  solve1(up:SUK, eps:Par) : List(F) ==
  denom := "lcm"/[lcm(denom real(c::GRN), denom imag(c::GRN))
                 for c in coefficients up]
  up:=denom*up
  upgi := map((c:GRN):GI+->complex(numer(real c), numer(imag c)),up)_
          $UnivariatePolynomialCategoryFunctions2(GRN,SUP GRN,GI,SUP GI)
  innerSolve1(upgi, eps)$InnerNumericFloatSolvePackage(GI,F,Par)
else error "unsupported matrix type"

---- the real eigenvectors expressed as floats ----
innerEigenvectors(A:MK,eps:Par,fact:UFactor) : List outForm ==
pol:= charpol A
sln1:List(IntForm):=inteigen(A,pol,fact)
n:=nrows A
sln: List(outForm) := []
for lev in sln1 repeat
  lev case outForm => sln := cons(lev, sln)
  leva := lev :: PEigenForm
  lval := List(F) := solve1(leva.algp1, eps)
  lvect := leva.poleigen
  lmult := leva.almult
  for alg in lval repeat
    nsl := [alg, lmult, [evalvect(ep, alg) for ep in lvect]] $ outForm
    sln := cons(nsl, sln)
sln

charpol(A: MK) : SUK ==
dimA : PI := (nrows A): PI
  dimA ^= ncols A => error " The matrix is not square"
B : Matrix SUK := zero(dimA, dimA)
for i in 1..dimA repeat
  for j in 1..dimA repeat
    B(i, j) := A(i, j) :: SUK
  B(i, i) := B(i, i) - monomial(i, i) $ SUK
determinant B

package INFSP InnerNumericFloatSolvePackage

— InnerNumericFloatSolvePackage.input —

)set break resume
)sys rm -f InnerNumericFloatSolvePackage.output
)spool InnerNumericFloatSolvePackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
--- R InnerNumericFloatSolvePackage(K::GcdDomain,F::Field,Par::Join(Field,OrderedRing)) is a package constructor
-- R Abbreviation for InnerNumericFloatSolvePackage is INFSP
-- R This constructor is not exposed in this frame.
-- R Issue )edit bookvol10.4.pamphlet to see algebra source code for INFSP
-- R
-- R---------------------------------- Operations --------------------------------
-- R innerSolve : (List(Polynomial(K)),List(Polynomial(K)),List(Symbol),Par) -> List(List(F))
-- R innerSolve1 : (SparseUnivariatePolynomial(K),Par) -> List(F)
-- R innerSolve1 : (Polynomial(K),Par) -> List(F)
-- R makeEq : (List(F),List(Symbol)) -> List(Equation(Polynomial(F)))
-- R
-- E 1

)spool
)lisp (bye)

---
--- InnerNumericFloatSolvePackage.help ---

====================================================================
InnerNumericFloatSolvePackage examples
====================================================================

This is an internal package
for computing approximate solutions to systems of polynomial equations.
The parameter K specifies the coefficient field of the input polynomials
and must be either Fraction(Integer) or Complex(Fraction Integer).

The parameter F specifies where the solutions must lie and can
be one of the following: Float, Fraction(Integer), Complex(Float), or
Complex(Fraction Integer). The last parameter specifies the type
of the precision operand and must be either Fraction(Integer) or Float.

See Also:
o )show InnerNumericFloatSolvePackage
INNERNUMERICFLOATSOLVEPACKAGE (INFSP)

Exports:
innerSolve  innerSolve1  makeEq

— package INFSP InnerNumericFloatSolvePackage —

)abbrev package INFSP InnerNumericFloatSolvePackage
++ Author: P. Gianni
++ Date Created: January 1990
++ Description:
++ This is an internal package
++ for computing approximate solutions to systems of polynomial equations.
++ The parameter K specifies the coefficient field of the input polynomials
++ and must be either \spad{Fraction(Integer)} or
++ \spad{Complex(Fraction Integer)}.
++ The parameter F specifies where the solutions must lie and can
++ be one of the following: \spad{Float}, \spad{Fraction(Integer)},
++ \spad{Complex(Float)},
++ \spad{Complex(Fraction Integer)}. The last parameter specifies the type
++ of the precision operand and must be either \spad{Fraction(Integer)} or
++ \spad{Float}.

InnerNumericFloatSolvePackage(K,F,Par): Cat == Cap where
  F : Field -- this is the field where the answer will be
  K : GcdDomain -- type of the input
  Par : Join(Field, OrderedRing ) -- it will be NF or RN

  I    ==> Integer
  NNI  ==> NonNegativeInteger
  P    ==> Polynomial
  EQ   ==> Equation
  L    ==> List
  SUP  ==> SparseUnivariatePolynomial
  RN   ==> Fraction Integer
  NF   ==> Float
  CF   ==> Complex Float
GI ==> Complex Integer
GRN ==> Complex RN
SE ==> Symbol
RFI ==> Fraction P I

Cat == with

innerSolve1 : (SUP K,Par) -> L F
++ innerSolve1(up,eps) returns the list of the zeros
++ of the univariate polynomial up with precision eps.
innerSolve1 : (P K,Par) -> L F
++ innerSolve1(p,eps) returns the list of the zeros
++ of the polynomial p with precision eps.
innerSolve : (L P K,L P K,L SE,Par) -> L L F
++ innerSolve(lnum,lden,lvar,eps) returns a list of
++ solutions of the system of polynomials lnum, with
++ the side condition that none of the members of lden
++ vanish identically on any solution. Each solution
++ is expressed as a list corresponding to the list of
++ variables in lvar and with precision specified by eps.
makeEq : (L F,L SE) -> L EQ P F
++ makeEq(lsol,lvar) returns a list of equations formed
++ by corresponding members of lvar and lsol.

Cap == add

------ Local Functions ------
isGeneric? : (L P K,L SE) -> Boolean
evaluate : (P K,SE,SE,F) -> F
numeric : K -> F
oldCoord : (L F,L I) -> L F
findGenZeros : (L P K,L SE,Par) -> L L F
failPolSolve : (L P K,L SE) -> Union(L L P K,"failed")

numeric(r:K):F ==
K is I =>
  F is Float => r::I::Float
  F is RN => r::I::RN
  F is CF => r::I::CF
  F is GRN => r::I::GRN
K is GI =>
gr:GI := r::GI
  F is GRN => complex(real(gr)::RN,imag(gr)::RN)$GRN
  F is CF => convert(gr)
error "case not handled"

-- construct the equation
makeEq(nres:L F,lv:L SE) : L EQ P F ==
  [equation(x::(P F),r::(P F)) for x in lv for r in nres]
evaluate(pol:PK,xvar:SE,zvar:SE,z:F):F ==
  rpp:=map(numeric,pol)$PolynomialFunctions2(K,F)
  rpp := eval(rpp,zvar,z)
  upol:=univariate(rpp,xvar)
  retract(-coefficient(upol,0))/retract(leadingCoefficient upol)

myConvert(eps:Par) : RN ==
  Par is RN => eps
  Par is NF => retract(eps)$NF

innerSolve1(pol:PK,eps:Par) : L F == innerSolve1(univariate pol,eps)

innerSolve1(upol:SUP K,eps:Par) : L F ==
  K is GI and (Par is RN or Par is NF) =>
    (complexZeros(upol,
      eps)$ComplexRootPackage(SUP K,Par)) pretend L(F)
  K is I =>
    F is Float =>
      z:= realZeros(upol,myConvert eps)$RealZeroPackage(SUP I)
      [(1/2)*(x.left+x.right)@Float for x in z] pretend L(F)
    F is RN =>
      z:= realZeros(upol,myConvert eps)$RealZeroPackage(SUP I)
      [(1/2)*(x.left + x.right) for x in z] pretend L(F)
    error "improper arguments to INFSP"
  error "improper arguments to INFSP"

-- find the zeros of components in "generic" position --
findGenZeros(lp:L P K,rlvar:L SE,eps:Par) : L L F ==
  rlp:=reverse lp
  f:=rlp.first
  zvar:= rlvar.first
  rlp:=rlp.rest
  lz:=innerSolve1(f,eps)
  [reverse cons(z,[evaluate(pol,xvar,zvar,z) for pol in rlp
    for xvar in rlvar.rest]) for z in lz]

-- convert to the old coordinates --
oldCoord(numres:L F,lval:L I) : L F ==
  rnumres:=reverse numres
  rnumres.first:= rnumres.first +
    (+/[n*nr for n in lval for nr in rnumres.rest])
  reverse rnumres

-- real zeros of a system of 2 polynomials lp (incomplete)
  mainvar := first lv
  up1:=univariate(lp.1, mainvar)
  up2:=univariate(lp.2, mainvar)
vec := subresultantVector(up1,up2)$SubResultantPackage(P K,SUP P K)
p0 := primitivePart multivariate(vec.0, mainvar)
p1 := primitivePart(multivariate(vec.1, mainvar),mainvar)
zero? p1 or
gcd(p0, leadingCoefficient(univariate(p1,mainvar))) ^=1 =>
innerSolve(cons(0,lp),empty(),lv,eps)
findGenZeros([p1, p0], reverse lv, eps)

-- real zeros of the system of polynomial lp --
-- empty?(ld) and (#lv = 2) and (# lp = 2) => innerSolve2(lp, lv, eps)
lv:= [pToDmp(p)$PolToPol(lv,K) for p in lp]
OV:=OrderedVariableList(lv)
lpv:L OV:= [variable(vv)::OV for vv in lv]
DP:=DirectProduct(#lv,NonNegativeInteger)
dmp:=DistributedMultivariatePolynomial(lv,K)
lq:L dmp:=[]
if ld"=[] then
  lq:= [(pToDmp(q1)$PolToPol(lv,K)) pretend dmp for q1 in ld]
partRes:=groebSolve(lnp,lvv)$GroebnerSolve(lv,K,K) pretend (L L dmp)
partRes=list [] => []
-- remove components where denominators vanish
if lq"=[] then
gb:=GroebnerInternalPackage(K,DirectProduct(#lv,NNI),OV,dmp)
partRes:=[pr for pr in partRes|
    and/
    [(redPol(fq,pr pretend List(dmp))$gb) ^=0 for fq in lq]]

-- select the components in "generic" form
rlv:=reverse lv
rrlvv:= rest reverse lvv
listGen:L L dmp:=[]
for res in partRes repeat
  res1:=rest reverse res
  "and/"[/degree(f,rrlvv)=1 for f in res1] =>
  listGen:=concat(res pretend (L dmp),listGen)
result:L L F := []
if listGen"=[] then
  listG :L L P K:=
  [[dmpToP(pf)$PolToPol(lv,K) for pf in pr] for pr in listG]
result:=
  "append"/[findGenZeros(res,rlv,eps) for res in listG]
for gres in listGen repeat
  partRes:=delete(partRes,position(gres,partRes))
-- adjust the non-generic components
for gres in partRes repeat
  genRecord := genericPosition(gres,lvv)$GroebnerSolve(lv,K,K)
lgen := genRecord.dpolys
lval := genRecord.coords
lgen1:=[dmpToP(pf)\$PolToPol(lv,K) for pf in lgen]
lris:=findGenZeros(lgen1,rlv,eps)
result:= append([oldCoord(r,lval) for r in lris],result)
result

---

--- INFSP.dotabb ---

"INFSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INFSP"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"INFSP" -> "COMPCAT"

---

package INPSIGN InnerPolySign

--- InnerPolySign.input ---

)set break resume
)sys rm -f InnerPolySign.output
)spool InnerPolySign.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerPolySign
--) InnerPolySign(R: Ring,UP: UnivariatePolynomialCategory(R)) is a package constructor
--) Abbreviation for InnerPolySign is INPSIGN
--) This constructor is not exposed in this frame.
--) Issue )edit bookvol10.4.pamphlet to see algebra source code for INPSIGN
--) 
--)---------------------------------- Operations ----------------------------------
--) signAround : (UP,Integer,(R -> Union(Integer,"failed"))) -> Union(Integer,"failed")
--) signAround : (UP,R,(R -> Union(Integer,"failed"))) -> Union(Integer,"failed")
--) 
--)E 1
)
)spool
)lisp (bye)
Find the sign of a polynomial around a point or infinity.

See Also:
o )show InnerPolySign

---

InnerPolySign (INPSIGN)

Exports:
signAround

---

)abbrev package INPSIGN InnerPolySign
++ Author: Manuel Bronstein
++ Date Created: 23 Aug 1989
++ Date Last Updated: 19 Feb 1990
++ Description:
++ Find the sign of a polynomial around a point or infinity.

InnerPolySign(R, UP): Exports == Implementation where
R : Ring
UP: UnivariatePolynomialCategory R
U ==> Union(Integer, "failed")

Exports ==> with
  signAround: (UP, Integer, R -> U) -> U
    + signAround(u,i,f) \ undocumented
  signAround: (UP, R, Integer, R -> U) -> U
    + signAround(u,r,i,f) \ undocumented
  signAround: (UP, R, R -> U) -> U
    + signAround(u,r,f) \ undocumented

Implementation ==> add
signAround(p:UP, x:R, rsign:R -> U) ==
  (ur := signAround(p, x, 1, rsign)) case "failed" => "failed"
  (ul := signAround(p, x, -1, rsign)) case "failed" => "failed"
  (ur::Integer) = (ul::Integer) => ur "failed"

signAround(p, x, dir, rsign) ==
  zero? p => 0
  zero?(r := p x) =>
    (u := signAround(differentiate p, x, dir, rsign)) case "failed"
    => "failed"
    dir * u::Integer
  rsign r

signAround(p:UP, dir:Integer, rsign:R -> U) ==
  zero? p => 0
  (u := rsign leadingCoefficient p) case "failed" => "failed"
  (dir > 0) or (even? degree p) => u::Integer
  - (u::Integer)

— INPSIGN.dotabb —

"INPSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INPSIGN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INPSIGN" -> "PFECAT"

— package ISUMP InnerPolySum —
')set break resume
)sys rm -f InnerPolySum.output
)spool InnerPolySum.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InnerPolySum
--R
--R InnerPolySum(E: OrderedAbelianMonoidSup,V: OrderedSet,R: IntegralDomain,P: PolynomialCategory(R,E,V)) is a package constructor
--R Abbreviation for InnerPolySum is ISUMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ISUMP
--R
--R------------------------------- Operations --------------------------------
--R sum : (P,V,Segment(P)) -> Record(num: P,den: Integer)
--R sum : (P,V) -> Record(num: P,den: Integer)
--R
--E 1

)spool
)lisp (bye)

— InnerPolySum.help —

====================================================================
InnerPolySum examples
====================================================================

Tools for the summation packages of polynomials

See Also:
  o )show InnerPolySum

—-
InnerPolySum (ISUMP)

Exports:

sum

— package ISUMP InnerPolySum —

)abbrev package ISUMP InnerPolySum
++ Author: SMW
++ Date Last Updated: 19 April 1991
++ Description:
++ Tools for the summation packages of polynomials

InnerPolySum(E, V, R, P): Exports == Impl where

E: OrderedAbelianMonoidSup
V: OrderedSet
R: IntegralDomain
P: PolynomialCategory(R, E, V)

Z ==> Integer
Q ==> Fraction Z
SUP ==> SparseUnivariatePolynomial

Exports ==> with

sum: (P, V, Segment P) -> Record(num:P, den:Z)
++ sum(p(n), n = a..b) returns \spad{p(a) + p(a+1) + \ldots + p(b)}.
sum: (P, V) -> Record(num:P, den: Z)
++ sum(p(n), n) returns \spad{P(n)},
++ the indefinite sum of \spad{p(n)} with respect to
++ upward difference on n, i.e. \spad{P(n+1) - P(n) = a(n)};

Impl ==> add
import PolynomialNumberTheoryFunctions()
import UnivariatePolynomialCommonDenominator(Z, Q, SUP Q)

pmul: (P, SUP Q) -> Record(num:SUP P, den:Z)
pmul(c, p) ==
  pn := (rec := splitDenominator p).num
  [map(x +-> numer(x) * c, pn)_
   $SparseUnivariatePolynomialFunctions2(Q, P), rec.den]

sum(p, v, s) ==
  indef := sum(p, v)
  [eval(indef.num, v, 1 + hi s) - eval(indef.num, v, lo s),
   indef.den]

sum(p, v) ==
  up := univariate(p, v)
  lp := nil()$List(SUP P)
  ld := nil()$List(Z)
  while up ^= 0 repeat
    ud := degree up; uc := leadingCoefficient up
    up := reductum up
    rec := pmul(uc, 1 / (ud+1) * bernoulli(ud+1))
    lp := concat(rec.num, lp)
    ld := concat(rec.den, ld)
  d := lcm ld
  vp := +/[d exquo di::Z * pi for di in ld for pi in lp]
  [multivariate(vp, v), d]

package ITRIGMNP InnerTrigonometricManipulations

— ISUMP.dotabb —

"ISUMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ISUMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ISUMP" -> "PFECAT"

package ITRIGMNP InnerTrigonometricManipulations

— InnerTrigonometricManipulations.input —

)set break resume
)sys rm -f InnerTrigonometricManipulations.output
)spool InnerTrigonometricManipulations.output
)set message test on
)set message auto off
)clear all
This package provides transformations from trigonometric functions to exponentials and logarithms, and back.

F and FG should be the same type of function space.

See Also:
- \( \text{show InnerTrigonometricManipulations} \)
InnerTrigonometricManipulations (ITRIGMNP)

Exports:
F2FG  FG2F  GF2FG  explogs2trigs  trigs2explogs

— package ITRIGMNP InnerTrigonometricManipulations —

)abbrev package ITRIGMNP InnerTrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 9 October 1993
++ Description:
++ This package provides transformations from trigonometric functions
++ to exponentials and logarithms, and back.
++ F and FG should be the same type of function space.

InnerTrigonometricManipulations(R,F,FG): Exports == Implementation where
  R : Join(IntegralDomain, OrderedSet)
  F : Join(FunctionSpace R, RadicalCategory,
        TranscendentalFunctionCategory)
  FG : Join(FunctionSpace Complex R, RadicalCategory,
            TranscendentalFunctionCategory)

Z  ==> Integer
SY ==> Symbol
OP ==> BasicOperator
GR ==> Complex R
GF ==> Complex F
KG ==> Kernel FG
PG ==> SparseMultivariatePolynomial(GR, KG)
UP ==> SparseUnivariatePolynomial PG
NTHR ==> "nthRoot":SY

Exports ==> with
  GF2FG : GF -> FG
  ++ GF2FG(a + i b) returns \spad{a + i b} viewed as a function with
  ++ the \spad{i} pushed down into the coefficient domain.
FG2F : FG -> F
++ FG2F(a + i b) returns \spad{a + sqrt(-1) b}.
F2FG : F -> FG
++ F2FG(a + sqrt(-1) b) returns \spad{a + i b}.
explogs2trigs: FG -> GF
++ explogs2trigs(f) rewrites all the complex logs and
++ exponentials appearing in \spad{f} in terms of trigonometric
++ functions.
trigs2explogs: (FG, List KG, List SY) -> FG
++ trigs2explogs(f, [k1,...,kn], [x1,...,xm]) rewrites
++ all the trigonometric functions appearing in \spad{f} and involving
++ one of the \spad{xi's} in terms of complex logarithms and
++ exponentials. A kernel of the form \spad{tan(u)} is expressed
++ using \spad{exp(u)**2} if it is one of the \spad{ki's}, in terms of
++ \spad{exp(2*u)} otherwise.

Implementation ==> add
er2explogs: (KG, List KG, List SY) -> FG
sm2explogs: (PG, List KG, List SY) -> FG
supexp : (UP, GF, GF, Z) -> GF
GR2GF : GR -> GF
GR2F : GR -> F
KG2F : KG -> F
PG2F : PG -> F
ker2trigs : (OP, List GF) -> GF
smp2trigs : PG -> GF
sup2trigs : (UP, GF) -> GF

nth := R has RetractableTo(Integer) and F has RadicalCategory
GR2F g == real(g)::F + sqrt(-(1::F)) * imag(g)::F
KG2F k == map(FG2F, k)$ExpressionSpaceFunctions2(FG, F)
FG2F f == (PG2F numer f) / (PG2F denom f)
F2FG f == map(x +-> x::GR, f)$FunctionSpaceFunctions2(R,F,GR,FG)
GF2FG f == (F2FG real f) + complex(0, 1)$GR ::FG * F2FG imag f
GR2GF gr == complex(real(gr)::F, imag(gr)::F)

-- This expects the argument to have only tan and atans left.
-- Does a half-angle correction if k is not in the initial kernel list.
er2explogs(k, 1, lx) ==
  empty?([v for v in variables(kf := k::FG) |
    member?(v, lx)$List(SY))]$List(GF)) => kf
  empty?(args := [trigs2explogs(a, 1, lx) for a in argument k]$List(FG)) => kf
  im := complex(0, 1)$GR :: FG
  z := first args
  is?(k, "tan":Symbol) =>
    e := (member?(k, 1) => exp(im * z) ** 2; exp(2 * im * z))
    - im * (e - 1) /$FG(e + 1)
  is?(k, "atan":Symbol) =>
im * log((1 - \$FG im * \$FG z)/\$FG (1 + \$FG im * \$FG z))\$FG / (2::\$FG)
(operator k) args

trigs2explogs(f, l, lx) ==
    smp2explogs(numer f, l, lx) / smp2explogs(denom f, l, lx)

-- return op(arg) as f + %i g
-- op is already an operator with semantics over R, not GR
ker2trigs(op, arg) ==
    "and"/[zero? imag x for x in arg] =>
        complex(op [real x for x in arg]$List(F), 0)
    a := first arg
    is?(op, "exp"::Symbol) => exp a
    is?(op, "log"::Symbol) => log a
    is?(op, "sin"::Symbol) => sin a
    is?(op, "cos"::Symbol) => cos a
    is?(op, "tan"::Symbol) => tan a
    is?(op, "cot"::Symbol) => cot a
    is?(op, "sec"::Symbol) => sec a
    is?(op, "csc"::Symbol) => csc a
    is?(op, "asin"::Symbol) => asin a
    is?(op, "acos"::Symbol) => acos a
    is?(op, "atan"::Symbol) => atan a
    is?(op, "acot"::Symbol) => acot a
    is?(op, "asec"::Symbol) => asec a
    is?(op, "acsc"::Symbol) => acsc a
    is?(op, "sinh"::Symbol) => sinh a
    is?(op, "cosh"::Symbol) => cosh a
    is?(op, "tanh"::Symbol) => tanh a
    is?(op, "coth"::Symbol) => coth a
    is?(op, "sech"::Symbol) => sech a
    is?(op, "csch"::Symbol) => csch a
    is?(op, "asinh"::Symbol) => asinh a
    is?(op, "acosh"::Symbol) => acosh a
    is?(op, "atanh"::Symbol) => atanh a
    is?(op, "acoth"::Symbol) => acoth a
    is?(op, "asech"::Symbol) => asech a
    is?(op, "acsch"::Symbol) => acsch a
    is?(op, "abs"::Symbol) => sqrt(norm a)::GF
    nth and is?(op, NTHR) => nthRoot(a, retract(second arg)$\mathbb{Z})
error "ker2trigs: cannot convert kernel to gaussian function"

sup2trigs(p, f) ==
    map(smp2trigs, p)$SparseUnivariatePolynomialFunctions2(PG, GF) f

smp2trigs p ==
    map(x -> explogs2trigs(x::FG),GR2GF, p)_
    $Polynomial1CategoryLifting(IndexedExponents KG, KG, GR, PG, GF)

explogs2trigs f ==
(m := mainKernel f) case "failed" =>
  GR2GF(retract(numer f)@GR) / GR2GF(retract(denom f)@GR)
op := operator(operator(k := m::KG))$F
arg := [explogs2trigs x for x in argument k]
num := univariate(numer f, k)
den := univariate(denom f, k)
is?(op, "exp"::Symbol) =>
  e := exp real first arg
  y := imag first arg
  g := complex(e * cos y, e * sin y)$GF
  gi := complex(cos(y) / e, - sin(y) / e)$GF
  supexp(num,g,gi,b := (degree num)::Z quo 2)/supexp(den,g,gi,b)
sup2trigs(num, g := ker2trigs(op, arg)) / sup2trigs(den, g)

supexp(p, f1, f2, bse) ==
  ans:GF := 0
  while p ^= 0 repeat
    g := explogs2trigs(leadingCoefficient(p)::FG)
    if ((d := degree(p)::Z - bse) >= 0) then
      ans := ans + g * f1 ** d
    else ans := ans + g * f2 ** (-d)
    p := reductum p
  ans

PG2F p ==
  map(KG2F, GR2F, p)$PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, F)

smp2explogs(p, l, lx) ==
  map(x +-> ker2explogs(x, l, lx), y +-> y::FG, p)_
  $PolynomialCategoryLifting(IndexedExponents KG, KG, GR, PG, FG)
package INFORM1 InputFormFunctions1

<table>
<thead>
<tr>
<th>InputFormFunctions1.input</th>
</tr>
</thead>
</table>

)set break resume
)sys rm -f InputFormFunctions1.output
)spool InputFormFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InputFormFunctions1
--R
--R InputFormFunctions1(R: Type) is a package constructor
--R Abbreviation for InputFormFunctions1 is INFORM1
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INFORM1
--R
--R----------------------------- Operations --------------------------------
--R interpret : InputForm -> R
packageCall : Symbol -> InputForm
--R
--E 1

)spool
)lisp (bye)

———

— InputFormFunctions1.help —

====================================================================
InputFormFunctions1 examples
====================================================================

Tools for manipulating input forms.

See Also:
o )show InputFormFunctions1

———
InputFormFunctions1 (INFORM1)

Exports:
  interpret  packageCall

— package INFORM1 InputFormFunctions1 —

)abbrev package INFORM1 InputFormFunctions1
--->boot $noSubsumption := false
++ Author: Manuel Bronstein
++ Date Last Updated: 19 April 1991
++ Description:
++ Tools for manipulating input forms.

InputFormFunctions1(R:Type):with
  packageCall: Symbol -> InputForm
  ++ packageCall(f) returns the input form corresponding to f$R.
  interpret : InputForm -> R
  ++ interpret(f) passes f to the interpreter, and transforms
  ++ the result into an object of type R.
== add
  Rname := devaluate(R)$Lisp :: InputForm

  packageCall name ==
  convert([convert("$elt":Symbol), Rname,
           convert name]$List(InputForm))@InputForm

  interpret form ==
  retract(interpret(convert([convert("@":Symbol), form,
                             Rname]$List(InputForm))@InputForm)$InputForm)$AnyFunctions1(R)

— INFORM1.dotabb —
package INTERGB InterfaceGroebnerPackage

— InterfaceGroebnerPackage.input —

)set break resume
)sys rm -f InterfaceGroebnerPackage.output
)spool InterfaceGroebnerPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InterfaceGroebnerPackage
--R InterfaceGroebnerPackage(K: Field,symb: List(Symbol),E: OrderedAbelianMonoidSup,OV: OrderedSet,R: PolynomialCategory(K,E,OV)) is a package constructor
--R Abbreviation for InterfaceGroebnerPackage is INTERGB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTERGB
--R
--R----------------------- Operations -------------------------------
--R groebner : List(R) -> List(R)
--E 1

)spool
)lisp (bye)

— InterfaceGroebnerPackage.help —

====================================================================
InterfaceGroebnerPackage examples
====================================================================

Part of the Package for Algebraic Function Fields in one variable PAFF

See Also:
o )show InterfaceGroebnerPackage
InterfaceGroebnerPackage (INTERGB)

---

Exports:
groebner

--- package INTERGB InterfaceGroebnerPackage ---

)abbrev package INTERGB InterfaceGroebnerPackage
++ Author: Gaetan Hache
++ Date Created: September 1996
++ Date Last Updated: April, 2010, by Tim Daly
++ Description:
++ Part of the Package for Algebraic Function Fields in one variable PAFF
InterfaceGroebnerPackage(K,symb,E,OV,R):Exports == Implementation where
  K:Field
  symb: List Symbol
  E:OrderedAbelianMonoidSup
  OV: OrderedSet
  R : PolynomialCategory(K,E,OV)

Exports ==> with
  groebner: List R -> List R

Implementation ==> add
  if ~(K has FiniteFieldCategory) then
    GBPackR ==> GroebnerPackage(K,E,OV,R)
groebner(l)groebner(l)$GBPackR
  else
    q:PositiveInteger:=(characteristic()$K pretend PositiveInteger)
PF ==> PrimeField(q)
DPF ==> DistributedMultivariatePolynomial(symb,PF)
D ==> DistributedMultivariatePolynomial(symb,K)
JCFGBPack ==> GroebnerPackage(PF,E,OV,DPF)
GBPack ==> GroebnerPackage(K,E,OV,D)

coerceKtoPF: K -> PF
c coerceKtoPF(a:K):PF==
 index(lookup(a)$K)$PF

c coercerPFtoK: PF -> K
c coercerPFtoK(a:PF):K==
 index(lookup(a)$PF)$K

c coercerToDwithPF: R -> DPF
coerceToDwithPF(pol) ==
 map(coerceKtoPF(#1),pol)$MPolyCatFunctions2(OV,E,E,K,PF,R,DPF)

c coercerToRwithPF: DPF->R
coerceToRwithPF(pol) ==
 map(coercePFtoK(#1),pol)$MPolyCatFunctions2(OV,E,E,PF,K,DPF,R)

c coercerToD: R -> D
c coercerToD(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,K,D)

c coercerToR: D->R
c coercerToR(pol) == map(#1,pol)$MPolyCatFunctions2(OV,E,E,K,D,R)

gs:= size()$K
if gs = q and (representationType()$K case "prime") then
groebner(l)==
 ldmp:List DPF:= [coerceToDwithPF(pol) for pol in l]
 gg:=groebner(ldmp)$JCFGBPack
 [coerceToRwithPF(pol) for pol in gg]
else
groebner(l)==
 ldmp:List D:= [coerceToD(pol) for pol in l]
 gg:=groebner(ldmp)$GBPack
 [coerceToR(pol) for pol in gg]
Bug! Cannot precompute params and return a function which simply computes the last call. e.g. ridHack1, below.

Functions related to the binary representation of integers. These functions directly access the bits in the big integer representation and so are much faster than using a quotient loop.

\section{package INTBIT IntegerBits}

\set break resume
\sys rm -f IntegerBits.output
\spool IntegerBits.output
\set message test on
\set message auto off
\clear all

--S 1 of 1
\show IntegerBits

--R
--R IntegerBits is a package constructor
--R Abbreviation for IntegerBits is INTBIT
--R This constructor is not exposed in this frame.
--R Issue \texttt{)edit bookvol10.4.pamphlet} to see algebra source code for INTBIT
--R
--R--------------------------- Operations ---------------------------
--R bitLength : Integer \to Integer
--R bitCoef : (Integer,Integer) \to Integer
--R bitTruth : (Integer,Integer) \to Boolean
--R
--E 1

\spool
\lisp (bye)

---

\section{IntegerBits.help}

====================================================================
IntegerBits examples
====================================================================

This package provides functions to lookup bits in integers
See Also:
  o )show IntegerBits

---

IntegerBits (INTBIT)

Exports:
  bitLength  bitCoef  bitTruth

--- package INTBIT IntegerBits ---

)abbrev package INTBIT IntegerBits
++ Description:
++ This package provides functions to lookup bits in integers

IntegerBits: with
  -- bitLength(n) == # of bits to represent abs(n)
  -- bitCoef (n,i) == coef of 2**i in abs(n)
  -- bitTruth(n,i) == true if coef of 2**i in abs(n) is 1

  bitLength: Integer -> Integer
  ++ bitLength(n) returns the number of bits to represent abs(n)
  bitCoef: (Integer, Integer) -> Integer
  ++ bitCoef(n,m) returns the coefficient of 2**m in abs(n)
  bitTruth: (Integer, Integer) -> Boolean
  ++ bitTruth(n,m) returns true if coefficient of 2**m in abs(n) is 1

== add
  bitLength n == INTEGER_-LENGTH(n)$Lisp
  bitCoef (n,i) == if INTEGER_-BIT(n,i)$Lisp then 1 else 0
  bitTruth(n,i) == INTEGER_-BIT(n,i)$Lisp
package COMBINAT IntegerCombinatoricFunctions

)set break resume
)sys rm -f IntegerCombinatoricFunctions.output
)spool IntegerCombinatoricFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
)show IntegerCombinatoricFunctions
--R
--R IntegerCombinatoricFunctions(I: IntegerNumberSystem) is a package constructor
--R Abbreviation for IntegerCombinatoricFunctions is COMBINAT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for COMBINAT
--R
--R Operations -------------------------------
--R binomial : (I,I) -> I
--R factorial : I -> I
--R multinomial : (I,List(I)) -> I
--R partition : I -> I
--R permutation : (I,I) -> I
--R stirling1 : (I,I) -> I
--R stirling2 : (I,I) -> I
--R
--E 5

--S 2 of 5
)set expose add constructor OutputForm
--R
--I OutputForm is already explicitly exposed in frame frame0
--E 1

--S 3 of 5
pascalRow(n) == [right(binomial(n,i),4) for i in 0..n]
IntegerCombinatoricFunctions examples

IntegerCombinatoricFunctions package provides some standard functions in combinatorics.

The binomial(n, r) returns the number of subsets of r objects taken among n objects, i.e. n!/r!*(n-r)!

The binomial coefficients are the coefficients of the series expansion of a power of a binomial, that is

\[ \binom{n}{k} = \frac{n!}{k!(n-k)!} \]
\[ x = (1 + x) \]
\[ k = 0 \]

This leads to the famous Pascal triangle. First we expose the OutputForm domain, which is normally hidden, so we can use it to format the lines.

\[ )set expose add constructor OutputForm \]

Next we define a function that will output the list of binomial coefficients right justified with proper spacing:

\[ \text{pascalRow}(n) == [\text{right}(\text{binomial}(n,i),4) \text{ for } i \text{ in } 0..n] \]

and now we format the whole line so that it looks centered:

\[ \text{displayRow}(n) == \text{output center blankSeparate pascalRow}(n) \]

and we compute the triangle

\[ \text{for } i \text{ in } 0..7 \text{ repeat displayRow } i \]

giving the pretty result:

\[
\begin{array}{cccccc}
  & 1 & & & & \\
 1 & 1 & & & & \\
 1 & 2 & 1 & & & \\
 1 & 3 & 3 & 1 & & \\
 1 & 4 & 6 & 4 & 1 & \\
 1 & 5 & 10 & 10 & 5 & 1 \\
 1 & 6 & 15 & 20 & 15 & 6 & 1 \\
 1 & 7 & 21 & 35 & 35 & 21 & 7 & 1 \\
\end{array}
\]

See Also:
- \( )\text{show IntegerCombinatoricFunctions} \)
- \( )\text{d op binomial} \)
- \( )\text{show OutputForm} \)
- \( )\text{help set} \)
IntegerCombinatoricFunctions (COMBINAT)

Exports:
- binomial
- factorial
- multinomial
- partition
- permutation
- stirling1
- stirling2

— package COMBINAT IntegerCombinatoricFunctions —

)abbrev package COMBINAT IntegerCombinatoricFunctions
++ Authors: Martin Brock, Robert Sutor, Michael Monagan
++ Date Created: June 1987
++ Description:
++ The \spadtype{IntegerCombinatoricFunctions} package provides some
++ standard functions in combinatorics.

IntegerCombinatoricFunctions(I:IntegerNumberSystem): EE == II where
Z ==> Integer
N ==> NonNegativeInteger
SUP ==> SparseUnivariatePolynomial
EE ==>
with
  binomial: (I, I) -> I
  ++ \spad{binomial(n,r)} returns the binomial coefficient
  ++ \spad{C(n,r) = n!/((r! \cdot (n-r)!))}, where \spad{n >= r >= 0}.
  ++ This is the number of combinations of n objects taken r at a time.
  ++
  ++X [binomial(5,i) for i in 0..5]

factorial: I -> I
  ++ \spad{factorial(n)} returns \spad{n!}. this is the product of all
  ++ integers between 1 and n (inclusive).
  ++ Note that \spad{0!} is defined to be 1.

multinomial: (I, List I) -> I
  ++ \spad{multinomial(n,[m1,m2,...,mk])} returns the multinomial
  ++ coefficient \spad{n!/(m1! \cdot m2! \cdot ... \cdot mk)!}.

partition: I -> I
  ++ \spad{partition(n)} returns the number of partitions of the integer n.
  ++ This is the number of distinct ways that n can be written as
++ a sum of positive integers.

permutation: (I, I) -> I
++ \text{\texttt{permutation(n)}} returns \texttt{!P(n,r) = n!/(n-r)!}. This is
++ the number of permutations of \(n\) objects taken \(r\) at a time.

stirling1: (I, I) -> I
++ \text{\texttt{stirling1(n,m)}} returns the Stirling number of the first kind
++ denoted \texttt{S[n,m]}.

stirling2: (I, I) -> I
++ \text{\texttt{stirling2(n,m)}} returns the Stirling number of the second kind
++ denoted \texttt{SS[n,m]}.

II ==>
add
F : Record(Fn:I, Fv:I) := [0,1]
B : Record(Bn:I, Bm:I, Bv:I) := [0,0,0]
S : Record(Sn:I, Sp:SUP I) := [0,0]
P : IndexedFlexibleArray(I,0) := new(1,1)$IndexedFlexibleArray(I,0)

partition n ==
-- This is the number of ways of expressing \(n\) as a sum of positive
-- integers, without regard to order. For example partition 5 = 7
-- since 5 = 1+1+1+1+1 = 1+1+1+2 = 1+1+3 = 1+4 = 2+3 = 5 .
-- Uses O(\sqrt{n}) term recurrence from Abramovitz & Stegun pp. 825
-- p(n) = sum (-1)**k p(n-j) where 0 < j := (3k**2+-k) quo 2 <= n
-- minIndex(P) ^= 0 => error "Partition: must have minIndex of 0"

m := #P
n < 0 => error "partition is not defined for negative integers"

n < m::I => P(convert(n)@Z)
for i in m..convert(n)@Z repeat
s:I := 1
t:I := 0
for k in 1.. repeat
l := (3*k+k-k) quo 2
l > i => leave
u := l+k
t := t + s * P(convert(i-l)@Z)
u > i => leave
t := t + s * P(convert(i-u)@Z)
s := -s
P.i := t
P(convert(n)@Z)

factorial n ==
s,f,t : I
n < 0 => error "factorial not defined for negative integers"
if n <= F.Fn then s := f := 1 else (s, f) := F
for k in convert(s+1)@Z .. convert(n)@Z by 2 repeat
if k::I = n then t := n else t := k::I * (k+1)::I
f := t * f
F.Fn := n
F.Fv := f

binomial(n, m) ==
  s,b:I
  n < 0 or m < 0 or m > n => 0
  m = 0 => 1
  n < 2*m => binomial(n, n-m)
  (s,b) := (0,1)
  if B.Bn = n then
    B.Bm = m+1 =>
      b := (B.Bv * (m+1)) quo (n-m)
      B.Bn := n
      B.Bm := m
      return(B.Bv := b)
    if m >= B.Bm then (s := B.Bm; b := B.Bv) else (s,b) := (0,1)
    for k in convert(s+1)@Z .. convert(m)@Z repeat
      b := (b*(n-k::I+1)) quo k::I
    B.Bn := n
    B.Bm := m
    B.Bv := b
  else
    multinomial(n, m) ==
      for t in m repeat t < 0 => return 0
      n < _+/m => 0
      s:I := 1
      for t in m repeat s := s * factorial t
      factorial n quo s

permutation(n, m) ==
  t:I
  m < 0 or n < m => 0
  m := n-m
  p:I := 1
  for k in convert(m+1)@Z .. convert(n)@Z by 2 repeat
    if k::I = n then t := n else t := (k*(k+1))::I
    p := p * t
  p

stirling1(n, m) ==
  -- Definition: (-1)**(n-m) S[n,m] is the number of
  -- permutations of n symbols which have m cycles.
  n < 0 or m < 1 or m > n => 0
  m = n => 1
  S.Sn = n => coefficient(S.Sp, convert(m)@Z :: N)
  x := monomial(1, 1)$SUP(I)
  S.Sn := n
  S.Sp := x
  for k in 1 .. convert(n-1)@Z repeat S.Sp := S.Sp * (x - k::SUP(I))
  coefficient(S.Sp, convert(m)@Z :: N)
stirling2(n, m) ==
    -- definition: SS[n,m] is the number of ways of partitioning
    -- a set of n elements into m non-empty subsets
    n < 0 or m < 1 or m > n => 0
    m = 1 or n = m => 1
    s:I := if odd? m then -1 else 1
    t:I := 0
    for k in 1..convert(m)@Z repeat
        s := -s
        t := t + s * binomial(m, k::I) * k::I ** (convert(n)@Z :: N)
    t quo factorial m

———

— COMBINAT.dotabb —

"COMBINAT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=COMBINAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"COMBINAT" -> "A1AGG"

———

package INTFACT IntegerFactorizationPackage

— IntegerFactorizationPackage.input —

)set break resume
)sys rm -f IntegerFactorizationPackage.output
)spool IntegerFactorizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerFactorizationPackage
--R
--R IntegerFactorizationPackage(I: IntegerNumberSystem) is a package constructor
--R Abbreviation for IntegerFactorizationPackage is INTFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTFACT
--R
--R----------------------------- Operations -----------------------------
--R BasicMethod : I -> Factored(I) factor : I -> Factored(I)
--R squareFree : I -> Factored(I)
This Package contains basic methods for integer factorization. The factor operation employs trial division up to 10,000. It then tests to see if n is a perfect power before using Pollard's rho method. Because Pollard's method may fail, the result of factor may contain composite factors. We should also employ Lenstra's elliptic curve method.

See Also:
 o )show IntegerFactorizationPackage

IntegerFactorizationPackage (INTFACT)

Exports:
 BasicMethod  factor  squareFree  PollardSmallFactor

— package INTFACT IntegerFactorizationPackage —
chapter 10. chapter i

)abbrev package INTFACT IntegerFactorizationPackage
++ Description:
++ This Package contains basic methods for integer factorization.
++ The factor operation employs trial division up to 10,000. It
++ then tests to see if n is a perfect power before using Pollard's
++ rho method. Because Pollard's method may fail, the result
++ of factor may contain composite factors. We should also employ
++ Lenstra's elliptic curve method.

IntegerFactorizationPackage(I): Exports == Implementation where
  I: IntegerNumberSystem
  B ==> Boolean
  FF ==> Factored I
  NNI ==> NonNegativeInteger
  LMI ==> ListMultiDictionary I
  FFE ==> Record(flg:Union("nil","sqfr","irred","prime"),
  fctr:I, xpnt:Integer)

Exports == with
  factor : I -> FF
    ++ factor(n) returns the full factorization of integer n
  squareFree : I -> FF
    ++ squareFree(n) returns the square free factorization of integer n
  BasicMethod : I -> FF
    ++ BasicMethod(n) returns the factorization
    ++ of integer n by trial division
  PollardSmallFactor: I -> Union(I,"failed")
    ++ PollardSmallFactor(n) returns a factor
    ++ of n or "failed" if no one is found

Implementation == add
  import IntegerRoots(I)

  BasicSieve: (I, I) -> FF

squareFree

— package INTFACT IntegerFactorizationPackage —

squareFree(n:I):FF ==
  u:I
  if n<0 then (m := -n; u := -1)
  else (m := n; u := 1)
  (m > i) and ((v := perfectSqrt m) case I) =>
for rec in (l := factorList(sv := squareFree(v::I))) repeat
  rec.xpnt := 2 * rec.xpnt
makeFR(u * unit sv, 1)

-- avoid using basic sieve when the lim is too big
-- we know the sieve constants up to sqrt(100000000)
lim := 1 + approxSqrt(m)
lim > (100000000::I) => makeFR(u, factorList factor m)
x := BasicSieve(m, lim)
y :=
((m:= unit x) = 1) => factorList x
(v := perfectSqrt m) case I =>
  concat!(factorList x, ["sqfr",v,2]$FFE)
  concat!(factorList x, ["sqfr",m,1]$FFE)
makeFR(u, y)

PollardSmallFactor

This is Brent's optimization of Pollard's rho factoring. Brent’s algorithm is about 24 percent faster than Pollard’s. Pollard's algorithm has complexity $O(p^{1/2})$ where $p$ is the smallest prime factor of the composite number $N$.

Pollard’s idea is based on the observation that two numbers $x$ and $y$ are congruent modulo $p$ with probability $0.5$ after $1.177 * \sqrt{p}$ numbers have been randomly chosen. If we try to factor $n$ and $p$ is a factor of $n$, then

$$1 < gcd(|x - y|, n) \leq n$$

since $p$ divides both $|x - y|$ and $n$.

Given a function $f$ which generates a pseudo-random sequence of numbers we allow $x$ to walk the sequence in order and $y$ to walk the sequence at twice the rate. At each cycle we compute $gcd(|x - y|, n)$. If this GCD ever equals $n$ then $x = y$ which means that we have walked "all the way around the pseudo-random cycle" and we terminate with failure.

This algorithm returns failure on all primes but also fails on some composite numbers.

Quoting Brent’s back-tracking idea:

The best-known algorithm for finding GCDs is the Euclidean algorithm which takes $O(\log N)$ times as long as one multiplication mod $N$. Pollard showed that most of the GCD computations in Floyd’s algorithm could be dispensed with.

... The idea is simple: if $P_F$ computes $GCD(z_1, N)$, $GCD(z_2, N)$, ..., then we compute

$$q_i = \prod_{j=1}^{i} z_j (\text{mod } N)$$
and only compute $GCD(q_i, N)$ when $i$ is a multiple of $m$, where $\log N << m << N^{1/4}$. Since $q_{i+1} = q_i \times z_{i+1} \mod N$, the work required for each GCD computation in algorithm $P_F$ is effectively reduced to that for a multiplication $\mod N$ in the modified algorithm. The probability of the algorithm failing because $q_i = 0$ increases, so it is best not to choose $m$ too large. This problem can be minimized by backtracking to the state after the previous GCD computation and setting $m = 1$.

Brent incorporates back-tracking, omits the random choice of $u$, and makes some minor modifications. His algorithm (p192-183) reads:

```plaintext
y := x_0; r := 1; q := 1;
repeat x := y;
  for i := 1 to r do y := f(y); k := 0;
  repeat ys := y;
    for i := 1 to min(m, r - k) do
      begin y := f(y); q := q \times |x - y| \mod N
    end;
    G := GCD(q, N); k := k + m
  until (k \geq r) or (G > 1); r := 2 \times r
until G > 1;
if G = N then
  repeat ys := f(ys); G := GCD(|y - yx|, N)
  until G > 1;
if G = N then failure else success
```

Here we use the function 

\[(y \times y + 5 : I) \mod n\]

as our pseudo-random sequence with a random starting value for $y$.

On possible optimization to explore is to keep a hash table for the computed values of the function $y_{i+1} := f(y_i)$ since we effectively walk the sequence several times. And we walk the sequence in a loop many times. But because we are generating a very large number of numbers the array can be a simple array of fixed size that captures the last $n$ values. So if we make a fixed array $F$ of, say $2^n$ elements we can store $f(y_i)$ in $F[y_i \mod 2^n]$.

One property that this algorithm assumes is that the function used to generate the numbers has a long, hopefully complete, period. It is not clear that the recommended function has that property.

--- package INTFACT IntegerFactorizationPackage ---

```plaintext
PollardSmallFactor(n:I):Union(I,"failed") ==
```

---
-- Use the Brent variation
x0 := random()$I
m := 100::I
y := x0 rem n
r:I := 1
q:I := 1
G:I := 1
until G > 1 repeat
  x := y
  for i in 1..convert(r)@Integer repeat
    y := (y*y+5::I) rem n
    k:I := 0
    until (k>=r) or (G>1) repeat
      ys := y
      for i in 1..convert(min(m,r-k))@Integer repeat
        y := (y*y+5::I) rem n
        q := q*abs(x-y) rem n
        G := gcd(q,n)
        k := k+m
      r := 2*r
    if G=n then
      until G>1 repeat
        ys := (ys*ys+5::I) rem n
        G := gcd(abs(x-ys),n)
        G=n => "failed"
      G

BasicSieve

We create a list of prime numbers up to the limit given. The prior code used a circular list but tests of that list show that on average more than 50% of the required prime numbers. Overall this is a small percentage of the time needed to factor.

This loop uses three pieces of information

1. n which is the number we are testing
2. d which is the current prime to test
3. lim which is the upper limit of the primes to test

We loop d over the list of primes. If the remaining number n is smaller than the square of d then n must be prime and if it is not one, we add it to the list of primes. If the remaining number is larger than the square of d we remove all factors of d, reducing n each time. Then we add a record of the new factor and its multiplicity, m. We continue the loop until we run out of primes.
Annoyingly enough, primes does not return an ordered list so we fix this.
The sieve works up to a given limit, reducing out the factors that it finds. If it can find all
of the factors than it returns a factored result where the first element is the unit 1. If there
is still a part of the number unfactored it returns the number and a list of the factors found
and their multiplicity.

Basically we just loop thru the prime factors checking to see if they are a component of the
number, n. If so, we remove the factor from the number n (possibly m times) and continue
thru the list of primes.

— package INTFACT IntegerFactorizationPackage —

BasicSieve(n, lim) ==
 p:=primes(1::I,lim::I)$IntegerPrimesPackage(I)
l:List(I) := append([first p],reverse rest p)
ls := empty()$List(FFE)
for d in l repeat
  if n<d*d then
    if n>1 then ls := concat_!(ls, "prime",n,1]$FFE)
    return makeFR(1, ls)
  for m in 0.. while zero?(n rem d) repeat n := n quo d
  if m>0 then ls := concat_!(ls, "prime",d,convert m]$FFE)
  makeFR(n,ls)

——

BasicMethod

— package INTFACT IntegerFactorizationPackage —

BasicMethod n ==
  u:I
  if n<0 then (m := -n; u := -1)
  else (m := n; u := 1)
  x := BasicSieve(m, 1 + approxSqrt m)
  makeFR(u, factorList x)

——

factor

The factor function is many orders of magnitude slower than the results of other systems. A
posting on sci.math.symbolic showed that NTL could factor the final value (t6) in about 11
seconds. Axiom takes about 8 hours.
a1:=101
a2:=109
t1:=a1*a2
factor t1

a3:=21525175387
t2:=t1*a3
factor t2

a4:=218301576858349
t3:=t2*a4
factor t3

a5:=13731482973783137
t4:=t3*a5
factor t4

a6:=23326138687706820109
t5:=t4*a6
factor t5

a7:=4328240801173188438252813716944518369161
t6:=t5*a7
factor t6

--- package INTFACT IntegerFactorizationPackage ---

factor m ==
  u:I
  zero? m => 0
  if negative? m then (n := -m; u := -1)
    else (n := m; u := 1)
  b := BasicSieve(n, 10000::I)
  flb := factorList b
  ((n := unit b) = 1) => makeFR(u, flb)
  a:LMI := dictionary() -- numbers yet to be factored
  b:LMI := dictionary() -- prime factors found
  f:LMI := dictionary() -- number which could not be factored
  insert_!(n, a)
  while not empty? a repeat
    n := inspect a; c := count(n, a); remove_!(n, a)
    prime?(n)$IntegerPrimesPackage(I) => insert_!(n, b, c)
    -- test for a perfect power
    (s := perfectNthRoot n).exponent > 1 =>
      insert_!(s.base, a, c * s.exponent)
    -- test for a difference of square
    x:=approxSqrt n
    if (x**2<n) then x:=x+1
    (y:=perfectSqrt (x**2-n)) case I =>
insert!(x+y,a,c)
insert!(x-y,a,c)
(d := PollardSmallFactor n) case I =>
  for m in 0.. while zero?(n rem d) repeat n := n quo d
  insert!(d, a, m * c)
  if n > 1 then insert!(n, a, c)
-- an elliptic curve factorization attempt should be made here
insert!(n, f, c)
-- insert prime factors found
while not empty? b repeat
  n := inspect b; c := count(n, b); remove_!(n, b)
  flb := concat_!(flb, ["prime",n,convert c]$FFE)
-- insert non-prime factors found
while not empty? f repeat
  n := inspect f; c := count(n, f); remove_!(n, f)
  flb := concat_!(flb, ["nil",n,convert c]$FFE)
makeFR(u, flb)

package ZLINDEP IntegerLinearDependence

-- IntegerLinearDependence.input --
)set break resume
)spool IntegerLinearDependence.output
)set message test on
)set message auto off
)clear all
--S 1 of 9
M := SQMATRIX(2,INT)
--R
--R
--R (1) SquareMatrix(2,Integer)
--E 1
--S 2 of 9
m1: M := squareMatrix matrix [ [1, 2], [0, -1] ]
--R
--R
--R +1 2 +
--R (2) | |
--R +0 - 1+
--R
Type: SquareMatrix(2,Integer)
--E 2

--S 3 of 9
m2: M := squareMatrix matrix [ [2, 3], [1, -2] ]
--R
--R
--R +2 3 +
--R (3) | |
--R +1 - 2+
--R
Type: SquareMatrix(2,Integer)
--E 3

--S 4 of 9
m3: M := squareMatrix matrix [ [3, 4], [2, -3] ]
--R
--R
--R +3 4 +
--R (4) | |
--R +2 - 3+
--R
Type: SquareMatrix(2,Integer)
--E 4

--S 5 of 9
linearlyDependentOverZ? vector [m1, m2, m3]
--R
--R
--R (5) true
--R
Type: Boolean
--E 5

--S 6 of 9
c := linearDependenceOverZ vector [m1, m2, m3]
--R
--R
--R (6) [1,- 2,1]
--R
Type: Union(Vector(Integer),...)
--E 6

--S 7 of 9
c.1 * m1 + c.2 * m2 + c.3 * m3
--R
--R Type: SquareMatrix(2,Integer)
--E 7

solveLinearlyOverQ(vector [m1, m3], m2)
--R
--R 1 1
--R Type: Union(Vector(Fraction(Integer)),...)
--E 8

)show IntegerLinearDependence
--R
--R IntegerLinearDependence(R: LinearlyExplicitRingOver(Integer)) is a package constructor
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ZLINDEP
--R
--R linearDependenceOverZ : Vector(R) -> Union(Vector(Integer),"failed")
--R linearlyDependentOverZ? : Vector(R) -> Boolean
--R solveLinearlyOverQ : (Vector(R),R) -> Union(Vector(Fraction(Integer)),"failed")
--E 9

)spool
)lisp (bye)

---

IntegerLinearDependence.help ---

This packages tests for linear dependence over the integers.

The elements v1,...,vN of a module M over a ring R are said to be linearly dependent over R if there exist c1,...,cN in R, not all 0, such that c1 v1 + ... cN vN = 0. If such ci's exist, they form
what is called a linear dependence relation over \( \mathbb{R} \) for the \( v_i \)'s.

The package IntegerLinearDependence provides functions for testing whether some elements of a module over the integers are linearly dependent over the integers, and to find the linear dependence relations, if any.

Consider the domain of two by two square matrices with integer entries.

```
M := SQMATRIX(2,INT)
  SquareMatrix(2,Integer)
  Type: Domain

Now create three such matrices.

m1: M := squareMatrix matrix [ [1, 2], [0, -1] ]
  +1  2 +
  |   |
  +0  - 1+
  Type: SquareMatrix(2,Integer)

m2: M := squareMatrix matrix [ [2, 3], [1, -2] ]
  +2  3 +
  |   |
  +1  - 2+
  Type: SquareMatrix(2,Integer)

m3: M := squareMatrix matrix [ [3, 4], [2, -3] ]
  +3  4 +
  |   |
  +2  - 3+
  Type: SquareMatrix(2,Integer)
```

This tells you whether \( m_1, m_2 \) and \( m_3 \) are linearly dependent over the integers.

```
linearlyDependentOverZ? vector [m1, m2, m3]
  true
  Type: Boolean
```

Since they are linearly dependent, you can ask for the dependence relation.

```
c := linearDependenceOverZ vector [m1, m2, m3]
  [1,- 2,1]
  Type: Union(Vector Integer,...)
```

This means that the following linear combination should be 0.

```
c.1 * m1 + c.2 * m2 + c.3 * m3
  +0  0+
  |   |
```
When a given set of elements are linearly dependent over \( \mathbb{R} \), this also means that at least one of them can be rewritten as a linear combination of the others with coefficients in the quotient field of \( \mathbb{R} \).

To express a given element in terms of other elements, use the operation `solveLinearlyOverQ`.

```
solveLinearlyOverQ(vector [m1, m3], m2)
  1 1
  [-,-]
  2 2
```

Type: Union(Vector Fraction Integer,...)

See Also:
- `)show IntegerLinearDependence`

---

**IntegerLinearDependence (ZLINDEP)**

Exports:
- `linearDependenceOverZ`
- `linearlyDependentOverZ?`
- `solveLinearlyOverQ`

> package ZLINDEP IntegerLinearDependence

)abbrev package ZLINDEP IntegerLinearDependence
++ Author: Manuel Bronstein
++ Date Last Updated: 14 May 1991
++ Description:
++ Test for linear dependence over the integers.
IntegerLinearDependence(R): Exports == Implementation where
R: LinearlyExplicitRingOver Integer
Z ==> Integer
Exports ==> with
linearlyDependentOverZ?: Vector R -> Boolean
++ \spad{linearlyDependentOverZ?([v1,...,vn])} returns true if the
++ vi's are linearly dependent over the integers, false otherwise.
linearDependenceOverZ : Vector R -> Union(Vector Z, "failed")
++ \spad{linearDependenceOverZ([v1,...,vn])} returns
++ \spad{[c1,...,cn]} if
++ \spad{c1*v1 + ... + cn*vn = 0} and not all the ci's are 0, "failed"
++ if the vi's are linearly independent over the integers.
solveLinearlyOverQ : (Vector R, R) ->
++ \spad{solveLinearlyOverQ([v1,...,vn], u)} returns \spad{[c1,...,cn]}
++ such that \spad{c1*v1 + ... + cn*vn = u},
++ "failed" if no such rational numbers ci's exist.

Implementation ==> add
import LinearDependence(Z, R)

linearlyDependentOverZ? v == linearlyDependent? v
linearDependenceOverZ v == linearDependence v
solveLinearlyOverQ(v, c) == solveLinear(v, c)

—— ZLINDEP.dotabb ——

"ZLINDEP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ZLINDEP"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ZLINDEP" -> "PID"
"ZLINDEP" -> "OAGROUP"

——

package INTHEORY IntegerNumberTheoryFunctions

—— IntegerNumberTheoryFunctions.input ——

)set break resume
```plaintext
--S 1 of 31
div144 := divisors(144)
--R
--R (1) [1,2,3,4,6,8,9,12,16,18,24,36,48,72,144]
--R Type: List(Integer)
--E 1

--S 2 of 31
#(div144)
--R
--R (2) 15
--R Type: PositiveInteger
--E 2

--S 3 of 31
reduce(+,div144)
--R
--R (3) 403
--R Type: PositiveInteger
--E 3

--S 4 of 31
numberOfDivisors(144)
--R
--R (4) 15
--R Type: PositiveInteger
--E 4

--S 5 of 31
sumOfDivisors(144)
--R
--R (5) 403
--R Type: PositiveInteger
--E 5

--S 6 of 31
f1(n)==reduce(+,[moebiusMu(d)*numberOfDivisors(quo(n,d)) for d in divisors(n)])
--R
--R Type: Void
```
--E 6

--S 7 of 31
f1(200)
--R
--R Compiling function f1 with type PositiveInteger -> Integer
--R
--R (7) 1
--R Type: PositiveInteger
--E 7

--S 8 of 31
f1(846)
--R
--R (8) 1
--R Type: PositiveInteger
--E 8

--S 9 of 31
f2(n) == reduce(+,[moebiusMu(d) * sumOfDivisors(quo(n,d))_
for d in divisors(n)])
--R
--R Type: Void
--E 9

--S 10 of 31
f2(200)
--R
--R Compiling function f2 with type PositiveInteger -> Integer
--R
--R (10) 200
--R Type: PositiveInteger
--E 10

--S 11 of 31
f2(846)
--R
--R (11) 846
--R Type: PositiveInteger
--E 11

--S 12 of 31
fibonacci(25)
--R
--R (12) 75025
--R Type: PositiveInteger
--E 12
[fibonacci(n) for n in 1..15]

(13) [1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]

Type: List(Integer)

fib(n) == reduce(+,[binomial(n-1-k,k) for k in 0..quo(n-1,2)])

fib(25)

Compiling function fib with type PositiveInteger -> Integer

(15) 75025

Type: PositiveInteger

[h(d) == quo(reduce(+,[jacobi(d,k) for k in 1..quo(-d, 2)]),2-jacobi(d,2))]
--R
--R Type: Void
--E 19

--S 20 of 31
h(-163)
--R
--R Compiling function h with type Integer -> Integer
--R
--R (20) 1
--R Type: PositiveInteger
--E 20

--S 21 of 31
h(-499)
--R
--R (21) 3
--R Type: PositiveInteger
--E 21

--S 22 of 31
h(-1832)
--R
--R (22) 26
--R Type: PositiveInteger
--E 22

--S 23 of 31
inverse:(INT,INT)->INT
--R
--R Type: Void
--E 23

--S 24 of 31
inverse(a,b) ==
  borg:INT:=b
  c1:INT := 1
  d1:INT := 0
  while b "= 0 repeat
    q := a quo b
    r := a-q*b
    print [a, "=" q, "*" b, "+", r]
    (a,b):=(b,r)
    (c1,d1):=(d1,c1-q*d1)
  a ^= 1 => error("moduli are not relatively prime")
  positiveRemainder(c1,borg)
--R
--R Type: Void
inverse(15, 26)
Compiling function inverse with type (Integer, Integer) -> Integer
[15, "\=" , 0 , "\*(-", 26 , ")\+", 15]
[26, "\=" , 1 , "\*(-", 15 , ")\+", 11]
[15, "\=" , 1 , "\*(-", 11 , ")\+", 4]
[11, "\=" , 2 , "\*(-", 4 , ")\+", 3]
[4, "\=" , 1 , "\*(-", 3 , ")\+", 1]
[3, "\=" , 3 , "\*(-", 1 , ")\+", 0]

(25) 7
Type: PositiveInteger

x1:=4

(26) 4
Type: PositiveInteger

m1:=5

(27) 5
Type: PositiveInteger

x2:=2

(28) 2
Type: PositiveInteger

m2:=3

(29) 3
Type: PositiveInteger
result:=chineseRemainder(x1,m1,x2,m2)
--R
--R (30) 14
--R Type: PositiveInteger
--E 30

--S 31 of 31
)show IntegerNumberTheoryFunctions
--R
--R IntegerNumberTheoryFunctions is a package constructor
--R Abbreviation for IntegerNumberTheoryFunctions is INTHEORY
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTHEORY
--R
--R--------------------------------- Operations ---------------------------------
--R divisors : Integer -> List(Integer) euler : Integer -> Integer
--R eulerPhi : Integer -> Integer fibonacci : Integer -> Integer
--R moebiusMu : Integer -> Integer sumOfDivisors : Integer -> Integer
--R bernoulli : Integer -> Fraction(Integer) chineseRemainder : (Integer,Integer,Integer,Integer) -> Integer
--R harmonic : Integer -> Fraction(Integer) jacobi : (Integer,Integer) -> Integer
--R legendre : (Integer,Integer) -> Integer numberOfDivisors : Integer -> Integer
--R sumOfKthPowerDivisors : (Integer,NonNegativeInteger) -> Integer
--R
--E 31

)spool
)lisp (bye)

---

--- IntegerNumberTheoryFunctions.help ---

====================================================================
IntegerNumberTheoryFunctions examples
====================================================================

This package provides various number theoretic functions on the integers.

The IntegerNumberTheoryFunctions package contains a variety of operations of interest to number theorists. Many of these operations deal with divisibility properties of integers. (Recall that an integer \(a\) divides an integer \(b\) if there is an integer \(c\) such that \(b = a \cdot c\).)

The operation divisors returns a list of the divisors of an integer.
\begin{verbatim}

div144 := divisors(144)
[1,2,3,4,6,8,9,12,16,18,24,36,48,72,144]
Type: List Integer

You can now compute the number of divisors of 144 and the sum of the divisors of 144 by counting and summing the elements of the list we just created.

#(div144)
15
Type: PositiveInteger

reduce(+,div144)
403
Type: PositiveInteger

Of course, you can compute the number of divisors of an integer \( n \), usually denoted \( d(n) \), and the sum of the divisors of an integer \( n \), usually denoted \( \sigma(n) \), without ever listing the divisors of \( n \).

In Axiom, you can simply call the operations numberOfDivisors and sumOfDivisors.

numberOfDivisors(144)
15
Type: PositiveInteger

sumOfDivisors(144)
403
Type: PositiveInteger

The key is that \( d(n) \) and \( \sigma(n) \) are "multiplicative functions." This means that when \( n \) and \( m \) are relatively prime, that is, when \( n \) and \( m \) have no prime factor in common, then \( d(nm) = d(n) d(m) \) and \( \sigma(nm) = \sigma(n) \sigma(m) \). Note that these functions are trivial to compute when \( n \) is a prime power and are computed for general \( n \) from the prime factorization of \( n \). Other examples of multiplicative functions are \( \sigma_k(n) \), the sum of the \( k \)-th powers of the divisors of \( n \) and \( \varphi(n) \), the number of integers between 1 and \( n \) which are prime to \( n \). The corresponding Axiom operations are called sumOfKthPowerDivisors and eulerPhi.

An interesting function is \( \mu(n) \), the Moebius mu function, defined as follows: \( \mu(1) = 1 \), \( \mu(n) = 0 \), when \( n \) is divisible by a square, and \( \mu = (-1)^k \), when \( n \) is the product of \( k \) distinct primes. The corresponding Axiom operation is called moebiusMu. This function occurs in the following theorem:

Theorem: (Moebius Inversion Formula):
Let \( f(n) \) be a function on the positive integers and let \( F(n) \) be defined by
\end{verbatim}
\[
F(n) = \sum_{d \mid n} f(n)
\]
the sum of \(f(n)\) over \(d \mid n\) where the sum is taken over the positive divisors of \(n\). Then the values of \(f(n)\) can be recovered from the values of \(F(n)\): 
\[
f(n) = \sum_{d \mid n} \mu(n) F(n/d)
\]
where again the sum is taken over the positive divisors of \(n\).

When \(f(n) = 1\), then \(F(n) = d(n)\). Thus, if you sum \(\mu(d) \cdot d(n/d)\) over the positive divisors \(d\) of \(n\), you should always get 1.

\[
f1(n) := \text{reduce}(\ast, [\mu(d) \cdot \text{numberOfDivisors}(\text{quo}(n,d)) \mid d \in \text{divisors}(n)])
\]
Type: Void

\[
f1(200)
\]
1
Type: PositiveInteger

\[
f1(846)
\]
1
Type: PositiveInteger

Similarly, when \(f(n) = n\), then \(F(n) = \sigma(n)\). Thus, if you sum \(\mu(d) \cdot \sigma(n/d)\) over the positive divisors \(d\) of \(n\), you should always get \(n\).

\[
f2(n) := \text{reduce}(\ast, [\mu(d) \cdot \text{sumOfDivisors}(\text{quo}(n,d)) \mid d \in \text{divisors}(n)])
\]
Type: Void

\[
f2(200)
\]
200
Type: PositiveInteger

\[
f2(846)
\]
846
Type: PositiveInteger

The Fibonacci numbers are defined by
\[
F(1) = F(2) = 1 \quad \text{and} \quad F(n) = F(n-1) + F(n-2) \quad \text{for} \quad n = 3, 4, \ldots
\]

The operation fibonacci computes the \(n\)-th Fibonacci number.

\[
\text{fibonacci}(25)
\]
75025
Type: PositiveInteger

\[
[\text{fibonacci}(n) \text{ for } n \text{ in } 1..15]
\]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
Type: List Integer
Fibonacci numbers can also be expressed as sums of binomial coefficients.

\[
\text{fib}(n) = \text{reduce}(+, [\text{binomial}(n-1-k,k) \text{ for } k \text{ in } 0..\text{quo}(n-1,2)])
\]

\[
\text{fib}(25) = 75025
\]

\[
[\text{fib}(n) \text{ for } n \text{ in } 1..15] = [1,1,2,3,5,8,13,21,34,55,89,144,233,377,610]
\]

Quadratic symbols can be computed with the operations \text{legendre} and \text{jacobi}. The Legendre symbol \(a/p\) is defined for integers \(a\) and \(p\) with \(p\) an odd prime number. By definition,

- \((a/p) = +1\), when \(a\) is a square \(\text{(mod } p)\),
- \((a/p) = -1\), when \(a\) is not a square \(\text{(mod } p)\), and
- \((a/p) = 0\), when \(a\) is divisible by \(p\).

You compute \((a/p)\) via the command \text{legendre}(a,p).

\[
\text{legendre}(3,5) = -1
\]

\[
\text{legendre}(23,691) = -1
\]

The Jacobi symbol \((a/n)\) is the usual extension of the Legendre symbol, where \(n\) is an arbitrary integer. The most important property of the Jacobi symbol is the following: if \(K\) is a quadratic field with discriminant \(d\) and quadratic character \(\chi\), then \(\chi(n) = (d/n)\). Thus, you can use the Jacobi symbol to compute, say, the class numbers of imaginary quadratic fields from a standard class number formula.

This function computes the class number of the imaginary quadratic field with discriminant \(d\).

\[
\text{h}(d) = \text{quotient}(\text{reduce}(+, [\text{jacobi}(d,k) \text{ for } k \text{ in } 1..\text{quo}(-d,2)]), 2-\text{jacobi}(d,2))
\]

\[
\text{h}(-163) = 1
\]

\[
\text{h}(-499) = 3
\]
The inverse function is derived from the Extended Euclidean Algorithm. If we divide one integer by another nonzero integer we get an integer quotient plus a remainder which is, in general, a rational number. For instance, 
\[13/5 = 2 + 3/5\]
where 2 is the quotient and 3/5 is the remainder.

If we multiply thru by the denominator of the remainder we get an answer in integer terms which no longer involves division:
\[13 = 2(5) + 3\]

This gives a method of dividing integers. Specifically, if \(a\) and \(b\) are positive integers, there exist unique non-negative integers \(q\) and \(r\) so that
\[a = qb + r\]
where \(0 \leq r < b\)

\(q\) is called the quotient and \(r\) the remainder.

The greatest common divisor of integers \(a\) and \(b\), denoted by \(\text{gcd}(a,b)\), is the largest integer that divides (without remainder) both \(a\) and \(b\). So, for example:
\[
\begin{align*}
\text{gcd}(15, 5) &= 5, \\
\text{gcd}(7, 9) &= 1, \\
\text{gcd}(12, 9) &= 3, \\
\text{gcd}(81, 57) &= 3.
\end{align*}
\]

The gcd of two integers can be found by repeated application of the division algorithm, this is known as the Euclidean Algorithm. You repeatedly divide the divisor by the remainder until the remainder is 0. The gcd is the last non-zero remainder in this algorithm. The following example shows the algorithm.

Finding the gcd of 81 and 57 by the Euclidean Algorithm:
\[
\begin{align*}
81 &= 1(57) + 24 \\
57 &= 2(24) + 9 \\
24 &= 2(9) + 6 \\
9 &= 1(6) + 3 \\
6 &= 2(3) + 0
\end{align*}
\]

So the greatest common divisor, the GCD(81,51)=3.
If the \( \gcd(a, b) = r \) then there exist integers \( s \) and \( t \) so that

\[
 s(a) + t(b) = r
\]

By back substitution in the steps in the Euclidean Algorithm, it is possible to find these integers \( s \) and \( t \). We shall do this with the above example:

Starting with the next to last line, we have:

\[
3 = 9 - 1(6)
\]

From the line before that, we see that \( 6 = 24 - 2(9) \), so:

\[
3 = 9 - 1(24 - 2(9)) = 3(9) - 1(24)
\]

From the line before that, we have \( 9 = 57 - 2(24) \), so:

\[
3 = 3(57 - 2(24)) - 1(24) = 3(57) - 7(24)
\]

And, from the line before that \( 24 = 81 - 1(57) \), giving us:

\[
3 = 3(57) - 7(81 - 1(57)) = 10(57) - 7(81)
\]

So we have found \( s = -7 \) and \( t = 10 \).

The Extended Euclidean Algorithm computes the \( \gcd(a,b) \) and the values for \( s \) and \( t \).

Suppose we were doing arithmetics modulo 26 and we needed to find the inverse of a number mod 26. This turned out to be a difficult task (and not always possible). We observed that a number \( x \) had an inverse mod 26 (i.e., a number \( y \) so that \( xy \equiv 1 \mod 26 \)) if and only if \( \gcd(x,26) = 1 \).

In the general case the inverse of \( x \) exists if and only if \( \gcd(x, n) = 1 \) and if it exists then there exist integers \( s \) and \( t \) so that

\[
 sx + tn = 1
\]

But this says that \( sx = 1 + (-t)n \), or in other words,

\[
 sx \equiv 1 \mod n
\]

So, \( s \) (reduced mod \( n \) if need be) is the inverse of \( x \mod n \).

The extended Euclidean algorithm calculates \( s \) efficiently.

====================================================================

Finding the inverse mod \( n \)

====================================================================
We will number the steps of the Euclidean algorithm starting with step 0. The quotient obtained at step $i$ will be denoted by $q_i$ and an auxiliary number, $s_i$. For the first two steps, the value of this number is given:

$s(0) = 0$ and $s(1) = 1$.

For the remainder of the steps, we recursively calculate

$$s(i) = s(i-2) - s(i-1) q(i-2) \mod n$$

The algorithm starts by "dividing" $n$ by $x$. If the last non-zero remainder occurs at step $k$, then if this remainder is 1, $x$ has an inverse and it is $s(k+2)$. If the remainder is not 1, then $x$ does not have an inverse.

For example, find the inverse of 15 mod 26.

Step 0: $26 = 1(15) + 11$ $s(0) = 0$

Step 1: $15 = 1(11) + 4$ $s(1) = 1$

Step 2: $11 = 2(4) + 3$ $s(2) = 0 - 1(1) \mod 26 = 25$

Step 3: $4 = 1(3) + 1$ $s(3) = 1 - 25(1) \mod 26 = -24 \mod 26 = 2$

Step 4: $3 = 3(1) + 0$ $s(4) = 25 - 2(2) \mod 26 = 21$

$s(5) = 2 - 21(1) \mod 26 = -19 \mod 26 = 7$

Notice that $15(7) = 105 = 1 + 4(26) \equiv 1 \mod 26$.

Using the half extended Euclidean algorithm we compute $1/a \mod b$.

inverse: (INT, INT) -> INT

Type: Void

inverse(a, b) ==

borg: INT := b
cl: INT := 1
d1: INT := 0

while b ~ 0 repeat
  q := a quo b
  r := a-q*b
  print [a, "=", q, "* (", b, "+", r]
  (a, b) := (b, r)
  (c1, d1) := (d1, c1-q*d1)

a "= 1 => error("moduli are not relatively prime")
positiveRemainder(c1, borg)

Type: Void

inverse(15, 26)

[15,"=",0,"* (",26,")+",15]
[26,"=",1,"* (",15,")+",11]
[15,"=",1,"* (",11,")+",4]
[4,"=",1,"* (",3,")+",1]
[3,"=",3,"* (",1,")+",0]
The Chinese Remainder Theorem says that given \( n \) moduli \( m_i \) for \( 1 \leq i \leq n \) of pairwise coprime integers and a set of congruential equations \( x \equiv c_i \pmod{m_i} \) (\( 1 \leq i \leq n \)) for an arbitrary integer \( x \), there is a unique solution \( c \) for \( x \) modulo the product \( M = \prod_{i=1}^{n} m_i \), say \( x \equiv c \pmod{M} \).

If the upper bound \( B \) of the absolute value of \( x \) is known, that is, if \( |x| \leq B \) then the number of moduli can be chosen such that \( B < M/2 \). If \( c \) is the absolutely least residue modulo \( M \) then \( c \) and \( x \) coincide so \( x \) is uniquely determined.

Let \( m_1, m_2, \ldots, m_r \) be positive integers that are pairwise relatively prime. Let \( x_1, x_2, \ldots, x_r \) be integers with \( 0 \leq x_i < m_i \). Then, there is exactly one \( x \) in the interval \( 0 \leq x < m_1 \cdots m_2 \cdots m_r \) that satisfies the remainder equations

\[
\text{irem}(x, m_i) = x_i, \quad i=1,2,\ldots,r
\]

where \( \text{irem} \) is the positive integer remainder function.

For example, let \( x_1 = 4 \), \( m_1 = 5 \), \( x_2 = 2 \), \( m_2 = 3 \). We know that

\[
\text{irem}(x, m_1) = x_1 \\
\text{irem}(x, m_2) = x_2
\]

where \( 0 \leq x_1 < m_1 \) and \( 0 \leq x_2 < m_2 \).

By the extended Euclidean Algorithm there are integers \( c \) and \( d \) such that

\[
c m_1 + d m_2 = 1
\]

In this case we are looking for an integer such that

\[
\text{irem}(x, 5) = 4, \\
\text{irem}(x, 3) = 2
\]

The algorithm we use is to first compute the positive integer remainder of \( x_1 \) and \( m_1 \) to get a new \( x_1 \):

\[
x_1 = \text{positiveRemainder}(x_1, m_1) \\
4 = \text{positiveRemainder}(4, 5)
\]

Next compute the positive integer remainder of \( x_2 \) and \( m_2 \) to get a new \( x_2 \):

\[
x_2 = \text{positiveRemainder}(x_2, m_2)
\]
2 = positiveRemainder(2,3)

Then we compute x1 + m1 ... positiveRemainder(((x2-x1)*inverse(m1,m2)),m2)
or
4+5*positiveRemainder(((2-4)*inverse(5,3)),3)
or
4+5*positiveRemainder(-2*2),3)
or
4+5*2
or
14

This function has a restricted signature which only allows for computing the chinese remainder of two numbers and two moduli.
x1:=4
4
Type: PositiveInteger
m1:=5
5
Type: PositiveInteger
x2:=2
2
Type: PositiveInteger
m2:=3
3
Type: PositiveInteger
result:=chineseRemainder(x1,m1,x2,m2)
14
Type: PositiveInteger

See Also:
  o )show IntegerNumberTheoryFunctions
IntegerNumberTheoryFunctions (INTHEORY)

Exports:
bernoulli  chineseRemainder  divisors  euler  eulerPhi
fibonacci   harmonic       jacobi   legendre  moebiusMu
numberOfDivisors  sumOfDivisors  sumOfKthPowerDivisors

— package INTHEORY IntegerNumberTheoryFunctions —

)abbrev package INTHEORY IntegerNumberTheoryFunctions
++ Author: Michael Monagan, Martin Brock, Robert Sutor, Timothy Daly
++ Date Created: June 1987
++ References: Knuth, The Art of Computer Programming Vol.2
++ Description:
++ This package provides various number theoretic functions on the integers.

IntegerNumberTheoryFunctions(): Exports == Implementation where
I ==> Integer
RN ==> Fraction I
SUP ==> SparseUnivariatePolynomial
NNI ==> NonNegativeInteger

Exports ==> with
bernoulli : I -> RN
  ++ \spad{bernoulli(n)} returns the nth Bernoulli number.
  ++ this is \spad{B(n,0)}, where \spad{B(n,x)} is the \spad{n}th Bernoulli
  ++ polynomial.
chineseRemainder: (I,I,I,I) -> I
  ++ \spad{chineseRemainder(x1,m1,x2,m2)} returns w, where w is such that
  ++ \spad{w = x1 mod m1} and \spad{w = x2 mod m2}. Note that \spad{m1} and
  ++ \spad{m2} must be relatively prime.
divisors : I -> List I
  ++ \spad{divisors(n)} returns a list of the divisors of n.
euler : I -> I
  ++ \spad{euler(n)} returns the \spad{n}th Euler number. This is
  ++ \spad{2^n E(n,1/2)}, where \spad{E(n,x)} is the nth Euler polynomial.
eulerPhi : I -> I
++ \spad{eulerPhi(n)} returns the number of integers between 1 and n ++ (including 1) which are relatively prime to n. This is the Euler phi ++ function \spad{\phi(n)} is also called the totient function.

fibonacci : I -> I
++ \spad{fibonacci(n)} returns the nth Fibonacci number. the Fibonacci ++ numbers \spad{F[n]} are defined by \spad{F[0] = F[1] = 1} and ++ \spad{F[n] = F[n-1] + F[n-2]}.
++ The algorithm has running time \spad{O(log(n)^3)}.

harmonic : I -> RN
++ \spad{harmonic(n)} returns the nth harmonic number. This is ++ \spad{H[n] = sum(1/k,k=1..n)}.

jacobi : (I,I) -> I
++ \spad{Jacobi(a,b)} returns the Jacobi symbol \spad{J(a/b)}.
++ When b is odd, \spad{Jacobi(a,b) = product(L(a/p) for p in factor b )}.
++ Note that by convention, 0 is returned if \spad{gcd(a,b) ^= 1}.
++ Iterative \spad{O(log(b)^2)} version coded by Michael Monagan June 1987.

legendre : (I,I) -> I
++ \spad{Legendre(a,p)} returns the Legendre symbol \spad{L(a/p)}.
++ \spad{Legendre(a,p) = (-1)**((p-1)/2) mod p} (p prime), which is 0 if \spad{gcd(a,p)} ++ is 0, 1 if \spad{Legendre(a,p)} is a quadratic residue \spad{mod(p)} and -1 otherwise.
++ Note that because the primality test is expensive, ++ if it is known that p is prime then use \spad{Jacobi(a,p)}.

moebiusMu : I -> I
++ \spad{moebiusMu(n)} returns the Moebius function \spad{mu(n)}.
++ \spad{mu(n)} is either -1,0 or 1 as follows:
++ \spad{mu(n) = 0} if n is divisible by a square > 1,
++ \spad{mu(n) = (-1)^k} if n is square-free and has k distinct ++ prime divisors.

numberOfDivisors: I -> I
++ \spad{numberOfDivisors(n)} returns the number of integers between 1 and n ++ (inclusive) which divide n. The number of divisors of n is often ++ denoted by \spad{tau(n)}.

sumOfDivisors : I -> I
++ \spad{sumOfDivisors(n)} returns the sum of the integers between 1 and n ++ (inclusive) which divide n. The sum of the divisors of n is often ++ denoted by \spad{sigma(n)}.

sumOfKthPowerDivisors: (I,NNI) -> I
++ \spad{sumOfKthPowerDivisors(n,k)} returns the sum of the \spad{k}th ++ powers of the integers between 1 and n (inclusive) which divide n.
++ The sum of the \spad{k}th powers of the divisors of n is often denoted ++ by \spad{sigma_k(n)}.

Implementation ==> add
import IntegerPrimesPackage(I)

-- we store the euler and bernoulli numbers computed so far in
-- a Vector because they are computed from an n-term recurrence
E: IndexedFlexibleArray(I,0) := new(1, 1)
B: IndexedFlexibleArray(RN,0) := new(1, 1)
H: Record(Hn:I, Hv:RN) := [1, 1]

harmonic n ==
s:I; h:RN
n < 0 => error("harmonic not defined for negative integers")
if n >= H.Hn then (s,h) := H else (s := 0; h := 0)
for k in s+1..n repeat h := h + 1/k
H.Hn := n
H.Hv := h
h

fibonacci n ==
n = 0 => 0
n < 0 => (odd? n => 1; -1) * fibonacci(-n)
f1, f2 : I
(f1,f2) := (0,1)
for k in length(n)-2 .. 0 by -1 repeat
  t := f2**2
  (f1,f2) := (t+f1**2, t+2*f1*f2)
  if bit?(n,k) then (f1,f2) := (f2,f1+f2)
f2

euler n ==
n < 0 => error "euler not defined for negative integers"
odd? n => 0
l := (#E) :: I
n < l => E(n)
concat_!(E, new((n+1-l)::NNI, 0)$IndexedFlexibleArray(I,0))
for i in 1 .. l by 2 repeat E(i) := 0
-- compute E(i) i = l+2,l+4,...,n given E(j) j = 0,2,...,i-2
  t,e : I
  for j in 2 .. i-2 by 2 repeat
    t := (t*(i-j+1)*(i-j+2)) quo (j*(j-1))
    e := e + t*E(j)
  E(i) := -e
E(n)

bernoulli n ==
n < 0 => error "bernoulli not defined for negative integers"
odd? n =>
n = 1 => -1/2
0
l := (#B) :: I
n < l => B(n)
concat_!(B, new((n+1-l)::NNI, 0)$IndexedFlexibleArray(RN,0))
-- compute B(i) i = l+2,l+4,...,n given B(j) j = 0,2,...,i-2
for i in l+1 .. n by 2 repeat
  t:I := 1
\begin{verbatim}
package intheory integernumbertheoryfunctions

b := (1-i)/2
for j in 2 .. i-2 by 2 repeat
    t := (t*(i-j+2)*(i-j+3)) quo (j*(j-1))
    b := b + (t::RN) * B(j)
B(i) := -b/((i+1)::RN)
B(n)

inverse : (I,I) -> I
inverse(a,b) ==
borg:I:=b
c1:I := 1
d1:I := 0
while b ^= 0 repeat
    q:I := a quo b
    r:I := a-q*b
    (a,b):=(b,r)
    (c1,d1):=(d1,c1-q*d1)
a ^= 1 => error("moduli are not relatively prime")
positiveRemainder(c1,borg)

chineseRemainder(x1,m1,x2,m2) ==
m1 < 0 or m2 < 0 => error "moduli must be positive"
x1 := positiveRemainder(x1,m1)
x2 := positiveRemainder(x2,m2)
x1 + m1 * positiveRemainder(((x2-x1) * inverse(m1,m2)),m2)

jacobi(a,b) ==
    -- Revised by Clifton Williamson January 1989.
    -- Previous version returned incorrect answers when b was even.
    -- The formula J(a/b) = product ( L(a/p) for p in factor b) is only
    -- valid when b is odd (the Legendre symbol L(a/p) is not defined
    -- for p = 2). When b is even, the Jacobi symbol J(a/b) is only
    -- defined for a = 0 or 1 (mod 4). When a = 1 (mod 8),
    -- J(a/2) = +1 and when a = 5 (mod 8), we define J(a/2) = -1.
    -- Extending by multiplicativity, we have J(a/b) for even b and
    -- appropriate a.
    -- We also define J(a/1) = 1.
    -- The point of this is the following: if d is the discriminant of
    -- a quadratic field K and chi is the quadratic character for K,
    -- then J(d/n) = chi(n) for n > 0.
    -- Reference: Hecke, Vorlesungen ueber die Theorie der Algebraischen
    -- Zahlen.
    if b < 0 then b := -b
    b = 0 => error "second argument of jacobi may not be 0"
b = 1 => 1
even? b and positiveRemainder(a,4) > 1 =>
    error "J(a/b) not defined for b even and a = 2 or 3 (mod 4)"
even? b and even? a => 0
for k in 0.. while even? b repeat b := b quo 2
\end{verbatim}
j := (odd? k and positiveRemainder(a,8) = 5 => -1; 1)
b = 1 => j
a := positiveRemainder(a,b)
-- assertion: 0 < a < b and odd? b
while a > 1 repeat
  if odd? a then
    -- J(a/b) = J(b/a) (-1)**(a-1)/(b-1)/2
    if a rem 4 = 3 and b rem 4 = 3 then j := -j
    (a,b) := (b rem a, a)
  else
    -- J(2*a/b) = J(a/b) (-1)*(b**2-1)/8
    for k in 0.. until odd? a repeat a := a quo 2
    if odd? k and (b+2) rem 8 > 4 then j := -j
  a = 0 => 0
j

legendre(a,p) ==
  prime? p => jacobi(a,p)
  error "characteristic of legendre must be prime"

eulerPhi n ==
  n = 0 => 0
  r : RN := 1
  for entry in factors factor n repeat
    r := ((entry.factor - 1)/$RN entry.factor) * r
  numer(n * r)

divisors n ==
  oldList : List Integer := [1]
  for f in factors factor n repeat
    newList : List Integer := oldList
    for k in 1..f.exponent repeat
      pow := f.factor ** k
      for m in oldList repeat
        newList := concat(pow * m, newList)
      oldList := newList
  sort((i1:Integer,i2:Integer):Boolean +-> i1 < i2, oldList)

numberOfDivisors n ==
  n = 0 => 0
  */[1+entry.exponent for entry in factors factor n]

sumOfDivisors n ==
  n = 0 => 0
  r : RN := */[(entry.factor**(entry.exponent::NNI + 1)-1)/(entry.factor-1) for entry in factors factor n]
  numer r

sumOfKthPowerDivisors(n,k) ==
  n = 0 => 0
package PRIMES IntegerPrimesPackage

We've expanded the list of small primes to include those between 1 and 10000.
The IntegerPrimesPackage implements a modification of Rabin's probabilistic primality test and the utility functions nextPrime, prevPrime, and primes.

See Also:
- )show IntegerPrimesPackage
++ Date Created: August 1987
++ Date Last Updated: 31 May 1993
++ References:
++ J. H. Davenport 'Primality testing revisited' Technical Report TR2/93
++ (ATR/6)(NP2556) Numerical Algorithms Group, Inc., Downer's Grove, IL, USA
++ and Oxford, UK, August 1993
++ Description:
++ The \spadtype{IntegerPrimesPackage} implements a modification of
++ Rabin's probabilistic
++ primality test and the utility functions \spadfun{nextPrime},
++ \spadfun{prevPrime} and \spadfun{primes}.

IntegerPrimesPackage(I:IntegerNumberSystem): with
    prime?: I -> Boolean
    ++ \spad{prime?(n)} returns true if n is prime and false if not.
    ++ The algorithm used is Rabin's probabilistic primality test
    ++ If \spad{prime? n} returns false, n is proven composite.
    ++ If \spad{prime? n} returns true, prime? may be in error
    ++ however, the probability of error is very low.
    ++ and is zero below 25*10**9 (due to a result of Pomerance et al),
    ++ below 10**12 and 10**13 due to results of Pinch,
    ++ and below 341550071728321 due to a result of Jaeschke.
    ++ Specifically, this implementation does at least 10 pseudo prime
    ++ tests and so the probability of error is \spad{< 4**(-10)}. 
    ++ The running time of this method is cubic in the length
    ++ of the input n, that is \spad{O( (log n)**3 )}, for n<10**20.
    ++ beyond that, the algorithm is quartic, \spad{O( (log n)**4 )}.
    ++ Two improvements due to Davenport have been incorporated
    ++ which catches some trivial strong pseudo-primes, such as
    ++ [Jaeschke, 1991] 1377161253229053 * 413148375987157, which
    ++ the original algorithm regards as prime

nextPrime: I -> I
    ++ \spad{nextPrime(n)} returns the smallest prime strictly larger than n
prevPrime: I -> I
    ++ \spad{prevPrime(n)} returns the largest prime strictly smaller than n
primes: (I,I) -> List I
    ++ \spad{primes(a,b)} returns a list of all primes p with
    ++ \spad{a <= p <= b}

== add

smallPrimes

This is a table of all of the primes in [2..10000]. It is used by the prime? function to check
for primality. It is used by the primes function to generate arrays of primes in a given range.
Changing the range included in this table implies changing the value of the nextSmallPrime
variable. There is a constant in the function squareFree from IntegerFactorizationPackage
that is the square of the upper bound of the table range, in this case 1000000.

— package PRIMES IntegerPrimesPackage —

smallPrimes: List I := [2::I, 3::I, 5::I, 7::I, 11::I, 13::I, 17::I, 19::I, 23::I, 29::I, 31::I, 37::I, 41::I, 41::I, 47::I, 53::I, 59::I, 61::I, 67::I, 71::I, 73::I, 79::I, 83::I, 89::I, 97::I, 101::I, 103::I, 107::I, 109::I, 113::I, 127::I, 131::I, 137::I, 139::I, 149::I, 151::I, 157::I, 163::I, 167::I, 173::I, 179::I, 181::I, 191::I, 193::I, 197::I, 199::I, 211::I, 223::I, 227::I, 229::I, 233::I, 239::I, 241::I, 251::I, 257::I, 263::I, 269::I, 271::I, 277::I, 281::I, 283::I, 293::I, 307::I, 311::I, 313::I, 317::I, 331::I, 337::I, 347::I, 349::I, 353::I, 359::I, 367::I, 373::I, 379::I, 383::I, 389::I, 397::I, 401::I, 409::I, 419::I, 421::I, 431::I, 433::I, 439::I, 443::I, 449::I, 457::I, 461::I, 463::I, 467::I, 479::I, 487::I, 491::I, 499::I, 503::I, 509::I, 521::I, 523::I, 541::I, 547::I, 557::I, 563::I, 569::I, 571::I, 577::I, 587::I, 593::I, 599::I, 601::I, 607::I, 613::I, 617::I, 619::I, 631::I, 641::I, 643::I, 647::I, 653::I, 659::I, 661::I, 673::I, 677::I, 683::I, 691::I, 701::I, 709::I, 719::I, 727::I, 733::I, 739::I, 743::I, 751::I, 757::I, 761::I, 769::I, 773::I, 777::I, 797::I, 809::I, 811::I, 821::I, 823::I, 827::I, 829::I, 839::I, 853::I, 857::I, 859::I, 863::I, 877::I, 881::I, 883::I, 887::I, 907::I, 911::I, 919::I, 929::I, 937::I, 941::I, 947::I, 953::I, 967::I, 971::I, 977::I, 983::I, 991::I, 997::I, 1009::I, 1013::I, 1019::I, 1021::I, 1031::I, 1033::I, 1039::I, 1049::I, 1051::I, 1061::I, 1063::I, 1069::I, 1087::I, 1091::I, 1093::I, 1097::I, 1103::I, 1109::I, 1117::I, 1123::I, 1129::I, 1151::I, 1153::I, 1157::I, 1163::I, 1171::I, 1181::I, 1187::I, 1193::I, 1201::I, 1207::I, 1213::I, 1217::I, 1223::I, 1229::I, 1231::I, 1237::I, 1249::I, 1259::I, 1277::I, 1279::I, 1283::I, 1289::I, 1291::I, 1297::I, 1301::I, 1303::I, 1307::I, 1319::I, 1321::I, 1327::I, 1361::I, 1367::I, 1373::I, 1381::I, 1399::I, 1409::I, 1423::I, 1427::I, 1429::I, 1433::I, 1439::I, 1447::I, 1451::I, 1453::I, 1459::I, 1471::I, 1481::I, 1483::I, 1487::I, 1489::I, 1493::I, 1499::I, 1511::I, 1523::I, 1531::I, 1543::I, 1549::I, 1553::I, 1559::I, 1567::I, 1571::I, 1579::I, 1583::I, 1597::I, 1601::I, 1607::I, 1609::I, 1613::I, 1619::I, 1621::I, 1627::I, 1637::I, 1657::I, 1663::I, 1667::I, 1669::I, 1693::I, 1697::I, 1699::I, 1709::I, 1721::I, 1723::I, 1733::I, 1741::I, 1747::I, 1753::I, 1759::I, 1777::I, 1783::I, 1787::I, 1789::I, 1801::I, 1811::I, 1823::I, 1831::I, 1847::I, 1861::I, 1867::I, 1871::I, 1873::I, 1877::I, 1879::I, 1889::I, 1901::I, 1907::I, 1913::I, 1931::I, 1933::I, 1949::I, 1951::I, 1973::I, 1979::I, 1987::I, 1993::I, 1997::I,
1999::I, 2003::I, 2011::I, 2017::I, 2027::I, 2029::I, _
2039::I, 2053::I, 2063::I, 2069::I, 2081::I, 2083::I, _
2087::I, 2089::I, 2099::I, 2111::I, 2113::I, 2129::I, _
2131::I, 2137::I, 2141::I, 2143::I, 2153::I, 2161::I, _
2179::I, 2203::I, 2207::I, 2213::I, 2221::I, 2237::I, _
2239::I, 2243::I, 2261::I, 2267::I, 2269::I, 2273::I, _
2281::I, 2287::I, 2293::I, 2297::I, 2309::I, 2311::I, _
2333::I, 2339::I, 2341::I, 2347::I, 2351::I, 2357::I, _
2371::I, 2377::I, 2381::I, 2383::I, 2389::I, 2393::I, _
2399::I, 2411::I, 2417::I, 2423::I, 2437::I, 2441::I, _
2447::I, 2459::I, 2467::I, 2473::I, 2477::I, 2503::I, _
2521::I, 2531::I, 2539::I, 2543::I, 2549::I, 2551::I, _
2557::I, 2579::I, 2591::I, 2593::I, 2609::I, 2617::I, _
2621::I, 2633::I, 2647::I, 2657::I, 2659::I, 2663::I, _
2671::I, 2677::I, 2683::I, 2687::I, 2689::I, 2693::I, _
2699::I, 2707::I, 2711::I, 2713::I, 2719::I, 2729::I, _
2731::I, 2741::I, 2749::I, 2753::I, 2767::I, 2777::I, _
2789::I, 2791::I, 2797::I, 2801::I, 2803::I, 2819::I, _
2833::I, 2837::I, 2843::I, 2851::I, 2857::I, 2861::I, _
2879::I, 2887::I, 2897::I, 2903::I, 2909::I, 2917::I, _
2927::I, 2939::I, 2953::I, 2957::I, 2963::I, 2969::I, _
2971::I, 2999::I, 3001::I, 3011::I, 3019::I, 3023::I, _
3037::I, 3041::I, 3049::I, 3061::I, 3067::I, 3079::I, _
3083::I, 3089::I, 3109::I, 3119::I, 3121::I, 3137::I, _
3163::I, 3167::I, 3169::I, 3181::I, 3187::I, 3191::I, _
3203::I, 3209::I, 3217::I, 3221::I, 3229::I, 3251::I, _
3253::I, 3257::I, 3259::I, 3271::I, 3299::I, 3301::I, _
3307::I, 3313::I, 3319::I, 3323::I, 3329::I, 3331::I, _
3343::I, 3347::I, 3359::I, 3361::I, 3371::I, 3373::I, _
3389::I, 3391::I, 3407::I, 3413::I, 3433::I, 3449::I, _
3457::I, 3461::I, 3463::I, 3467::I, 3469::I, 3491::I, _
3499::I, 3511::I, 3517::I, 3527::I, 3529::I, 3533::I, _
3539::I, 3541::I, 3547::I, 3557::I, 3559::I, 3571::I, _
3581::I, 3583::I, 3593::I, 3607::I, 3613::I, 3617::I, _
3623::I, 3631::I, 3637::I, 3643::I, 3659::I, 3671::I, _
3673::I, 3677::I, 3691::I, 3697::I, 3701::I, 3709::I, _
3719::I, 3727::I, 3733::I, 3739::I, 3761::I, 3767::I, _
3769::I, 3779::I, 3793::I, 3797::I, 3803::I, 3821::I, _
3823::I, 3833::I, 3847::I, 3851::I, 3853::I, 3863::I, _
3877::I, 3881::I, 3889::I, 3907::I, 3911::I, 3917::I, _
3919::I, 3923::I, 3929::I, 3931::I, 3943::I, 3947::I, _
3967::I, 3989::I, 4001::I, 4003::I, 4007::I, 4013::I, _
4019::I, 4021::I, 4027::I, 4049::I, 4051::I, 4057::I, _
4073::I, 4079::I, 4091::I, 4093::I, 4099::I, 4111::I, _
4127::I, 4129::I, 4133::I, 4139::I, 4153::I, 4157::I, _
4159::I, 4177::I, 4201::I, 4211::I, 4217::I, 4219::I, _
4229::I, 4231::I, 4241::I, 4243::I, 4253::I, 4259::I, _
4261::I, 4271::I, 4273::I, 4283::I, 4289::I, 4297::I, _
4327::I, 4337::I, 4339::I, 4349::I, 4357::I, 4363::I, _
4373::I, 4391::I, 4397::I, 4409::I, 4421::I, 4423::I, _
7019::I, 7027::I, 7039::I, 7043::I, 7057::I, 7069::I, _,
7079::I, 7103::I, 7109::I, 7121::I, 7127::I, 7129::I,_,
7151::I, 7159::I, 7177::I, 7187::I, 7193::I, 7207::I,_,
7211::I, 7213::I, 7219::I, 7229::I, 7237::I, 7243::I,_,
7247::I, 7253::I, 7283::I, 7297::I, 7307::I, 7309::I,_,
7321::I, 7331::I, 7333::I, 7349::I, 7351::I, 7369::I,_,
7393::I, 7411::I, 7417::I, 7433::I, 7451::I, 7457::I,_,
7459::I, 7477::I, 7481::I, 7487::I, 7489::I, 7499::I,_,
7507::I, 7517::I, 7523::I, 7529::I, 7537::I, 7541::I,_,
7547::I, 7549::I, 7559::I, 7561::I, 7573::I, 7577::I,_,
7583::I, 7589::I, 7591::I, 7603::I, 7607::I, 7621::I,_,
7639::I, 7643::I, 7649::I, 7669::I, 7673::I, 7681::I,_,
7687::I, 7691::I, 7699::I, 7703::I, 7717::I, 7723::I,_,
7727::I, 7741::I, 7753::I, 7757::I, 7759::I, 7789::I,_,
7793::I, 7817::I, 7823::I, 7829::I, 7841::I, 7853::I,_,
7867::I, 7873::I, 7877::I, 7879::I, 7883::I, 7901::I,_,
7907::I, 7919::I, 7927::I, 7933::I, 7937::I, 7949::I,_,
7951::I, 7963::I, 7993::I, 8009::I, 8011::I, 8017::I,_,
8039::I, 8053::I, 8059::I, 8069::I, 8081::I, 8087::I,_,
8089::I, 8093::I, 8101::I, 8111::I, 8117::I, 8123::I,_,
8147::I, 8161::I, 8167::I, 8171::I, 8179::I, 8191::I,_,
8209::I, 8219::I, 8221::I, 8231::I, 8233::I, 8237::I,_,
8243::I, 8263::I, 8269::I, 8273::I, 8287::I, 8291::I,_,
8293::I, 8297::I, 8311::I, 8317::I, 8329::I, 8353::I,_,
8363::I, 8369::I, 8377::I, 8387::I, 8389::I, 8419::I,_,
8423::I, 8429::I, 8431::I, 8443::I, 8447::I, 8461::I,_,
8467::I, 8501::I, 8513::I, 8521::I, 8527::I, 8537::I,_,
8539::I, 8543::I, 8563::I, 8573::I, 8581::I, 8597::I,_,
8599::I, 8609::I, 8623::I, 8627::I, 8629::I, 8641::I,_,
8647::I, 8663::I, 8669::I, 8677::I, 8681::I, 8689::I,_,
8693::I, 8699::I, 8707::I, 8713::I, 8719::I, 8731::I,_,
8737::I, 8741::I, 8747::I, 8753::I, 8761::I, 8779::I,_,
8783::I, 8803::I, 8807::I, 8819::I, 8821::I, 8831::I,_,
8837::I, 8839::I, 8849::I, 8861::I, 8863::I, 8867::I,_,
8887::I, 8893::I, 8923::I, 8929::I, 8933::I, 8941::I,_,
8951::I, 8963::I, 8969::I, 8971::I, 8999::I, 9001::I,_,
9007::I, 9011::I, 9013::I, 9029::I, 9041::I, 9043::I,_,
9049::I, 9059::I, 9067::I, 9091::I, 9103::I, 9109::I,_,
9127::I, 9133::I, 9137::I, 9151::I, 9157::I, 9161::I,_,
9173::I, 9181::I, 9187::I, 9199::I, 9203::I, 9209::I,_,
9221::I, 9227::I, 9239::I, 9241::I, 9257::I, 9277::I,_,
9281::I, 9283::I, 9293::I, 9311::I, 9319::I, 9323::I,_,
9337::I, 9341::I, 9343::I, 9349::I, 9371::I, 9377::I,_,
9391::I, 9397::I, 9403::I, 9413::I, 9419::I, 9421::I,_,
9431::I, 9433::I, 9437::I, 9439::I, 9461::I, 9463::I,_,
9467::I, 9473::I, 9479::I, 9491::I, 9497::I, 9511::I,_,
9521::I, 9533::I, 9539::I, 9547::I, 9551::I, 9587::I,_,
9601::I, 9613::I, 9619::I, 9623::I, 9629::I, 9631::I,_,
9643::I, 9649::I, 9661::I, 9677::I, 9679::I, 9689::I,_,
9697::I, 9719::I, 9721::I, 9733::I, 9739::I, 9743::I,
primes

--- package PRIMES IntegerPrimesPackage ---

primes(m, n) ==
  -- computes primes from m to n inclusive using prime?
  l:List(I) :=
    m <= two => [two]
    empty()
    n < two or n < m => empty()
    if even? m then m := m + 1
  ll:List(I) := [k::I for k in
                   convert(m)@Integer..convert(n)@Integer by 2 | prime?(k::I)]
  reverse_! concat_!(ll, l)

rabinProvesComposite : (I,I,I,I,NonNegativeInteger) -> Boolean
rabinProvesCompositeSmall : (I,I,I,I,NonNegativeInteger) -> Boolean

----

rabinProvesCompositeSmall

— package PRIMES IntegerPrimesPackage —

rabinProvesCompositeSmall(p,n,nm1,q,k) ==
    -- probability n prime is > 3/4 for each iteration
    -- for most n this probability is much greater than 3/4
    t := powmod(p, q, n)
    -- neither of these cases tells us anything
    if not ((t = 1) or t = nm1) then
        for j in 1..k-1 repeat
            oldt := t
            t := mulmod(t, t, n)
            (t = 1) => return true
            -- we have squared something not -1 and got 1
            t = nm1 =>
                leave
            not (t = nm1) => return true
        false

----

rabinProvesComposite

— package PRIMES IntegerPrimesPackage —

rabinProvesComposite(p,n,nm1,q,k) ==
    -- probability n prime is > 3/4 for each iteration
    -- for most n this probability is much greater than 3/4
    t := powmod(p, q, n)
    -- neither of these cases tells us anything
    if t=nm1 then count2Order(1):=count2Order(1)+1
    if not ((t = 1) or t = nm1) then
        for j in 1..k-1 repeat
            oldt := t
            t := mulmod(t, t, n)
            (t = 1) => return true
            -- we have squared something not -1 and got 1
\[ t = n_1 \Rightarrow \]
\[ \text{rootsMinus1} := \text{union(rootsMinus1, oldt)} \]
\[ \text{count2Order}(j+1) := \text{count2Order}(j+1) + 1 \]
\[ \text{leave} \]
\[ \neg (t = n_1) \Rightarrow \text{return true} \]
\# rootsMinus1 > 2 \Rightarrow \text{true} -- \mathbb{Z}/n\mathbb{Z} can’t be a field
\text{false}

---

\text{prime?}

--- package PRIMES IntegerPrimesPackage ---

\text{prime? } n ==
\begin{align*}
& n < \text{two} \Rightarrow \text{false} \\
& n < \text{nextSmallPrime} \Rightarrow \text{member?}(n, \text{smallPrimes}) \\
& \neg (\gcd(n, \text{productSmallPrimes}) = 1) \Rightarrow \text{false} \\
& n < \text{nextSmallPrimeSquared} \Rightarrow \text{true} \\
\end{align*}
\begin{align*}
n_1 & := n-1 \\
q & := (n_1) \text{ quo two} \\
& \text{for } k \text{ in 1.. while not odd? } q \text{ repeat } q := q \text{ quo two} \\
& \quad -- q = (n-1) \text{ quo } 2^*k \text{ for largest possible } k \\
\end{align*}
\begin{align*}
n & < \text{JaeschkeLimit} \Rightarrow \\
& \text{rabinProvesCompositeSmall}(2::I,n,n_1,q,k) \Rightarrow \text{return false} \\
& \text{rabinProvesCompositeSmall}(3::I,n,n_1,q,k) \Rightarrow \text{return false} \\
n & < \text{PomeranceLimit} \Rightarrow \\
& \text{rabinProvesCompositeSmall}(5::I,n,n_1,q,k) \Rightarrow \text{return false} \\
& \quad \text{member?}(n, \text{PomeranceList}) \Rightarrow \text{return false} \\
& \quad \text{true} \\
& \text{rabinProvesCompositeSmall}(7::I,n,n_1,q,k) \Rightarrow \text{return false} \\
n & < \text{PinchLimit} \Rightarrow \\
& \text{rabinProvesCompositeSmall}(10::I,n,n_1,q,k) \Rightarrow \text{return false} \\
& \quad \text{member?}(n, \text{PinchList}) \Rightarrow \text{return false} \\
& \quad \text{true} \\
& \text{rabinProvesCompositeSmall}(5::I,n,n_1,q,k) \Rightarrow \text{return false} \\
& \text{rabinProvesCompositeSmall}(11::I,n,n_1,q,k) \Rightarrow \text{return false} \\
n & < \text{PinchLimit2} \Rightarrow \\
& \text{member?}(n, \text{PinchList2}) \Rightarrow \text{return false} \\
& \quad \text{true} \\
& \text{rabinProvesCompositeSmall}(13::I,n,n_1,q,k) \Rightarrow \text{return false} \]
rabinProvesCompositeSmall(17::I,n,nm1,q,k) => return false
true

rootsMinus1 := empty()
count2Order := new(k,0) -- vector of k zeroes

mn := minIndex smallPrimes
for i in mn+1..mn+10 repeat
  rabinProvesComposite(smallPrimes i,n,nm1,q,k) => return false
import IntegerRoots(I)
q > 1 and perfectSquare?(3*n+1) => false
((n9:=n rem (9::I))=1 or n9 = -1) and perfectSquare?(8*n+1) => false
-- Both previous tests from Damgard & Landrock
currPrime := smallPrimes(mn+10)
probablySafe := tenPowerTwenty
while count2Order(k) = 0 or n > probablySafe repeat
  currPrime := nextPrime currPrime
  probablySafe := probablySafe*(100::I)
rabinProvesComposite(currPrime,n,nm1,q,k) => return false
true

nextPrime

— package PRIMES IntegerPrimesPackage —

nextPrime n ==
  -- computes the first prime after n
  n < two => two
  if odd? n then n := n + two else n := n + 1
  while not prime? n repeat n := n + two
  n

prevPrime

— package PRIMES IntegerPrimesPackage —

prevPrime n ==
  -- computes the first prime before n
  n < 3::I => error "no primes less than 2"
n = 3 :: I => two
if odd? n then n := n - two else n := n - 1
while not prime? n repeat n := n - two
n

---

--- PRIMES.dotabb ---

"PRIMES" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRIMES"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"PRIMES" -> "FSAGG"

package INTRET IntegerRetractions

--- IntegerRetractions.input ---

)set break resume
)sys rm -f IntegerRetractions.output
)spool IntegerRetractions.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show IntegerRetractions
--R
--R IntegerRetractions(S: RetractableTo(Integer)) is a package constructor
--R Abbreviation for IntegerRetractions is INTRET
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTRET
--R
--R----------------------------------- Operations ----------------------------------
--R integer : S -> Integer        integer? : S -> Boolean
--R integerIfCan : S -> Union(Integer,"failed")
--R
--R
)spool
)lisp (bye)
--- IntegerRetractions.help ---

====================================================================
IntegerRetractions examples
====================================================================

Provides integer testing and retraction functions.

See Also:
o )show IntegerRetractions

---

IntegerRetractions (INTRET)

Exports:
  integer  integer?  integerIfCan

--- package INTRET IntegerRetractions ---

)abbrev package INTRET IntegerRetractions
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Provides integer testing and retraction functions.

IntegerRetractions(S:RetractableTo(Integer)): with
  integer : S -> Integer
    ++ integer(x) returns x as an integer;
    ++ error if x is not an integer;
  integer? : S -> Boolean
++ integer?(x) is true if x is an integer, false otherwise;
integerIfCan: S -> Union(Integer, "failed")
++ integerIfCan(x) returns x as an integer,
++ "failed" if x is not an integer;
== add
integer s     == retract s
integer? s     == retractIfCan(s) case Integer
integerIfCan s == retractIfCan s

package IROOT IntegerRoots

-- IntegerRoots.input --

)set break resume
)sys rm -f IntegerRoots.output
)spool IntegerRoots.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerRoots
--R
--R IntegerRoots(I: IntegerNumberSystem) is a package constructor
--R Abbreviation for IntegerRoots is IROOT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IROOT
--R
--R----------------------------- Operations -----------------------------
--R perfectSquare? : I -> Boolean
--R approxNthRoot : (I,NonNegativeInteger) -> I
--R perfectNthPower? : (I,NonNegativeInteger) -> Boolean
--R perfectNthRoot : (I,NonNegativeInteger) -> Union(I,"failed")
---R perfectNthRoot : I -> Record(base: I, exponent: NonNegativeInteger)
---R
---E 1

)spool
)lisp (bye)

---

<table>
<thead>
<tr>
<th>IntegerRoots.help</th>
</tr>
</thead>
</table>

====================================================================
IntegerRoots examples
====================================================================

The IntegerRoots package computes square roots and nth roots of integers efficiently.

See Also:
  o )show IntegerRoots

---

IntegerRoots (IROOT)

Exports:
  approxSqrt  approxNthRoot  perfectNthPower?  perfectNthRoot  perfectSqrt  perfectSquare?

--- package IROOT IntegerRoots ---

)abbrev package IROOT IntegerRoots
++ Author: Michael Monagan
++ Date Created: November 1987
++ Description:
++ The \spadtype{IntegerRoots} package computes square roots and
++ nth roots of integers efficiently.

IntegerRoots(I:IntegerNumberSystem): Exports == Implementation where
NNI ==> NonNegativeInteger

Exports ==> with
perfectNthPower?: (I, NNI) -> Boolean
++ \spad{perfectNthPower?(n,r)} returns true if n is an \spad{r}th
++ power and false otherwise
perfectNthRoot: (I,NNI) -> Union(I, "failed")
++ \spad{perfectNthRoot(n,r)} returns the \spad{r}th root of n if n
++ is an \spad{r}th power and returns "failed" otherwise
perfectNthRoot: I -> Record(base:I, exponent:NNI)
++ \spad{perfectNthRoot(n)} returns \spad{[x,r]}, where \spad{n = x^r}
++ and r is the largest integer such that n is a perfect \spad{r}th power
approxNthRoot: (I,NNI) -> I
++ \spad{approxNthRoot(n,r)} returns an approximation x
++ to \spad{n**(1/r)} such that \spad{-1 < x - n**(1/r) < 1}
perfectSquare?: I -> Boolean
++ \spad{perfectSquare?(n)} returns true if n is a perfect square
++ and false otherwise
perfectSqrt: I -> Union(I, "failed")
++ \spad{perfectSqrt(n)} returns the square root of n if n is a
++ perfect square and returns "failed" otherwise
approxSqrt: I -> I
++ \spad{approxSqrt(n)} returns an approximation x
++ to \spad{sqrt(n)} such that \spad{-1 < x - sqrt(n) < 1}.
++ Compute an approximation s to \spad{sqrt(n)} such that
++ \spad{-1 < s - sqrt(n) < 1}
++ A variable precision Newton iteration is used.
++ The running time is \spad{(log(n)**2)}.

Implementation ==> add
import IntegerPrimesPackage(I)

resMod144: List I := [0::I,1::I,4::I,9::I,16::I,25::I,36::I,49::I,_,
52::I,64::I,73::I,81::I,97::I,100::I,112::I,121::I]
two := 2::I

perfectSquare?

— package IROOT IntegerRoots —
perfectSquare? a \quad == \quad (\text{perfectSqrt} \ a) \text{ case I}


perfectNthPower?

— package IROOT IntegerRoots —

\text{perfectNthPower?(b, n)} \quad == \quad \text{perfectNthRoot}(b, n) \text{ case I}


perfectNthRoot

— package IROOT IntegerRoots —

\text{perfectNthRoot} n \quad == \quad \text{-- complexity } (\log \log n)^{2} \quad (\log n)^{2} \\
\quad m: \text{NNI} \\
\quad (n = 1) \quad \text{or zero?} \quad n \quad \text{or} \quad n = -1 \quad => \quad [n, 1] \\
\quad e: \text{NNI} \quad := \quad 1 \\
\quad p: \text{NNI} \quad := \quad 2 \\
\quad \text{while } p::\text{I} \quad <= \quad \text{length}(n) + 1 \quad \text{repeat} \\
\quad \quad \text{for } m \quad \text{in } 0.. \quad \text{while } (r := \text{perfectNthRoot}(n, p)) \text{ case I repeat} \\
\quad \quad \quad n \quad := \quad r::\text{I} \\
\quad \quad \quad e \quad := \quad e \quad * \quad p \quad ** \quad m \\
\quad \quad \quad p \quad := \quad \text{convert(nextPrime(p::\text{I}))@Integer :: NNI} \\
\quad [n, e]


approxNthRoot

— package IROOT IntegerRoots —

\text{approxNthRoot}(a, n) \quad == \quad \text{-- complexity } (\log \log n) \quad (\log n)^{2} \\
\quad \text{zero? } n \quad => \quad \text{error "invalid arguments"} \\
\quad (n = 1) \quad => \quad a \\
\quad n=2 \quad => \quad \text{approxSqrt} \ a \\
\quad \text{negative? } a \quad => \\
\quad \quad \text{odd? } n \quad => \quad - \text{approxNthRoot}(-a, n)
0
zero? a => 0
(a = 1) => 1
-- quick check for case of large n
((3*n) quo 2):I >= (l := length a) => two
-- the initial approximation must be >= the root
y := max(two, shift(1, (n::I+1-1) quo (n::I)))
z:I := 1
n1:= (n-1)::NNI
while z > 0 repeat
  x := y
  xn:= x**n1
  y := (n1*x*xn+a) quo (n*xn)
  z := x-y
x

perfectNthRoot

— package IROOT IntegerRoots —

perfectNthRoot(b, n) ==
  (r := approxNthRoot(b, n)) ** n = b => r
  "failed"

perfectSqrt

— package IROOT IntegerRoots —

perfectSqrt a ==
a < 0 or not member?(a rem (144::I), resMod144) => "failed"
(s := approxSqrt a) * s = a => s
  "failed"
approxSqrt

--- package IROOT IntegerRoots ---

approxSqrt a ==
  a < 1 => 0
  if (n := length a) > (100::I) then
    -- variable precision newton iteration
    n := n quo (4::I)
    s := approxSqrt shift(a, -2 * n)
    s := shift(s, n)
    return ((1 + s + a quo s) quo two)
  -- initial approximation for the root is within a factor of 2
  (new, old) := (shift(1, n quo two), 1)
  while new ^= old repeat
    (new, old) := ((1 + new + a quo new) quo two, new)
  new

---

--- IROOT.dotabb ---

"IROOT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IROOT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"IROOT" -> "FLAGG"

---

package INTSLPE IntegerSolveLinearPolynomialEquation

--- IntegerSolveLinearPolynomialEquation.input ---

)set break resume
)sys rm -f IntegerSolveLinearPolynomialEquation.output
)spool IntegerSolveLinearPolynomialEquation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegerSolveLinearPolynomialEquation
--R
--R IntegerSolveLinearPolynomialEquation is a package constructor
This package provides the implementation for the solveLinearPolynomialEquation operation over the integers. It uses a lifting technique from the package GenExEuclid.

See Also:
- )show IntegerSolveLinearPolynomialEquation
--- package INTSLPE IntegerSolveLinearPolynomialEquation ---

)abbrev package INTSLPE IntegerSolveLinearPolynomialEquation
++ Author: Davenport
++ Date Created: 1991
++ Description:
++ This package provides the implementation for the
++ \texttt{solveLinearPolynomialEquation} operation over the integers. It uses a lifting technique
++ from the package \texttt{GenExEuclid}

\texttt{IntegerSolveLinearPolynomialEquation()}: C == T
where
\texttt{ZP} == \texttt{SparseUnivariatePolynomial} \\texttt{Integer}
\texttt{C} == with
\texttt{solveLinearPolynomialEquation}: (\texttt{List ZP}, \texttt{ZP}) \to \texttt{Union(List ZP,} \texttt{"failed")}
\texttt{solveLinearPolynomialEquation([f1, \ldots, fn], g)}
\texttt{where the fi are relatively prime to each other}
\texttt{returns a list of ai such that}
\texttt{g/prod fi = sum ai/fi}
\texttt{or returns "failed" if no such list of ai's exists.}

T == add
oldlp: \texttt{List ZP} := []
slpePrime: \texttt{Integer} := (2::Integer)
oldtable: \texttt{Vector List ZP} := empty()
solveLinearPolynomialEquation(lp, p) ==
if (oldlp = lp) then
  -- we have to generate a new table
  deg: = 
  [degree u for u in lp]
  ans: \texttt{Union(Vector List ZP,} \texttt{"failed")} := \texttt{"failed"}
  slpePrime: = 2147483647::Integer \quad -- 2**31 -1 : a prime
  \quad -- a good test case for this package is
  \quad -- (x**31-1,x-2),2
  while (ans case "failed") repeat
    ans: = \texttt{tablePow}(deg, slpePrime, lp) \$\texttt{GenExEuclid(Integer,ZP)}
    if (ans case "failed") then
      slpePrime: = \texttt{prevPrime}(slpePrime) \$\texttt{IntegerPrimesPackage(Integer)}
      oldtable: = (ans:: Vector List ZP)
    answer: = solveid(p, slpePrime, oldtable)
  answer

--- INTSLPE.dotabb ---

"INTSLPE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTSLPE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTSLPE" -> "PFECAT"

package IBATOOL IntegralBasisTools

-- IntegralBasisTools.input --

)set break resume
)sys rm -f IntegralBasisTools.output
)spool IntegralBasisTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegralBasisTools

--R IntegralBasisTools(R,UP: UnivariatePolynomialCategory(R),F: FramedAlgebra(R,UP))where
--R R: EuclideanDomainwith
--R squareFree : % -> Factored(%) is a package constructor
--R Abbreviation for IntegralBasisTools is IBATOOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IBATOOL
--R
--R---------------------- Operations --------------------------
--R diagonalProduct : Matrix(R) -> R
--R divideIfCan! : (Matrix(R),Matrix(R),R,Integer) -> R
--R idealiser : (Matrix(R),Matrix(R)) -> Matrix(R)
--R idealiser : (Matrix(R),Matrix(R),R) -> Matrix(R)
--R idealiserMatrix : (Matrix(R),Matrix(R)) -> Matrix(R)
--R leastPower : (NonNegativeInteger,NonNegativeInteger) -> NonNegativeInteger
--R matrixGcd : (Matrix(R),R,NonNegativeInteger) -> R
--R moduleSum : (Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R)),Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R)))
--R
--E 1

)spool
)lisp (bye)

-- IntegralBasisTools.help --

====================================================================
This package contains functions used in the packages FunctionFieldIntegralBasis and NumberFieldIntegralBasis.

See Also:
- )show IntegralBasisTools
NNI ==> NonNegativeInteger
Ans ==> Record(basis: Mat, basisDen: R, basisInv: Mat)

Exports => with

diagonalProduct: Mat -> R
++ diagonalProduct(m) returns the product of the elements on the
++ diagonal of the matrix m

matrixGcd: (Mat,R,NNI) -> R
++ matrixGcd(mat,sing,n) is \spad{gcd(sing,g)} where \spad{g} is the
++ gcd of the entries of the \spad{n}-by-\spad{n} upper-triangular
++ matrix \spad{mat}.

divideIfCan_!: (Matrix R,Matrix R,R,Integer) -> R
++ divideIfCan!(matrix,matrixOut,prime,n) attempts to divide the
++ entries of \spad{matrix} by \spad{prime} and store the result in
++ \spad{matrixOut}. If it is successful, 1 is returned and if not,
++ \spad{prime} is returned. Here both \spad{matrix} and
++ \spad{matrixOut} are \spad{n}-by-\spad{n} upper triangular matrices.

leastPower: (NNI,NNI) -> NNI
++ leastPower(p,n) returns e, where e is the smallest integer
++ such that \spad{p ** e >= n}

idealiser: (Mat,Mat) -> Mat
++ idealiser(m1,m2) computes the order of an ideal defined by m1 and m2

idealiser: (Mat,Mat,R) -> Mat
++ idealiser(m1,m2,d) computes the order of an ideal defined by m1 and m2
++ where d is the known part of the denominator

idealiserMatrix: (Mat, Mat) -> Mat
++ idealiserMatrix(m1, m2) returns the matrix representing the linear
++ conditions on the Ring associated with an ideal defined by m1 and m2.

moduleSum: (Ans,Ans) -> Ans
++ moduleSum(m1,m2) returns the sum of two modules in the framed
++ algebra \spad{F}. Each module \spad{mi} is represented as follows:
++ F is a framed algebra with R-module basis \spad{w1,w2,...,wn} and
++ \spad{mi} is a record \spad{[basis,basisDen,basisInv]}. If
++ \spad{basis} is the matrix \spad{[aij, i = 1..n, j = 1..n]}, then
++ \spad{basisDen} is the matrix \spad{[aij * vj, j = 1..n]}, i.e. the
++ \spad{aij}th row of 'basis' contains the coordinates of the
++ \spad{aij}th basis vector. Similarly, \spad{basisInv} is
++ respect to the basis \spad{w1,...,vn}: if \spad{basisInv} is the
++ matrix \spad{[bij, i = 1..n, j = 1..n]}, then
++ \spad{[bij * wij, j = 1..n]}

Implementation => add
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

diagonalProduct m ==
ans : R := 1
for i in minRowIndex m . . maxRowIndex m
for j in minColIndex m . . maxColIndex m repeat
    ans := ans * qelt(m, i, j)
ans

matrixGcd(mat,sing,n) ==
    -- note that 'matrix' is upper triangular;
    -- no need to do anything below the diagonal;
    d := sing
    for i in 1 .. n repeat
        for j in i .. n repeat
            if not zero?(mij := qelt(mat,i,j)) then d := gcd(d,mij)
        if (d = 1) => return d
    d

divideIfCan!(matrix,matrixOut,prime,n) ==
    -- note that both 'matrix' and 'matrixOut' will be upper triangular;
    -- no need to do anything below the diagonal.
    for i in 1 .. n repeat
        for j in i .. n repeat
            a := (qelt(matrix,i,j) exquo prime) case "failed" => return prime
            qsetelt_!(matrixOut,i,j,a :: R)

leastPower(p,n) ==
    -- efficiency is not an issue here
    e : NNI := 1; q := p
    while q < n repeat (e := e + 1; q := q * p)
    e

idealiserMatrix(ideal,idealinv) ==
    -- computes the Order of the ideal
    n := rank()$F
    bigm := zero(n * n,n)$Mat
    mr := minRowIndex bigm; mc := minColIndex bigm
    v := basis()$F
    for i in 0 .. n-1 repeat
        r := regularRepresentation qelt(v,i + minIndex v)
        m := ideal * r * idealinv
        for j in 0 .. n-1 repeat
            for k in 0 .. n-1 repeat
                bigm(j * n + k + mr,i + mc) := qelt(m,j + mr,k + mc)
    bigm

idealiser(ideal,idealinv) ==
    bigm := idealiserMatrix(ideal, idealinv)
    transpose squareTop rowEch bigm

idealiser(ideal,idealinv,denom) ==
bigm := (idealiserMatrix(ideal, idealinv) exquo denom)::Mat
transpose squareTop rowEchelon(bigm, denom)

moduleSum(mod1, mod2) ==
  rb1 := mod1.basis; rbden1 := mod1.basisDen; rbinv1 := mod1.basisInv
  rb2 := mod2.basis; rbden2 := mod2.basisDen; rbinv2 := mod2.basisInv
  -- compatibility check: doesn't take much computation time
  (not square? rb1) or (not square? rbinv1) or (not square? rb2) _
    or (not square? rbinv2) =>
    error "moduleSum: matrices must be square"
  (not square? rb1) or (not square? rbinv1) or (not square? rb2) _
    or (not square? rbinv2) =>
    error "moduleSum: matrices of incompatible dimensions"
  (zero? rbden1) or (zero? rbden2) =>
    error "moduleSum: denominator must be non-zero"
  den := lcm(rbden1, rbden2); c1 := den quo rbden1; c2 := den quo rbden2
  rb := squareTop rowEchelon(vertConcat(c1 * rb1, c2 * rb2), den)
  rbinv := UpTriBddDenomInv(rb, den)
[rb, den, rbinv]

package IBPTOOLS IntegralBasisPolynomialTools

— IBATOOL.dotabb —

"IBATOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IBATOOL"]
"FRAMALG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FRAMALG"]
"IBATOOL" -> "FRAMALG"

— IntegralBasisPolynomialTools.input —

)set break resume
)sys rm -f IntegralBasisPolynomialTools.output
)spool IntegralBasisPolynomialTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegralBasisPolynomialTools
--R
--R IntegralBasisPolynomialTools(K: Ring,R: UnivariatePolynomialCategory(K),UP: UnivariatePolynomialCategory(K),L: Ring) is a package constructor
--R Abbreviation for IntegralBasisPolynomialTools is IBPTOOLS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IBPTOOLS
--R
--R--------------------------------------------- Operations ---------------------------------------------
--R mapBivariate : ((K -> L),UP) -> SparseUnivariatePolynomial(SparseUnivariatePolynomial(L))
--R mapMatrixIfCan : ((L -> Union(K,"failed")),Matrix(SparseUnivariatePolynomial(L))) -> Union(Matrix(R),"failed")
--R mapUnivariate : ((L -> K),SparseUnivariatePolynomial(L)) -> R
--R mapUnivariate : ((K -> L),R) -> SparseUnivariatePolynomial(L)
--R mapUnivariateIfCan : ((L -> Union(K,"failed")),SparseUnivariatePolynomial(L)) -> Union(R,"failed")
--R

)spool
)lisp (bye)

---

— IntegralBasisPolynomialTools.help —

====================================================================
IntegralBasisPolynomialTools examples
====================================================================

IntegralBasisPolynomialTools provides functions for mapping functions on the coefficients of univariate and bivariate polynomials.

See Also:
  o )show IntegralBasisPolynomialTools
IntegralBasisPolynomialTools (IBPTOOLS)

Exports:
mapBivariate mapMatrixIfCan mapUnivariate mapUnivariateIfCan

— package IBPTOOLS IntegralBasisPolynomialTools —

)abbrev package IBPTOOLS IntegralBasisPolynomialTools
++ Author: Clifton Williamson
++ Date Created: 13 August 1993
++ Date Last Updated: 17 August 1993
++ Description:
++ IntegralBasisPolynomialTools provides functions for mapping functions
++ on the coefficients of univariate and bivariate polynomials.

IntegralBasisPolynomialTools(K,R,UP,L): Exports == Implementation where
K : Ring
R : UnivariatePolynomialCategory K
UP : UnivariatePolynomialCategory R
L : Ring

MAT ==> Matrix
SUP ==> SparseUnivariatePolynomial

Exports => with
mapUnivariate: (L -> K,SUP L) -> R
  ++ mapUnivariate(f,p(x)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x)}.

mapUnivariate: (K -> L,R) -> SUP L
  ++ mapUnivariate(f,p(x)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x)}.

mapUnivariateIfCan: (L -> Union(K,"failed"),SUP L) -> Union(R,"failed")
  ++ mapUnivariateIfCan(f,p(x)) applies the function \spad{f} to the
  ++ coefficients of \spad{p(x)}, if possible, and returns
  ++ \spad{"failed"} otherwise.
mapMatrixIfCan: (L \rightarrow \text{Union}(K, "failed"), \text{MAT SUP} L) \rightarrow \text{Union}(\text{MAT} R, "failed")
++ mapMatrixIfCan(f, mat) applies the function \text{\textbackslash spad\text{f}} to the
++ coefficients of the entries of \text{\textbackslash spad\{mat\}} if possible, and returns
++ \text{\textbackslash spad\{"failed\}} otherwise.

mapBivariate: (K \rightarrow L, \text{UP}) \rightarrow \text{SUP SUP} L
++ mapBivariate(f, p(x,y)) applies the function \text{\textbackslash spad\{f\}} to the
++ coefficients of \text{\textbackslash spad\{p(x,y)\}}.

Implementation ==> add

mapUnivariate(f:L \rightarrow K, poly: \text{SUP} L) ==
ans : R := 0
while not zero? poly repeat
  ans := ans + monomial(f leadingCoefficient poly, degree poly)
poly := reductum poly
ans

mapUnivariate(f:K \rightarrow L, poly:R) ==
ans : \text{SUP} L := 0
while not zero? poly repeat
  ans := ans + monomial(f leadingCoefficient poly, degree poly)
poly := reductum poly
ans

mapUnivariateIfCan(f, poly) ==
ans : R := 0
while not zero? poly repeat
  (lc := f leadingCoefficient poly) case "failed" => return "failed"
  ans := ans + monomial(lc :: K, degree poly)
poly := reductum poly
ans

mapMatrixIfCan(f, mat) ==
m := nrows mat; n := ncols mat
matOut : MAT R := new(m, n, 0)
for i in 1 .. m repeat for j in 1 .. n repeat
  (poly := mapUnivariateIfCan(f, qelt(mat, i, j))) case "failed" =>
    return "failed"
  qsetelt_!(matOut, i, j, poly :: R)
matOut

mapBivariate(f, poly) ==
ans : SUP SUP L := 0
while not zero? poly repeat
  ans :=
    ans + monomial(mapUnivariate(f, leadingCoefficient poly), degree poly)
poly := reductum poly
ans
package IR2 IntegrationResultFunctions2

IntegrationResultFunctions2.input

)set break resume
)sys rm -f IntegrationResultFunctions2.output
)spool IntegrationResultFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegrationResultFunctions2
--R
--R IntegrationResultFunctions2(E: Field,F: Field) is a package constructor
--R Abbreviation for IntegrationResultFunctions2 is IR2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IR2
--R
--R-------------------------------------------------- Operations -------------------------------
--R map : ((E -> F),IntegrationResult(E)) -> IntegrationResult(F)
--R map : ((E -> F),Union(Record(ratpart: E,coeff: E),"failed")) -> Union(Record(ratpart: F,coeff: E),"failed")
--R map : ((E -> F),Union(E,"failed")) -> Union(F,"failed")
--R map : ((E -> F),Union(Record(mainpart: E,limitedlogs: List(Record(coefficient: E,logand: E))),"failed")) -> Union(F,"failed")
--R
--E 1

)spool
)lisp (bye)
IntegrationResultFunctions2 examples
====================================================================

Internally used by the integration packages

See Also:
- )show IntegrationResultFunctions2

---

IntegrationResultFunctions2 (IR2)

Exports:
map

---

)abbrev package IR2 IntegrationResultFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 August 1992
++ Description:
++ Internally used by the integration packages

IntegrationResultFunctions2(E, F): Exports == Implementation where
  E : Field
  F : Field

  SE  ==> Symbol
  Q   ==> Fraction Integer
  IRE ==> IntegrationResult E
  IRF ==> IntegrationResult F
UPE ==> SparseUnivariatePolynomial E
UPF ==> SparseUnivariatePolynomial F
NEE ==> Record(integrand:E, intvar:E)
NEF ==> Record(integrand:F, intvar:F)
LGE ==> Record(scalar:Q, coeff:UPE, logand:UPE)
LGF ==> Record(scalar:Q, coeff:UPF, logand:UPF)
NLE ==> Record(coeff:E, logand:E)
NLF ==> Record(coeff:F, logand:F)
UFE ==> Union(Record(mainpart:E, limitedlogs:List NLE), "failed")
URE ==> Union(Record(ratpart:E, coeff:E), "failed")
UE ==> Union(E, "failed")

Exports ==> with
   map: (E -> F, IRE) -> IRF
   ++ map(f,ire) \undocumented
   map: (E -> F, URE) -> Union(Record(ratpart:F, coeff:F), "failed")
   ++ map(f,ure) \undocumented
   map: (E -> F, UE) -> Union(F, "failed")
   ++ map(f,ue) \undocumented
   map: (E -> F, UFE) ->
      Union(Record(mainpart:F, limitedlogs:List NLF), "failed")
   ++ map(f,ufe) \undocumented

Implementation ==> add
   import SparseUnivariatePolynomialFunctions2(E, F)
   NEE2F: (E -> F, NEE) -> NEF
   LGE2F: (E -> F, LGE) -> LGF
   NLE2F: (E -> F, NLE) -> NLF
   NLE2F(func, r) == [func(r.coeff), func(r.logand)]
   NEE2F(func, n) == [func(n.integrand), func(n.intvar)]
   map(func:E -> F, u:UE) == (u case "failed" => "failed"; func(u::E))
   map(func:E -> F, ir:IRE) ==
      mkAnswer(func ratpart ir, [LGE2F(func, f) for f in logpart ir],
                [NEE2F(func, g) for g in notelem ir])
   map(func:E -> F, u:URE) ==
      u case "failed" => "failed"
      [func(u.ratpart), func(u.coeff)]
   map(func:E -> F, u:UFU) ==
      u case "failed" => "failed"
      [func(u.mainpart), [NLE2F(func, f) for f in u.limitedlogs]]
   LGE2F(func, lg) ==
      [lg.scalar, map(func, lg.coeff), map(func, lg.logand)]
---

- IR2.dotabb

"IR2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IR2"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"IR2" -> "FIELD"

---

package IRRF2F IntegrationResultRFToFunction

--- IntegrationResultRFToFunction.input ---

)set break resume
)sys rm -f IntegrationResultRFToFunction.output
)spool IntegrationResultRFToFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegrationResultRFToFunction
--R IntegrationResultRFToFunction(R: Join(GcdDomain,RetractableTo(Integer),OrderedSet,LinearlyExplicitRingOver(Integer))) is a package constructor
--R Abbreviation for IntegrationResultRFToFunction is IRRF2F
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IRRF2F
--R
--R------------------------------------------ Operations ------------------------------------------
--R complexExpand : IntegrationResult(Fraction(Polynomial(R))) -> Expression(R)
--R complexIntegrate : (Fraction(Polynomial(R)),Symbol) -> Expression(R) if R has CHARZ
--R expand : IntegrationResult(Fraction(Polynomial(R))) -> List(Expression(R))
--R integrate : (Fraction(Polynomial(R)),Symbol) -> Union(Expression(R),List(Expression(R))) if R has CHARZ
--R split : IntegrationResult(Fraction(Polynomial(R))) -> IntegrationResult(Fraction(Polynomial(R)))
--R
--E 1

)spool
)lisp (bye)

---

- IntegrationResultRFToFunction.help

---
IntegrationResultRFToFunction examples

Conversion of integration results to top-level expressions. This package allows a sum of logs over the roots of a polynomial to be expressed as explicit logarithms and arc tangents, provided that the indexing polynomial can be factored into quadratics.

See Also:
- \( \text{show IntegrationResultRFToFunction} \)

---

IntegrationResultRFToFunction (IRRF2F)

Exports:
- complexExpand
- complexIntegrate
- expand
- integrate
- split

— package IRRF2F IntegrationResultRFToFunction —

)abbrev package IRRF2F IntegrationResultRFToFunction
++ Author: Manuel Bronstein
++ Date Created: 21 August 1988
++ Date Last Updated: 4 October 1993
++ Description:
++ Conversion of integration results to top-level expressions.
++ This package allows a sum of logs over the roots of a polynomial to be expressed as explicit logarithms and arc tangents, provided that the indexing polynomial can be factored into quadratics.

IntegrationResultRFToFunction(R):Exports==Implementation where
R:Join(GcdDomain, RetractableTo Integer, OrderedSet),
LinearlyExplicitRingOver Integer)

RF ==> Fraction Polynomial R
F ==> Expression R
IR ==> IntegrationResult RF

Exports ==> with
split : IR -> IR
  ++ split(u(x) + sum_{P(a)=0} Q(a,x)) returns
  ++ \spad{u(x) + sum_{P1(a)=0} Q(a,x) + ... + sum_{Pn(a)=0} Q(a,x)}
  ++ where P1,...,Pn are the factors of P.
expand : IR -> List F
  ++ expand(i) returns the list of possible real functions
  ++ corresponding to i.
complexExpand : IR -> F
  ++ complexExpand(i) returns the expanded complex function
  ++ corresponding to i.
if R has CharacteristicZero then
  integrate : (RF, Symbol) -> Union(F, List F)
  ++ integrate(f, x) returns the integral of \spad{f(x)dx}
  ++ where x is viewed as a real variable.
  complexIntegrate: (RF, Symbol) -> F
  ++ complexIntegrate(f, x) returns the integral of \spad{f(x)dx}
  ++ where x is viewed as a complex variable.

Implementation ==> add
import IntegrationTools(R, F)
import TrigonometricManipulations(R, F)
import IntegrationResultToFunction(R, F)
toEF: IR -> IntegrationResult F

toEF i == map(z1+->z1::F, i)$IntegrationResultFunctions2(RF, F)
expand i == expand toEF i
complexExpand i == complexExpand toEF i

split i ==
  map(retract, split toEF i)$IntegrationResultFunctions2(F, RF)
if R has CharacteristicZero then
  import RationalFunctionIntegration(R)
  complexIntegrate(f, x) == complexExpand internalIntegrate(f, x)
-- do not use real integration if R is complex
if R has imaginary: () -> R then integrate(f, x) == complexIntegrate(f, x)
else
  integrate(f, x) ==
    l := [mkPrim(real g, x) for g in expand internalIntegrate(f, x)]
    empty? rest l => first l
package IR2F IntegrationResultToF

--- IntegrationResultToF.input ---

)set break resume
)sys rm -f IntegrationResultToF.output
)spool IntegrationResultToF.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegrationResultToF
--R
--R IntegrationResultToF(R: Join(GcdDomain,RetractableTo(Integer),OrderedSet,LinearlyExplicitRingOver(Integer)),F: Join(AlgebraicallyClosedFunctionSpace(R),TranscendentalFunctionCategory)) is a package constructor
--R Abbreviation for IntegrationResultToF is IR2F
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IR2F
--R
--R----------------------------------- Operations -----------------------------------
--R complexExpand : IntegrationResult(F) -> F
--R expand : IntegrationResult(F) -> List(F)
--R split : IntegrationResult(F) -> IntegrationResult(F)
--R
--E 1

)spool
)lisp (bye)

--- IntegrationResultToF.help ---
IntegrationResultToFunction examples

Conversion of integration results to top-level expressions
This package allows a sum of logs over the roots of a polynomial
to be expressed as explicit logarithms and arc tangents, provided
that the indexing polynomial can be factored into quadratics.

See Also:
o )show IntegrationResultToFunction

IntegrationResultToFunction (IR2F)

Exports:
complexExpand expand split

— package IR2F IntegrationResultToFunction —

)abbrev package IR2F IntegrationResultToFunction
++ Author: Manuel Bronstein
++ Date Created: 4 February 1988
++ Date Last Updated: 9 October 1991
++ Description:
++ Conversion of integration results to top-level expressions
++ This package allows a sum of logs over the roots of a polynomial
++ to be expressed as explicit logarithms and arc tangents, provided
++ that the indexing polynomial can be factored into quadratics.

IntegrationResultToFunction(R, F):Exports == Implementation where
R: Join(GcdDomain, RetractableTo Integer, OrderedSet,
LinearlyExplicitRingOver Integer)
F: Join(AlgebraicallyClosedFunctionSpace R,
      TranscendentalFunctionCategory)
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Z
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
IR ==> IntegrationResult F
REC ==> Record(ans1:F, ans2:F)
LOG ==> Record(scalar:Q, coeff:UP, logand:UP)

Exports ==> with
    split : IR -> IR
      ++ split(u(x) + sum_{P(a)=0} Q(a,x)) returns
      ++ \spad{u(x) + sum_{P1(a)=0} Q(a,x) + \ldots + sum_{Pn(a)=0} Q(a,x)}
      ++ where P1,...,Pn are the factors of P.
    expand : IR -> List F
      ++ expand(i) returns the list of possible real functions
      ++ corresponding to i.
    complexExpand: IR -> F
      ++ complexExpand(i) returns the expanded complex function
      ++ corresponding to i.

Implementation ==> add
    import AlgebraicManipulations(R, F)
    import ElementaryFunctionSign(R, F)

IR2F : IR -> F
insqrt : F -> Record(sqrt:REC, sgn:Z)
pairsum : (List F, List F) -> List F
pairprod : (F, List F) -> List F
quadeval : (UP, F, F, F) -> REC
linear : (UP, UP) -> F
tantrick : (F, F) -> F
ilog : (F, F, List K) -> F
nlogs : LOG -> List LOG
lg2func : LOG -> List F
quadratic : (UP, UP) -> List F
mkRealFunc : List LOG -> List F
lg2cfunc : LOG -> F
loglist : (Q, UP, UP) -> List LOG
cmplex : (F, UP) -> F
evenRoots : F -> List F
compatible?: (List F, List F) -> Boolean

cmplex(alpha, p) == alpha * log p alpha
IR2F i  == retract mkAnswer(ratpart i, empty(), notelem i)
pairprod(x, l)  == [x * y for y in l]

evenRoots x ==
[first argument k for k in tower x | is?(k,"nthRoot"::Symbol) and even?(retract(second argument k)@Z) and (not empty? variables first argument k)]

expand i ==
j := split i
pairsum([IR2F j], mkRealFunc logpart j)

split i ==
mkAnswer(ratpart i, concat [nlogs l for l in logpart i], notelem i)

complexExpand i ==
j := split i
IR2F j + [/lg.scalar::F * lg2cfunc lg for lg in logpart j]

-- p = a t^2 + b t + c
-- Expands sum_{p(t) = 0} t log(lg(t))
quadratic(p, lg) ==
zero?(delta := (b := coefficient(p, 1))**2 - 4 *
(a := coefficient(p,2)) * (p0 := coefficient(p, 0))) =>
 [linear(monomial(1, 1) + (b / a)::UP, lg)]
e := (q := quadeval(lg, c := - b * (d := inv(2*a)),d, delta)).ans1
lgp := c * log(nrm := (e**2 - delta * (f := q.ans2)**2))
s := (sqr := insqrt delta).sqrt
pp := nn := 0$F
if sqr.sgn >= 0 then
  sqrp := s.ans1 * rootSimp sqrt(s.ans2)
  pp := lgp + d * sqrp * log(((2 * e + f) / nrm) * sqrp
   + (e**2 + delta * f**2) / nrm)
if sqr.sgn <= 0 then
  sqrn := s.ans1 * rootSimp sqrt(-s.ans2)
nn := lgp + d * sqrn * ilog(e, f * sqrn,
   setUnion(setUnion(kernels a, kernels b), kernels p0))
sqr.sgn > 0 => [pp]
sqr.sgn < 0 => [nn]
[pp, nn]

-- returns 2 atan(a/b) or 2 atan(-b/a) whichever looks better
-- they differ by a constant so it's ok to do it from an IR
tantrick(a, b) ==
  retractIfCan(a@Union(Q, "failed") case Q => 2 * atan(-b/a)
2 * atan(a/b)

-- transforms i log((a + i b) / (a - i b)) into a sum of real
-- arc-tangents using Rioboo’s algorithm
-- 1k is a list of kernels which are parameters for the integral
\begin{verbatim}
ilog(a, b, lk) ==
  l := setDifference(setUnion(variables numer a, variables numer b),
    setUnion(lk, setUnion(variables denom a, variables denom b)))
  empty? l => tantrick(a, b)
  k := "max"/l
  ilog0(a, b, numer univariate(a, k), numer univariate(b, k), k::F)

-- transforms log((a + i b) / (a - i b)) into a sum of real
-- arc-tangents using Rioboo's algorithm
-- the arc-tangents will not have k in the denominator
-- we always keep upa(k) = a and upb(k) = b
ilog0(a, b, upa, upb, k) ==
  if degree(upa) < degree(upb) then
    (upa, upb) := (-upb, upa)
    (a, b) := (-b, a)
  zero? degree upb => tantrick(a, b)
  r := extendedEuclidean(upa, upb)
  (g:= retractIfCan(r.generator)@Union(F,"failed")) case "failed" =>
    tantrick(a, b)
  if degree(r.coef1) >= degree upb then
    qr := divide(r.coef1, upb)
    r.coef1 := qr.remainder
    r.coef2 := r.coef2 + qr.quotient * upa
    aa := (r.coef2) k
    bb := -(r.coef1) k
    tantrick(aa * a + bb * b, g::F) + ilog0(aa,bb,r.coef2,-r.coef1,k)

lg2func lg ==
  zero?(d := degree(p := lg.coeff)) => error "poly has degree 0"
  (d = 1) => [linear(p, lg.logand)]
  d = 2 => quadratic(p, lg.logand)
  odd? d and
    (r := retractIfCan(reductum p)@Union(F, "failed")) case F) =>
      pairsum([cmplex(alpha := rootSimp zeroOf p, lg.logand)],
        lg2func [lg.scalar,
          (p exquo (monomial(1, 1)$UP - alpha::UP))::UP,
          lg.logand])
    [lg2func lg]
lg2func lg ==
  +/-[cmplex(alpha, lg.logand) for alpha in zerosOf(lg.coeff)]

mkRealFunc l ==
  ans := empty()$List(F)
  for lg in l repeat
    ans := pairsum(ans, pairprod(lg.scalar::F, lg2func lg))
  ans

-- returns a log(b)
\end{verbatim}
linear(p, lg) ==
    alpha := - coefficient(p, 0) / coefficient(p, 1)
    alpha * log lg alpha
-- returns (c, d) s.t. p(a + b t) = c + d t, where t^2 = delta
quadeval(p, a, b, delta) ==
    zero? p => [0, 0]
    bi := c := d := 0$F
    ai := 1$F
    v := vectorise(p, 1 + degree p)
    for i in minIndex v .. maxIndex v repeat
        c := c + qelt(v, i) * ai
        d := d + qelt(v, i) * bi
        temp := a * ai + b * bi * delta
        bi := a * bi + b * ai
        ai := temp
    [c, d]

compatible?(lx, ly) ==
    empty? ly => true
    for x in lx repeat
        for y in ly repeat
            ((s := sign(x*y)) case Z) and (s::Z < 0) => return false
    true

pairsum(lx, ly) ==
    empty? lx => ly
    empty? ly => lx
    l := empty()$List(F)
    for x in lx repeat
        ls := evenRoots x
        if not empty?(ln :=
            [x + y for y in ly | compatible?(ls, evenRoots y)]) then
            l := removeDuplicates concat(l, ln)
    l

-- returns [[a, b], s] where sqrt(y) = a sqrt(b) and
-- s = 1 if b > 0, -1 if b < 0, 0 if the sign of b cannot be determined
insqrt y ==
    rec := froot(y, 2)$PolynomialRoots(IndexedExponents K, K, R, P, F)
    -- one?(rec.exponent) => [[rec.coef * rec.radicand, 1], 1]
    (rec.exponent = 1) => [[rec.coef * rec.radicand, 1], 1]
    rec.exponent ^=2 => error "Should not happen"
    [[rec.coef, rec.radicand],
     ((s := sign(rec.radicand)) case "failed" => 0; s::Z)]

nlogs lg ==
    [[f.exponent * lg.scalar, f.factor, lg.logand] for f in factors
        ffactor(primitivePart(lg.coeff))
    ]$FunctionSpaceUnivariatePolynomialFactor(R, F, UP)]
package INTTOOLS IntegrationTools

IntegrationTools.input

)set break resume
)sys rm -f IntegrationTools.output
)spool IntegrationTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IntegrationTools
--R
--R IntegrationTools(R: OrderedSet,F: FunctionSpace(R)) is a package constructor
--R Abbreviation for IntegrationTools is INTTOOLS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTTOOLS
--R
--R------------------------------- Operations --------------------------------
--R kmax : List(Kernel(F)) -> Kernel(F)
--R intPatternMatch : (F,Symbol,((F,Symbol) -> IntegrationResult(F)),((F,Symbol) -> Union(Record(special: F,integrand: ... ELEMFUN and F has LFCAT and F has RETRACT(SYMBOL) and R has KONVERT(PATTERN(INT)) and R has GCDDOM and R has PATMAB(INT)
--R ksec : (Kernel(F),List(Kernel(F)),Symbol) -> Kernel(F)
--R mkPrim : (F,Symbol) -> F if F has ELEMFUN and R has GCDDOM
--R removeConstantTerm : (F,Symbol) -> F if R has INTDOM
--R union : (List(Kernel(F)),List(Kernel(F))) -> List(Kernel(F))
--R vark : (List(F),Symbol) -> List(Kernel(F))
--R varselect : (List(Kernel(F)),Symbol) -> List(Kernel(F))
--R
--E 1

)spool
)lisp (bye)
---

--- IntegrationTools.help ---

IntegrationTools examples

Tools for the integrator

See Also:
  o )show IntegrationTools

---

IntegrationTools (INTTOOLS)

Exports:
  kmax  intPatternMatch  ksec  mkPrim  removeConstantTerm
  union  vark  varselect

--- package INTTOOLS IntegrationTools ---

)abbrev package INTTOOLS IntegrationTools
++ Author: Manuel Bronstein
++ Date Created: 25 April 1990
++ Date Last Updated: 9 June 1993
++ Description:
++ Tools for the integrator

IntegrationTools(R:OrderedSet, F:FunctionSpace R): Exp == Impl where
  K ==> Kernel F
  SE ==> Symbol
CHAPTER 10. CHAPTER I

P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial F
IR ==> IntegrationResult F
ANS ==> Record(special:F, integrand:F)
U ==> Union(ANS, "failed")
ALGOP ==> "%alg"

Exp ==> with
   varselect: (List K, SE) -> List K
   ++ varselect([k1,...,kn], x) returns the ki which involve x.
kmax : List K -> K
   ++ kmax([k1,...,kn]) returns the top-level ki for integration.
ksec : (K, List K, SE) -> K
   ++ ksec(k, [k1,...,kn], x) returns the second top-level ki
   ++ after k involving x.
union : (List K, List K) -> List K
   ++ union(l1, l2) returns set-theoretic union of l1 and l2.
vark : (List F, SE) -> List K
   ++ vark([f1,...,fn],x) returns the set-theoretic union of
   ++ \spad{varselect(f1,x),...,varselect(fn,x)}.
if R has IntegralDomain then
   removeConstantTerm: (F, SE) -> F
   ++ removeConstantTerm(f, x) returns f minus any additive constant
   ++ with respect to x.
if R has GcdDomain and F has ElementaryFunctionCategory then
   mkPrim: (F, SE) -> F
   ++ mkPrim(f, x) makes the logs in f which are linear in x
   ++ primitive with respect to x.
if R has ConvertibleTo Pattern Integer and R has PatternMatchable Integer
   and F has LiouvillianFunctionCategory and F has RetractableTo SE then
   intPatternMatch: (F, SE, (F, SE) -> IR, (F, SE) -> U) -> IR
   ++ intPatternMatch(f, x, int, pmint) tries to integrate \spad{f}
   ++ first by using the integration function \spad{int}, and then
   ++ by using the pattern match integration function \spad{pmint}
   ++ on any remaining unintegrable part.

Impl ==> add
   better?: (K, K) -> Boolean
   union(l1, l2) == setUnion(l1, l2)
   varselect(l, x) == [k for k in l | member?(x, variables(k::F))]
   ksec(k, l, x) == kmax setUnion(remove(k, l), vark(argument k, x))

vark(l, x) ==
   varselect(reduce("setUnion", [kernels f for f in l], empty()$List(K)), x)

kmax l ==
   ans := first l
   for k in rest l repeat
      if better?(k, ans) then ans := k
ans

-- true if x should be considered before y in the tower
better?(x, y) ==
    height(y) ^= height(x) => height(y) < height(x)
has?(operator y, ALGOP) or
    (is?(y, "exp":SE) and not is?(x, "exp":SE)
     and not has?(operator x, ALGOP))

if R has IntegralDomain then
    removeConstantTerm(f, x) ==
        not freeOf?((den := denom f)::F, x) => f
        (u := isPlus(num := numer f)) case "failed" =>
            freeOf?(num::F, x) => 0
            f
        ans:P := 0
        for term in u::List(P) repeat
            if not freeOf?(term::F, x) then ans := ans + term
        ans / den

if R has GcdDomain and F has ElementaryFunctionCategory then
    psimp : (P, SE) -> Record(coef:Integer, logand:F)
    cont : (P, List K) -> P
    logsimp : (F, SE) -> F
    linearLog?: (K, F, SE) -> Boolean

    logsimp(f, x) ==
        r1 := psimp(numer f, x)
        r2 := psimp(denom f, x)
        g := gcd(r1.coef, r2.coef)
        g * log(r1.logand ** (r1.coef quo g) / r2.logand ** (r2.coef quo g))

    cont(p, l) ==
        empty? l => p
        q := univariate(p, first l)
        cont(unitNormal(leadingCoefficient q).unit * content q, rest l)

    linearLog?(k, f, x) ==
        is?(k, "log":SE) and
        ((u := retractIfCan(univariate(f,k))@Union(UP,"failed")) case UP)
        and one?(degree(u::UP))
        and (degree(u::UP) = 1)
        and not member?(x, variables leadingCoefficient(u::UP))

    mkPrim(f, x) ==
        lg := [k for k in kernels f | linearLog?(k, f, x)]
        eval(f, lg, [logsimp(first argument k, x) for k in lg])

    psimp(p, x) ==
        (u := isExpt(p := ((p exquo cont(p, varselect(variables p, x)))::P)))
case "failed" => [1, p::F]
[u.exponent, u.var::F]

if R has Join(ConvertibleTo Pattern Integer, PatternMatchable Integer)
and F has Join(LiouvilleFunctionCategory, RetractableTo SE) then
  intPatternMatch(f, x, int, pmint) ==
    ir := int(f, x)
    empty?(l := notelem ir) => ir
    ans := ratpart ir
    nl:List(Record(integrand:F, intvar:F)) := empty()
    lg := logpart ir
    for rec in l repeat
      u := pmint(rec.integrand, retract(rec.intvar))
      if u case ANS then
        rc := u::ANS
        ans := ans + rc.special
        if rc.integrand ^= 0 then
          ir0 := intPatternMatch(rc.integrand, x, int, pmint)
          ans := ans + ratpart ir0
          lg := concat(logpart ir0, lg)
          nl := concat(notelem ir0, nl)
        else nl := concat(rec, nl)
      fi
    od
    mkAnswer(ans, lg, nl)

— INTTOOLS.dotabb —

"INTTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTTOOLS"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTTOOLS" -> "FS"

package IPRNTPK InternalPrintPackage

— InternalPrintPackage.input —

)set break resume
)sys rm -f InternalPrintPackage.output
)spool InternalPrintPackage.output
)set message test on
)set message auto off
)clear all
InternalPrintPackage (IPRNTPK)

---

InternalPrintPackage examples

A package to print strings without line-feed nor carriage-return.

See Also:
- )show InternalPrintPackage
Exports:

iprint

— package IPRNTPK InternalPrintPackage —

)abbrev package IPRNTPK InternalPrintPackage
++ Author: Themos Tsikas
++ Date Created: 09/09/1998
++ Date Last Updated: 09/09/1998
++ Description:
++ A package to print strings without line-feed nor carriage-return.

InternalPrintPackage(): Exports == Implementation where

Exports == with
  iprint: String -> Void
    ++ \axiom{iprint(s)} prints \axiom{s} at the current position
    ++ of the cursor.

Implementation == add
  iprint(s: String) ==
    PRINC(coerce(s)@Symbol)$Lisp
    FORCE_-OUTPUT()$Lisp

— IPRNTPK.dotabb —

"IPRNTPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IPRNTPK"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"IPRNTPK" -> "ALIST"

——

package IRURPK InternalRationalUnivariateRepresentationPackage

— InternalRationalUnivariateRepresentationPackage.input —

)set break resume
)sys rm -f InternalRationalUnivariateRepresentationPackage.output
)spool InternalRationalUnivariateRepresentationPackage.output
)set message test on
An internal package for computing the rational univariate representation
of a zero-dimensional algebraic variety given by a square-free
triangular set. The main operation is rur.

See Also:
  o )show InternalRationalUnivariateRepresentationPackage
InternalRationalUnivariateRepresentationPackage (IRURPK)

Exports:

checkRur rur

— package IRURPK InternalRationalUnivariateRepresentationPackage —

)abbrev package IRURPK InternalRationalUnivariateRepresentationPackage
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ References:
++ Description:

InternalRationalUnivariateRepresentationPackage(R,E,V,P,TS): Exports == Implementation where
R : Join(EuclideanDomain,CharacteristicZero)
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : SquareFreeRegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LV ==> List V
LP ==> List P
PWT ==> Record(val: P, tower: TS)
LPWT ==> Record(val: LP, tower: TS)
WIP ==> Record(pol: P, gap: Z, tower: TS)
BWT ==> Record(val:Boolean, tower: TS)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
normpack ==> NormalizationPackage(R,E,V,P,TS)

Exports == with
rur: (TS,B) -> List TS
++ \spad{rur(ts,univ?)} returns a rational univariate representation
++ of \spad{ts}. This assumes that the lowest polynomial in \spad{ts}
++ is a variable \spad{v} which does not occur in the other polynomials
++ of \spad{ts}. This variable will be used to define the simple
++ algebraic extension over which these other polynomials will be
++ rewritten as univariate polynomials with degree one.
++ If \spad{univ?} is \spad{true} then these polynomials will have
++ a constant initial.

checkRur: (TS, List TS) -> Boolean
++ \spad{checkRur(ts,lus)} returns \spad{true} if \spad{lus}
++ is a rational univariate representation of \spad{ts}.

Implementation == add

convert(p:P,sqfr?:B):TS ==
  -- if sqfr? ASSUME p is square-free
  newts: TS := empty()
  sqfr? => internalAugment(p,newts)
  p := squareFreePart(p)
  internalAugment(p,newts)

prepareRur(ts: TS): List LPWT ==
  not purelyAlgebraic?(ts)$TS =>
    error "rur$IRURPK: #1 is not zero-dimensional"
  lp: LP := parts(ts)$TS
  lp := sort(infRittWu?,lp)
  empty? lp =>
    error "rur$IRURPK: #1 is empty"
  f0 := first lp; lp := rest lp
  not (one?(init(f0)) and one?(mdeg(f0)) and zero?(tail(f0))) =>
  not ((init(f0) = 1) and (mdeg(f0) = 1) and zero?(tail(f0))) =>
    error "rur$IRURPK: #1 has generating root but no indeterminates"
z: V := mvar(f0)
f1 := first lp; lp := rest lp
x1: V := mvar(f1)
newf1 := x1::P - z::P
toSave: List LPWT := []
for ff1 in irreducibleFactors([f1])$polsetpack repeat
  newf0 := eval(ff1,mvar(f1),f0)
  ts := internalAugment(newf1,convert(newf0,true)@TS)
  toSave := cons([lp,ts],toSave)
toSave

-- ASSUME r is a irreducible univariate polynomial in z
-- ASSUME c and s only depends on z and mvar(s)
-- ASSUME c and a have main degree 1
-- ASSUME c and s have a constant initial
-- ASSUME mvar(ts) < mvar(s)
lp: LP := parts(ts)
lp := sort(infRittWu?,lp)
newts: TS := convert(r,true)@TS
s := remainder(s,newts).polnum
if univ?
  then
    s := normalizedAssociate(s,newts)$normpack
for p in lp repeat
  p := lazyPrem(eval(p,z,c),s)
  p := remainder(p,newts).polnum
newts := internalAugment(p,newts)
internalAugment(s,newts)

next(lambda:Z):Z ==
  if lambda < 0 then lambda := - lambda + 1 else lambda := - lambda

-- if check? THEN some VERIFICATIONS are performed
-- if info? THEN some INFORMATION is displayed
f0 := last(ts)::P
z: V := mvar(f0)
lambda: Z := 1
ts := collectUpper(ts,z)
toSee: List WIP := [[f0,lambda,ts]$WIP]
toSave: List TS := []
while not empty? toSee repeat
  wip := first toSee; toSee := rest toSee
  (f0, lambda, ts) := (wip.pol, wip.gap, wip.tower)
  if check? and ((not univariate?(f0)$polsetpack) or (mvar(f0) ~= z))
    then
      output("Bad f0: ")$OutputPackage
      output(f0::OutputForm)$OutputPackage
  c: P := lambda * xi::P + z::P
f := eval(f0,z,c); q := eval(p,z,c)
prs := subResultantChain(q,f)
r := first prs; prs := rest prs
check? and ((not zero? degree(r,xi)) or (empty? prs)) =>
    error "rur$IRURPK: should never happen !"
s := first prs; prs := rest prs
check? and (zero? degree(s,xi)) and (empty? prs) =>
    error "rur$IRURPK: should never happen !!"
if zero? degree(s,xi) then s := first prs
-- not one? degree(s,xi) =>
not (degree(s,xi) = 1) =>
toSee := cons([f0,next(lambda),ts]$WIP,toSee)
h := init(s)
r := squareFreePart(r)
ground?(h) or ground?(gcd(h,r)) =>
    for fr in irreducibleFactors([r])$polsetpack repeat
        ground? fr => "leave"
toSave := cons(makeMonic(z,c,fr,ts,s,univ?),toSave)
if info?
    then
        output("Unlucky lambda")$OutputPackage
        output(h::OutputForm)$OutputPackage
        output(r::OutputForm)$OutputPackage
    toSee := cons([f0,next(lambda),ts]$WIP,toSee)
toSave

rur (ts: TS,univ?:Boolean): List TS ==
toSee: List LPWT := prepareRur(ts)
toSave: List TS := []
while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
ts: TS := wip.tower
lp: LP := wip.val
empty? lp => toSave := cons(ts,toSave)
p := first lp; lp := rest lp
xi: V := mvar(p)
p := remainder(p,ts).polnum
if not univ?
    then
        p := primitivePart stronglyReduce(p,ts)
ground?(p) or (mvar(p) < xi) =>
            error "rur$IRURPK: should never happen"
-- (one? mdeg(p)) and (ground? init(p)) =>
    (mdeg(p) = 1) and (ground? init(p)) =>
ts := internalAugment(p,ts)
wip := [lp,ts]
toSee := cons(wip,toSee)
lts := makeLinearAndMonic(p,xi,ts,univ?,false,false)
for ts in lts repeat
    wip := [lp,ts]
toSee := cons(wip,toSee)
toSave

---

<table>
<thead>
<tr>
<th>IRURPK.dotabb</th>
</tr>
</thead>
</table>

"IRURPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IRURPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"IRURPK" -> "SFRTCAT"

---

package INTFRSP InterpolateFormsPackage

--- InterpolateFormsPackage.input ---

)set break resume
)sys rm -f InterpolateFormsPackage.output
)spool InterpolateFormsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InterpolateFormsPackage
--R
--R InterpolateFormsPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: ... LocalPowerSeriesCategory(K),Plc: PlacesCategory(K,PCS),DIVISOR: DivisorCategory(Plc)) is a package constructor
--R Abbreviation for InterpolateFormsPackage is INTFRSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTFRSP
--R
--R------------------------------- Operations --------------------------------
--R basisOfInterpolateForms : (DIVISOR,List(PolyRing)) -> List(Vector(K))
--R basisOfInterpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(Vector(K))
--R interpolateForms : (DIVISOR,NonNegativeInteger,PolyRing,List(PolyRing)) -> List(PolyRing)
--R interpolateFormsForFact : (DIVISOR,List(PolyRing)) -> List(PolyRing)
--R
--E 1

)spool
)lisp (bye)

---
--- InterpolateFormsPackage.help ---

====================================================================
InterpolateFormsPackage examples
====================================================================

The following is part of the PAFF package

See Also:
o )show InterpolateFormsPackage

---

InterpolateFormsPackage (INTFRSP)

Exports:
basisOfInterpolateForms basisOfInterpolateFormsForFact
interpolateForms interpolateFormsForFact

--- package INTFRSP InterpolateFormsPackage ---

)abbrev package INTFRSP InterpolateFormsPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
InterpolateFormsPackage(K,symb,PolynomialRing,E,ProjPt,PCS,P1c,DIVISOR):_
Exports == Implementation where
K:Field
symb: List(Symbol)

GV => OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)

PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)

INT ==> Integer
NNI ==> NonNegativeInteger
ParamPack ==> ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
PackPoly ==> PackageForPoly(K,PolyRing,E,#symb)
LINPACK ==> LinearSystemFromPowerSeriesPackage(K,PCS)

Exports ==> with

basisOfInterpolateForms: (DIVISOR,List PolyRing) -> List(Vector(K))

basisOfInterpolateFormsForFact: (DIVISOR,List PolyRing) -> List(Vector(K))

interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

++ interpolateForms(D,n,pol,base) compute the basis of the sub-vector
++ space W of V = <base>, such that for all G in W, the
++ divisor (G) >= D. All the elements in base must be homogeneous
++ polynomial of degree n. Typically, base is the set of all monomial
++ of degree n: in that case, interpolateForms(D,n,pol,base)
++ returns the basis of the vector space of all forms of degree d that
++ interpolated D. The argument pol must be the same polynomial that
++ defined the curve form which the divisor D is defined.

Implementation ==> add

import PolyRing
import PCS

sbSpcOfCurve: (NNI,PolyRing) -> List(List(K))
exponant2monomial: List(NNI) -> PolyRing
crtV: (List(K),List(INT),NNI) -> List(K)
createLinSys: (List Plc, List INT,List PolyRing) -> Matrix(K)
createLinSysWOVectorise: (List Plc, List INT,List PolyRing) -> Matrix(K)
basisOfInterpolateFormsForFact(divis,lm)==
-- permet d'intépoler un diviseur qui n'est pas rationnel.
-- La partie non rationel
-- est dans sptdiv (note: une place de sptdiv est une place qui identifie
-- l'ensemble des places qui lui sont conjuguées.
-- Note: On utilise ici la fonction createLinSysWOVectorise
-- qui ne vectorise pas les éléments du corps de base.
lstOfPlc:= supp divis
lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
-- ppsol contiendra la base des formes interpolant ke diviseur divis
linSys:=Matrix(K)
linSysT:=Matrix(K)
l1:=List Matrix K
"empty?(lstOfPlc) =>
  linSys:=createLinSysWOVectorise(lstOfPlc,lstOfv,lm)
  nullSpace linSys
  zeroMat:=Matrix(K):=zero(1,#lm)$Matrix(K)
  nullSpace zeroMat
interpolateForms(divis,d,laCrb,lm)==
  -- ppsol contiendra la base des formes interpolant le diviseur divis
  -- mieux vaut prendre divOfZero de divis ?
  ppsol:= basisOfInterpolateForms(divis,lm)
  psol:=List(List(K)):=[entries(vec) for vec in ppsol]
  mpsol:=psol
  sbspc:=List(List(K))
  if "^(totalDegree(laCrb)$PackPoly > d) then
    -- retourne une base des formes de degrés d
    sbspc:=sbSpcOfCurve(d,laCrb)
    mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
  empty?(mpsol) => [0]
  rowEchmpsol:=rowEchelon(matrix(mpsol)$Matrix(K))
  npsol:=listOfLists(rowEchmpsol)
  [reduce("+",[a*f for a in ll for f in lm]) for ll in npsol]
interpolateFormsForFact(divis,lm)==
  -- ppsol contiendra la base des formes interpolant le diviseur divis
  ppsol:= basisOfInterpolateFormsForFact(divis,lm)
  psol:=List(List(K)):=[entries(vec) for vec in ppsol]
  mpsol:=psol
  empty?(mpsol) => [0]
  rowEchmpsol:=rowEchelon matrix(mpsol)$Matrix(K)
  npsol:=listOfLists(rowEchmpsol)
  [reduce("+",[a*f for a in ll for f in lm]) for ll in npsol]
createLinSys(lstOfPlc,lstOfv,lm)==
  lplsT:=[ [parametrize(f,pl)$ParamPack for f in lm]
    for pl in lstOfPlc]
  lpls:=[[filterUpTo(s,v) for s in souslplsT] _
for souslplsT in lplsT_
for v in lstOfv]_
linSys:=reduce("vertConcats",_
    [finiteSeries2LinSys(souslplsT,v)$LINPACK_
    for souslplsT in lpls_
for v in lstOfv])
linSys

createLinSysWOVectorise(lstOfPlc,lstOfv,lm)==
    lplsT:=
        [ [parametrize(f,pl)$ParamPack for f in lm]_
        for pl in lstOfPlc]
lpls:=
        [[filterUpTo(s,v) for s in souslplsT]_
        for souslplsT in lplsT_
for v in lstOfv]
linSys:=reduce("vertConcats",_
    [finiteSeries2LinSysWOVectorise(souslplsT,v)$LINPACK_
    for souslplsT in lpls_
for v in lstOfv])
linSys

basisOfInterpolateForms(divis,lm)==
    lstOfPlc:= supp divis
    lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
    -- ppsol contiendra la base des formes interpolant le diviseur divis
    linSys:Matrix(K)
    if empty?(lstOfPlc) =>
        linSys:=createLinSys(lstOfPlc,lstOfv,lm)
    -- ppsol contiendra la base des formes passant par le diviseur divv
    nullSpace(linSys)
    zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
nullSpace zeroMat

interpolateForms(divis,d,laCrb,lm)==
    lstOfPlc:= supp divis
    lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
    -- ppsol contiendra la base des formes interpolant le diviseur divis
    ppsol:List(Vector(K))
    -- linSys:Matrix(K)
    if empty?(lstOfPlc) then
        linSys:=createLinSys(lstOfPlc,lstOfv,lm)
    -- ppsol contiendra la base des formes passant par le diviseur divv
    ppsol:=nullSpace(linSys)
    else
        zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
-- ppsol:=nullSpace zeroMat
-- mpsol:=psol:List(List(K)):=\[entries(vec) for vec in ppsol\]
--
-- if ^(totalDegree(laCrb) > d) then
-- -- retourne une base des formes de degrés d
-- -- qui sont un multiple de la courbe
-- sbspc:=sbSpcOfCurve(d,laCrb)
-- mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
--
-- empty?(mpsol) => [0]
--
-- rowEchmpsol:=rowEchelon(matrix(mpsol))
-- mpsol:=listOfLists(rowEchmpsol)
-- [reduce("+",[a*f for a in ll for f in lm]) for ll in npsol]
--
-- interpolateForms(divis,d,laCrb,lm)==
-- lstOfPlc:= supp divis
-- lstOfv:= [coefficient(pl,divis) for pl in lstOfPlc]
--
-- lpls : List(List(PCS))
-- lplsT: List(List(PCS))
--
-- -- ppsol contiendra la base des formes interpolant le diviseur divis
-- ppsol:=List(Vector(K))
-- linSys:Matrix(K)
-- if ^empty?(lstOfPlc) then
--
-- lplsT:=[ [parametrize(f,pl)$ParamPack for f in lm]_
-- for pl in lstOfPlc]
--
-- lpls:=([[filterUpTo(s,v) for s in souslplsT] _
-- for souslplsT in lplsT]_
-- for v in lstOfv]
--
-- linSys:=reduce("vertConcat",_
-- [finiteSeries2LinSys(souslplsT,v)$LINPACK_
-- for souslplsT in lpls_
-- for v in lstOfv])
--
-- -- ppsol contiendra la base des formes passant par le diviseur divv
-- ppsol:=nullSpace(linSys)
--
-- else
-- zeroMat:Matrix(K):=zero(1,#lm)$Matrix(K)
-- ppsol:=nullSpace zeroMat
-- mpsol:=psol:List(List(K)):=\[entries(vec) for vec in ppsol\]
--
-- if ^(totalDegree(laCrb) > d) then
-- -- retourne une base des formes de degrés d
-- -- qui sont un multiple de la courbe
CHAPTER 10. CHAPTER I

-- sbspc:=sbSpcOfCurve(d,laCrb)
-- mpsol:=quotVecSpaceBasis(psol,sbspc)$LinesOpPack(K)
--
-- empty?(mpsol) => [0]
--
-- rowEchmpsol:=rowEchelon(matrix(mpsol))
-- npsol:=listOfLists(rowEchmpsol)
-- [reduce("+",[a*f for a in ll for f in lm]) for ll in npsol]

listVar:List(OV):= [index(i::PositiveInteger)$OV for i in 1..#symb]

listMonoPols:List(PolyRing):=[monomial(1,vv,1) for vv in listVar]

crtV(lcoef,lpos,1)==
vvv:List(K):=[0 for i in 1..l]
for c in lcoef for p in lpos repeat
   setelt(vvv,p,c)
vvv

sbSpcOfCurve(m,laCrb)==
d:=totalDegree(laCrb)$PackPoly
lm:List(PolyRing):=listAllMono(m)$PackPoly
m<d => [[0$K for i in 1..#lm]]
sd:NNI:=((m pretend INT)-(d pretend INT)) pretend NNI
slm:List(PolyRing):=listAllMono(sd)$PackPoly
allPol:=[laCrb*f for f in slm]
lpos:=[[position(m,lm) for m in primitiveMonomials(f)] for f in allPol]
lcoef:=[coefficients(f) for f in allPol]
clm:=#lm
[crtV(lc,lp,clm) for lc in lcoef for lp in lpos]

inVecSpace?: (List(K),List(List(K))) -> Boolean
inVecSpace?(line,basis)==
mat:Matrix(K):=matrix(basis)
rmat:=rank(mat)
augmat:Matrix(K):=matrix(concat(line,basis))
raugmat:=rank(augmat)
rmat=raugmat

exponent2monomial(lexp)==
reduce("*",[m**e for m in listMonoPols for e in lexp])

-- interpolateFunctions(lstOfPlc,lstOfv,lmnumer)==
---- lstOfPlc:= supp divis
---- lstOfv:=[coef(divis,pl) for pl in lstOfPlc]
--
-- lpls:List(List(PCS))
-- lplsT:List(List(PCS))
-- llll:List(List(Integer))
-- 1Ord:List(Integer)
-- ordMin:Integer
-- ppsol:List(Vector(K))
-- linSys:Matrix(K)
-- if ~empty?(lstOfPlc) then
-- lplsT:=[[parametrize(f,pl)$ParamPack for f in lmnumer ]
--   for pl in lstOfPlc]
-- lplsT:=[[removeFirstZeroes(s) for s in l] for l in lplsT]
--
-- -- series must be shift if some of them has negative order
-- llll:= [[order(s)$PCS for s in l] for l in lplsT]
-- 1Ord:= concat llll
-- 1Ord:=cons(0,1Ord)
-- ordMin:Integer:= "min"/1Ord
-- lplsT:=[[shift(s,-ordMin) for s in l] for l in lplsT]
--
-- lpls:=[[filterUpTo(s,v-ordMin) for s in souslplsT ]
--   for souslplsT in lplsT for v in lstOfv]
-- linSys:=reduce("vertConcat",_
--   [finiteSeries2LinSys(souslplsT,v-ordMin)$LINPACK 
--     for souslplsT in lpls for v in lstOfv])
--
-- ppsol contient la base des formes passant par le diviseur divv
-- ppsol:=nullSpace(linSys)
-- else
-- zeroMat:Matrix(K):=zero(1,#lmnumer)$Matrix(K)
-- ppsol:=nullSpace zeroMat
-- mpsol:=psol:List(List(K)):= [entries(vec) for vec in ppsol]
-- -- inserer ici le code pour calculer la base modulo l'ideal ...
-- empty?(mpsol) => [0]
-- rowEchmpsol:=rowEchelon(matrix(mpsol))
-- npsol:=list0flsLists(rowEchmpsol)
-- [reduce("+",[a*f for a in ll for f in lmnumer]) for ll in npsol]
package INTDIVP IntersectionDivisorPackage

--- IntersectionDivisorPackage.input ---

)set break resume
)sys rm -f IntersectionDivisorPackage.output
)spool IntersectionDivisorPackage.output
)set message test on
)set message auto off
)clear all

--) 1 of 1
)show IntersectionDivisorPackage
--)R
)--R IntersectionDivisorPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)),E: ... DesingTreeCategory(InfClsPoint),BLMET: BlowUpMethodCategory) is a package constructor
)--R Abbreviation for IntersectionDivisorPackage is INTDIVP
)--R This constructor is exposed in this frame.
)--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTDIVP
--)R
)--R-------------------------------------------------- Operations -----------------------------
)--R intersectionDivisor : (PolyRing,PolyRing,List(DesTree),List(ProjPt)) -> DIVISOR
)--R placesOfDegree : (PositiveInteger,PolyRing,List(ProjPt)) -> Void
)--R
)--E 1

)spool
)lisp (bye)

---

--- IntersectionDivisorPackage.help ---

====================================================================
IntersectionDivisorPackage examples
====================================================================

The following is part of the PAFF package

See Also:
  o )show IntersectionDivisorPackage

---
IntersectionDivisorPackage (INTDIVP)

Exports:
intersectionDivisor placesOfDegree

-- package INTDIVP IntersectionDivisorPackage --

)abbrev package INTDIVP IntersectionDivisorPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
IntersectionDivisorPackage(K,symb,PolyRing,E,ProjPt, PCS,Plc,DIVISOR,_
InfClsPoint,DesTree,BLMET):

Exports == Implementation where

K:Field
symb: List(Symbol)

OV ==&gt; OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)

PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)
DIVISOR : DivisorCategory(Plc)
InfClsPoint : InfinitelyClosePointCategory(K,symb,PolyRing,E,ProjPt,_
PCS,Plc,DIVISOR,BLMET)

DesTree : DesingTreeCategory(InfClsPoint)
BLMET : BlowUpMethodCategory

OF ==&gt; OutputForm
PackPoly ==&gt; PackageForPoly(K,PolyRing,E,#symb)
PPPFC1 ==&gt; PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
ParamPackFC ==&gt; LocalParametrizationOfSimplePointPackage(K,symb,PolyRing,_
\begin{verbatim}
E,ProjPt,PCS,Plc)

ParamPack  ==>  ParametrizationPackage(K,symb,PolyRing,E,ProjPt,PCS,Plc)
RatSingPack  ==>  ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt)
DesingPack  ==>  DesingTreePackage(K,symb,PolyRing,E,ProjPt,PCS,Plc,_,
             DIVISOR,InfClsPoint,DesTree,BLMET)

Exports  ==>  with

  intersectionDivisor:(PolyRing,PolyRing,List DesTree,List ProjPt) -> DIVISOR
  ++  intersectionDivisor(f,pol,listOfTree) returns the intersection
  ++  divisor of f with a curve defined by pol. listOfTree must contain
  ++  all the desingularisation trees of all singular points on the curve
  ++  defined by pol.

  placesOfDegree: (PositiveInteger, PolyRing, List ProjPt) -> Void()
  ++  placesOfDegree(d, f, pts) compute the places of degree
  ++  dividing d of the curve f. pts should be the singular points
  ++  of the curve f. For d > 1 this only works if K has
  ++  \texttt{axiomType(PseudoAlgebraicClosureOfFiniteFieldCategory)}.

Implementation  ==>  add

  intersectionDivisor(pol,curve,ltr,listOfSingPt)==
    intDeg:Integer:= (totalDegree(pol)$PackPoly * 
           totalDegree(curve)$PackPoly) pretend Integer
    -- compute at places over singular Points
    lDivAtSingPt:DIVISOR:=
      reduce("+",[divisorAtDesingTree(pol,tr)$DesingPack for tr in ltr],0)
    -- By Bezout Thorem, if all intersection points with mult.
    -- have been found then return the divisor
    degD:Integer:=degree lDivAtSingPt
    degD = intDeg => lDivAtSingPt
    setOfFdPlc:List Plc:=foundPlaces()$Plc
    plcFrSplPts:List Plc:=[pl for pl in setOfFdPlc | ^leaf?(pl)]
    ordAtPlcFrSplPts:List Integer:=
      [order(parametrize(pol,pl)$ParamPack)$PCS for pl in plcFrSplPts]
    divAtSplPts:DIVISOR:=
      reduce("+",[o * (pl :: DIVISOR) _
                  for o in ordAtPlcFrSplPts _
                  for pl in plcFrSplPts],0)
    tDiv:=lDivAtSingPt+divAtSplPts
    -- By Bezout Thorem, if all intersection points with mult.
    -- have been found then return the divisor
    degD:Integer:=degree tDiv
    degD = intDeg => tDiv
    intPts:List ProjPt:=algebraicSet([pol,curve])$RatSingPack
    intPtsNotSing:=setDifference(intPts,listOfSingPt)
    intPls:List(Plc):=_
      [pointToPlace(pt,curve)$ParamPackFC for pt in intPtsNotSing]
    remPlc:=setDifference(intPls , plcFrSplPts)
\end{verbatim}
ordAtPlcRem: List Integer :=
    [order(parametrize(pol, pl)$ParamPack$PCS for pl in remPlc]
 divAtRem: DIVISOR :=
    reduce("+",[o*(pl :: DIVISOR) for o in ordAtPlcRem for pl in remPlc], 0)
theDivisor := 1DivAtSingPt + divAtSplPts + divAtRem
degD: Integer := degree theDivisor
if ~(degD = intDeg) then
    print("error while computing the intersection divisor" :: OF )
print("Otherwise the Bezout Theoreme is not true !!!!! " :: OF)
print("Of course its the machine that make the mistake !!!!!" :: OF)
theDivisor

placesOfDegree(d, curve, singPts) ==
    --Return the number of places of degree i of the functionfield, no
    --constant field extension
    allPoints: List ProjPt := rationalPoints(curve, d)$RatSingPack
    remindingSimplePts: List ProjPt := setDifference(allPoints, singPts)
    for tpt in remindingSimplePts repeat
        pointToPlace(tpt, curve)$ParamPackFC
        Void()
IrredPolyOverFiniteField (IRREDFFX)

Exports:
  generateIrredPoly
package IRREDFFX IrredPolyOverFiniteField

)abbrev package IRREDFFX IrredPolyOverFiniteField
++ Author: Robert S. Sutor (original)
++ Date Last Updated: 29 May 1990
++ Description:
++ This package exports the function generateIrredPoly that computes
++ a monic irreducible polynomial of degree n over a finite field.

IrredPolyOverFiniteField(GF:FiniteFieldCategory): Exports == Impl where
N ==> PositiveInteger
Z ==> Integer
SUP ==> SparseUnivariatePolynomial GF
QR ==> Record(quotient: Z, remainder: Z)

Exports ==> with
generateIrredPoly: N -> SUP
++ generateIrredPoly(n) generates an irreducible univariate
++ polynomial of the given degree n over the finite field.

Impl ==> add
import DistinctDegreeFactorize(GF, SUP)

getIrredPoly : (Z, N) -> SUP
qAdicExpansion: Z -> SUP

p := characteristic()$GF :: N
q := size()$GF :: N

qAdicExpansion(z : Z): SUP ==
-- expands z as a sum of powers of q, with coefficients in GF
-- z = HornerEval(qAdicExpansion z,q)
qr := divide(z, q)
zero?(qr.remainder) => monomial(1, 1) * qAdicExpansion(qr.quotient)
r := index(qr.remainder pretend N)$GF :: SUP
zero?(qr.quotient) => r
r + monomial(1, 1) * qAdicExpansion(qr.quotient)

getIrredPoly(start : Z, n : N) : SUP ==
-- idea is to iterate over possibly irreducible monic polynomials
-- until we find an irreducible one. The obviously reducible ones
-- are avoided.
mon := monomial(1, 1)$SUP
pol: SUP := 0
found: Boolean := false
end: Z := q**n - 1
while not ((end < start) or found) repeat
if gcd(start, p) = 1 then
if irreducible?(pol := mon + qAdicExpansion(start)) then
found := true
start := start + 1
zero? pol => error "no irreducible poly found"
pol

generateIrredPoly(n : N) : SUP ==

-- want same poly every time
-- one?(n) => monomial(1, 1)$SUP
(n = 1) => monomial(1, 1)$SUP
-- one?(gcd(p, n)) or (n < q) =>
(gcd(p, n) = 1) or (n < q) =>
odd?(n) => getIrredPoly(2, n)
getIrredPoly(1, n)
getIrredPoly(q + 1, n)

package IRSN IrrRepSymNatPackage

-- IrrRepSymNatPackage.input --

)set break resume
)sys rm -f IrrRepSymNatPackage.output
)spool IrrRepSymNatPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show IrrRepSymNatPackage

--R
--R IrrRepSymNatPackage is a package constructor
--R Abbreviation for IrrRepSymNatPackage is IRSN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for IRSN
--R
IrrRepSymNatPackage contains functions for computing the ordinary irreducible representations of symmetric groups on n letters \{1,2,...,n\} in Young's natural form and their dimensions.

These representations can be labelled by number partitions of n, i.e. a weakly decreasing sequence of integers summing up to n, e.g. \[3,3,3,1\] labels an irreducible representation for n equals 10.

Note that whenever a \spadtype{List Integer} appears in a signature, a partition required.

See Also:
- )show IrrRepSymNatPackage

---
IrrRepSymNatPackage (IRSN)

Exports:

dimensionOfIrreducibleRepresentation irreducibleRepresentation

package IRSN IrrRepSymNatPackage

IrrRepSymNatPackage(): public == private where

NNI ==> NonNegativeInteger
I ==> Integer
L ==> List

Authors: Johannes Grabmeier, Thorsten Werther
Date Created: 04 August 1988
Date Last Updated: 24 May 1991
References:
G. James, A. Kerber: The Representation Theory of the Symmetric
J. Grabmeier, A. Kerber: The Evaluation of Irreducible
Polynomial Representations of the General Linear Groups
and of the Unitary Groups over Fields of Characteristic 0,
H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
their Realization in the Computer Algebra System Scratchpad,
Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23
Description:
IrrRepSymNatPackage contains functions for computing
the ordinary irreducible representations of symmetric groups on
n letters {1,2,...,n} in Young’s natural form and their dimensions.
These representations can be labelled by number partitions of n,
i.e. a weakly decreasing sequence of integers summing up to n, e.g.
Note that whenever a \spadtype{List Integer} appears in a signature,
a partition required.
-- NOT TRUE in current system, but should:
-- also could be an element of \spadtype{Partition}
\textbf{PACKAGE IRSN IRRREPSYMNATPACKAGE}

\texttt{M} \implies \text{Matrix} \\
\texttt{V} \implies \text{Vector} \\
\texttt{B} \implies \text{Boolean} \\
\texttt{SGCF} \implies \text{SymmetricGroupCombinatoricFunctions} \\
\texttt{ICF} \implies \text{IntegerCombinatoricFunctions Integer} \\
\texttt{PP} \implies \text{PartitionsAndPermutations} \\
\texttt{PERM} \implies \text{Permutation} \\

\texttt{public} \implies \text{with}

\texttt{dimensionOfIrreducibleRepresentation} : \texttt{L I} \to \texttt{NNI} \\
\texttt{irreducibleRepresentation} : \texttt{(L I, PERM I)} \to \texttt{M I} \\
\texttt{irreducibleRepresentation} : \texttt{(L I, L PERM I)} \to \texttt{L M I} \\

\texttt{private} \implies \text{add}

\texttt{oldlambda} : \texttt{L I} := \text{nil}\$(\texttt{L I}) \\
\texttt{flambda} : \texttt{NNI} := 0 \quad \text{-- dimension of the irreducible repr.} \\
\texttt{younglist} : \texttt{L M I} := \text{nil}\$(\texttt{L M I}) \quad \text{-- list of all standard tableaux} \\
\texttt{lprime} : \texttt{L I} := \text{nil}\$(\texttt{L I}) \quad \text{-- conjugated partition of lambda} \\
\texttt{n} : \texttt{NNI} := \text{0} \quad \text{-- concerning symmetric group S\_n} \\
\texttt{rows} : \texttt{NNI} := \text{0} \quad \text{-- \# of rows of standard tableau} \\
\texttt{columns} : \texttt{NNI} := \text{0} \quad \text{-- \# of columns of standard tableau} \\
\texttt{aid} : \texttt{M I} := \text{new}(1,1,0) \\

\texttt{aIdInverse} : () \to \text{Void} \\
\text{\quad -- computes aId, the inverse of the matrix} \\
\text{\quad -- (signum(k,1,id))}\_1 <= k, l <= \text{flambda, where id}
-- denotes the identity permutation

alreadyComputed? : L I -> Void
-- test if the last calling of an exported function concerns
-- the same partition lambda as the previous call

listPermutation : PERM I -> L I -- should be in Permutation
-- converts a permutation pi into the list
-- [pi(1),pi(2),...,pi(n)]

-- if there exists a vertical permutation v of the tableau
-- tl := pi o younglist(l) (l-th standard tableau)
-- and a horizontal permutation h of the tableau
-- tk := younglist(k) (k-th standard tableau) such that
-- v o tl = h o tk,
-- then
-- signum(k,l,pi) = sign(v),
-- otherwise
-- signum(k,l,pi) = 0.

sumPartition : L I -> NNI
-- checks if lambda is a proper partition and results in
-- the sum of the entries

testPermutation : L I -> NNI
-- testPermutation(pi) checks if pi is an element of S_n,
-- the set of permutations of the set {1,2,...,n}.
-- If not, an error message will occur, if yes it replies n.

-- definition of local functions

aIdInverse() ==

    aId := new(flambda,flambda,0)
    for k in 1..flambda repeat
        aId(k,k) := 1
    if n < 5 then return aId

    idperm : L I := nil$(L I)
    for k in n..1 by -1 repeat
        idperm := cons(k,idperm)
    for k in 1..(flambda-1) repeat
        for l in (k+1)..flambda repeat
            aId(k::NNI,l::NNI) := signum(k::NNI,l::NNI,idperm)

    -- invert the upper triangular matrix aId
    for j in flambda..2 by -1 repeat
for i in (j-1)..<1 by -1 repeat
  aid(i::NNI,j::NNI) := -aid(i::NNI,j::NNI)
for k in (j+1)..<lambda repeat
  for i in (j-1)..<1 by -1 repeat
    aid(i::NNI,k::NNI) := aid(i::NNI,k::NNI) + aid(i::NNI,j::NNI) * aid(j::NNI,k::NNI)

alreadyComputed?(lambda) ==
  if not(lambda = oldlambda) then
    oldlambda := lambda
    lprime := conjugate(lambda)$PP
    rows := (first(lprime)$(LI))::NNI
    columns := (first(lambda)$(LI))::NNI
    n := (+/lambda)::NNI
    younglist := listYoungTableaus(lambda)$SGCF
    flambda := #younglist
    aidInverse() -- side effect: creates actual aid

listPermutation(pi) ==
  li : LI := nil$(LI)
  for k in n..<1 by -1 repeat
    li := cons(eval(pi,k)$(PERM I),li)
  li

signum(numberOfRowTableau, numberOfColumnTableau,pi) ==
  rowtab : MI := copy younglist numberOfRowTableau
  columntab : MI := copy younglist numberOfColumnTableau
  swap : I
  sign : I := 1
  end : B := false
  endk : B
  ctrl : B

  -- k-loop for all rows of tableau rowtab
  k : NNI := 1
  while (k <= rows) and (not end) repeat
    -- l-loop along the k-th row of rowtab
    l : NNI := 1
    while (l <= oldlambda(k)) and (not end) repeat
      z : NNI := l
      endk := false
      -- z-loop for k-th row of rowtab beginning at column 1.
      -- test wether the entry rowtab(k,z) occurs in the l-th column
      -- beginning at row k of pi o columntab
      while (z <= oldlambda(k)) and (not endk) repeat
        s : NNI := k
        ctrl := true
        while ctrl repeat
if (s <= lprime(l))
    then
        if (1+rowtab(k,z) = pi(1+columntab(s,l)))
            -- if entries in the tableaus were from 1,...,n, then
            -- it should be .columntab(s,l)...
            then ctrl := false
            else s := s + 1
        else ctrl := false
        -- end of ctrl-loop
    endk := (s <= lprime(l)) -- same entry found ?
    if not endk
        then -- try next entry
            z := z + 1
        else
            if k < s
                then -- vertical permutation
                    sign := -sign
                    swap := columntab(s,l)
                    columntab(s,l) := columntab(k,l)
                    columntab(k,l) := swap
                if l < z
                    then -- horizontal permutation
                        swap := rowtab(k,z)
                        rowtab(k,z) := rowtab(k,l)
                        rowtab(k,l) := swap
                    -- end of else
                -- end of z-loop
            if (z > oldlambda(k)) -- no corresponding entry found
                then
                    sign := 0
                    end := true
                    l := l + 1
                -- end of l-loop
            k := k + 1
            -- end of k-loop
        sign

sumPartition(lambda) ==
    ok : B := true
    prev : I := first lambda
    sum : I := 0
    for x in lambda repeat
        sum := sum + x
        ok := ok and (prev >= x)
        prev := x
    if not ok then
        error("No proper partition ")
    sum::NNI
testPermutation(pi : L I) : NNI ==
ok : B := true
n : I := 0
for i in pi repeat
  if i > n then n := i -- find the largest entry n in pi
  if i < 1 then ok := false -- check whether there are entries < 1
  -- now n should be the number of permuted objects
  if (not (n=#pi)) or (not ok) then
    error("No permutation of 1,2,...,n")
  -- now we know that pi has n Elements ranging from 1 to n
  test : Vector(B) := new((n)::NNI,false)
  for i in pi repeat
    test(i) := true -- this means that i occurs in pi
  if member?(false,test) then error("No permutation") -- pi not surjective
n::NNI

-- definitions of exported functions

dimensionOfIrreducibleRepresentation(lambda) ==
nn : I := sumPartition(lambda)::I -- also checks whether lambda
  -- is a partition
  dd : I := 1
  -- run through all rows of the Youngtableau corr. to lambda
  lambdaprime : L I := conjugate(lambda)$PP
  for i in 1..lambdaprime.1 repeat
    -- run through all nodes in row i of the Youngtableau
    for j in 1..lambda.i repeat
      -- the hooklength of node (i,j) of the Youngtableau
      dd := dd * (lambda.i + lambdaprime.j - i - j + 1)
    (factorial(nn)$ICF quo dd)::NNI

irreducibleRepresentation(lambda:(L I),pi:(PERM I)) ==
nn : NNI := sumPartition(lambda)
  -- alreadyComputed?(lambda)
  piList : L I := listPermutation pi
  if not (nn = testPermutation(piList)) then
    error("Partition and permutation are not consistent")
aPi : M I := new(flamba,flamba,0)
  for k in 1..flamba repeat
    for l in 1..flamba repeat
      aPi(k,l) := signum(k,l,piList)
aId * aPi

irreducibleRepresentation(lambda) ==
listperm : L PERM I := nil$(L PERM I)
li : L I := nil$(L I)
sumPartition(lambda)
alreadyComputed?(lambda)
listperm := 
n = 1 => cons(1$(PERM I),listperm)
n = 2 => cons(cycle([1,2])$(PERM I),listperm)
-- the n-cycle (1,2,...,n) and the 2-cycle (1,2) generate S_n
for k in n..1 by -1 repeat
li := cons(k,li) -- becomes n-cycle (1,2,...,n)
listperm := cons(cycle(li)$(PERM I),listperm)
-- 2-cycle (1,2)
cons(cycle([1,2])$(PERM I),listperm)
irreducibleRepresentation(lambda,listperm)

irreducibleRepresentation(lambda:(L I),listperm:(L PERM I)) == 
sumPartition(lambda)
alreadyComputed?(lambda)
[irreducibleRepresentation(lambda, pi) for pi in listperm]

— IRSN.dotabb —

"IRSN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=IRSN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"IRSN" -> "IVECTOR"

— package INVLAPLA InverseLaplaceTransform —

)set break resume
/sys rm -f InverseLaplaceTransform.output
/spool InverseLaplaceTransform.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show InverseLaplaceTransform
InverseLaplaceTransform (INVLAPLA)

Exports:

inverseLaplace
--- package INVLAPLA InverseLaplaceTransform ---

)abbrev package INVLAPLA InverseLaplaceTransform
++ Author: Barry Trager
++ Date Created: 3 Sept 1991
++ Date Last Updated: 3 Sept 1991
++ Description:
++ This package computes the inverse Laplace Transform.

InverseLaplaceTransform(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
         RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
          SpecialFunctionCategory, AlgebraicallyClosedFunctionSpace R)
  SE ==> Symbol
  PI ==> PositiveInteger
  N ==> NonNegativeInteger
  K ==> Kernel F
  UP ==> SparseUnivariatePolynomial F
  RF ==> Fraction UP

Exports ==> with
  inverseLaplace: (F, SE, SE) -> Union(F,"failed")
  ++ inverseLaplace(f, s, t) returns the Inverse
  ++ Laplace transform of \spad{f(s)}
  ++ using t as the new variable or "failed" if unable to find
  ++ a closed form.
  ++ Handles only rational \spad{f(s)}.

Implementation ==> add

-- local ops --
ilt : (F,Symbol,Symbol) -> Union(F,"failed")
ilt1 : (RF,F) -> F
ilsqfr : (RF,F) -> F
iltirred: (UP,UP,F) -> F
freeOf?: (UP,Symbol) -> Boolean

inverseLaplace(expr,ivar,ovar) == ilt(expr,ivar,ovar)

freeOf?(p:UP,v:Symbol) ==
  "and"/[freeOf?(c,v) for c in coefficients p]

ilt(expr,var,t) ==
  expr = 0 => 0
  r := univariate(expr,kernel(var))

  -- Check that r is a rational function such that degree of
-- the numerator is lower than degree of denominator
not(\text{numerator}(r) \mod \text{denominator}(r) = 0) \Rightarrow "failed"
not(\text{freeOf?}(\text{numerator}(r,\text{var}) \text{and freeOf?}(\text{denominator}(r,\text{var})) \Rightarrow "failed"

ilt1(r,t::F)

\text{hintpac} := \text{TranscendentalHermiteIntegration}(F, \text{UP})

ilt1(r,t) ==
r = 0 \Rightarrow 0
\text{rsplit} := \text{HermiteIntegrate}\,(r, \text{differentiate})\,\text{hintpac}
-t\,\text{ilt1}\,(\text{rsplit}.\text{answer},t) + \text{iltsqfr}\,(\text{rsplit}.\text{logpart},t)

iltsqfr(r,t) ==
r = 0 \Rightarrow 0
p:=\text{numerator}\,r
q:=\text{denominator}\,r
-- \text{ql} := [\text{qq.factor} \text{for} \text{qq} \text{in factors} \text{factor} q]
\text{ql} := [\text{qq.factor} \text{for} \text{qq} \text{in factors} \text{squareFree} q]
# \text{ql} = 1 \Rightarrow \text{iltirred}\,(p,q,t)
\text{nl} := \text{multiEuclidean}(\text{ql},p)::\text{List}(\text{UP})
+/[\text{iltirred}\,(a,b,t) \text{for} a \text{in} \text{nl} \text{for} b \text{in} \text{ql}]

-- q is irreducible, monic, degree p < degree q
iltirred(p,q,t) ==
degree q = 1 \Rightarrow
\text{cp} := \text{coefficient}(p,0)
(c:=\text{coefficient}(q,0))=0 \Rightarrow \text{cp}
\text{cp} \times \exp(-c\,t)
degree q = 2 \Rightarrow
a := \text{coefficient}(p,1)
b := \text{coefficient}(p,0)
c:=(-1/2)\times\text{coefficient}(q,1)
d:= \text{coefficient}(q,0)
e := \exp(c\,t)
b := b+a*c
d := d-c**2
d > 0 \Rightarrow
\text{alpha} : F := \text{sqrt} \,d
\text{e} \times (a\times\cos(t*\text{alpha}) + b\times\sin(t*\text{alpha}))/\text{alpha}
\text{alpha} : F := \text{sqrt}(-d)
\text{e} \times (a\times\cosh(t*\text{alpha}) + b\times\sinh(t*\text{alpha}))/\text{alpha}
\text{roots} : \text{List} \, F := \text{zerosOf} \, q
q1 := \text{differentiate} \, q
+/[p(\text{root})/q1(\text{root}) \times \exp(\text{root}*t) \text{for} \text{root} \text{in} \text{roots}]

---
"INVLAPLA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INVLAPLA"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"INVLAPLA" -> "ACFS"
Chapter 11

Chapter J
Chapter 12

Chapter K

package KERNEL2 KernelFunctions2

— KernelFunctions2.input —

)set break resume
)sys rm -f KernelFunctions2.output
)spool KernelFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show KernelFunctions2
--R
--R KernelFunctions2(R: OrderedSet, S: OrderedSet) is a package constructor
--R Abbreviation for KernelFunctions2 is KERNEL2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for KERNEL2
--R
--R-------------------------------- Operations --------------------------------
--R constantKernel : R -> Kernel(S)
--R constantIfCan : Kernel(S) -> Union(R, "failed")
--R
--E 1

)spool
)lisp (bye)
--- KernelFunctions2.help ---

KernelFunctions2 examples

This package exports some auxiliary functions on kernels

See Also:
- )show KernelFunctions2

---

KernelFunctions2 (KERNEL2)

Exports:
constantKernel  constantIfCan

--- package KERNEL2 KernelFunctions2 ---

)abbrev package KERNEL2 KernelFunctions2
++ Description:
++ This package exports some auxiliary functions on kernels

KernelFunctions2(R:OrderedSet, S:OrderedSet): with
  constantKernel: R -> Kernel S
  ++ constantKernel(r) \undocumented
  constantIfCan : Kernel S -> Union(R, "failed")
  ++ constantIfCan(k) \undocumented

== add
import BasicOperatorFunctions1(R)
constantKernel r == kernel(constantOperator r, nil(), 1)
constantIfCan k == constantOpIfCan operator k


---
---

KERNEL2.dotabb

"KERNEL2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=KERNEL2"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"KERNEL2" -> "ORDSET"

package KOVACIC Kovacic

--- Kovacic.input ---

)set break resume
)sys rm -f Kovacic.output
)spool Kovacic.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show Kovacic
--R
--R Kovacic(F: Join(CharacteristicZero,AlgebraicallyClosedField,RetractableTo(Integer),RetractableTo(Fraction(Integer))),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for Kovacic is KUVACIC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for KUVACIC
--R
--R----------------------------------------- Operations -----------------------------------------
--R kovacic : (Fraction(UP),Fraction(UP),Fraction(UP)) -> Union(SparseUnivariatePolynomial(Fraction(UP)),"failed")
--R kovacic : (Fraction(UP),Fraction(UP),Fraction(UP),(UP -> Factored(UP))) -> Union(SparseUnivariatePolynomial(Fraction(UP)),"failed")
--R

)spool
)lisp (bye)

---
Kovacic provides a modified Kovacic’s algorithm for solving explicitely irreducible 2nd order linear ordinary differential equations.

See Also:
o )show Kovacic

---

Kovacic (KOVACIC)

Exports:
kovacic

--- package KOVACIC Kovacic ---

)abbrev package KOVACIC Kovacic
++ Author: Manuel Bronstein
++ Date Created: 14 January 1992
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{Kovacic} provides a modified Kovacic’s algorithm for
++ solving explicitely irreducible 2nd order linear ordinary
++ differential equations.

Kovacic(F, UP):Exports == Impl where
F : Join(CharacteristicZero, AlgebraicallyClosedField,
    RetractableTo Integer, RetractableTo Fraction Integer)
UP : UnivariatePolynomialCategory F
RF ==> Fraction UP
SUP ==> SparseUnivariatePolynomial RF
LF ==> List Record(factor:UP, exponent:Integer)
LODO==> LinearOrdinaryDifferentialOperator1 RF

Exports ==> with
kovacic: (RF, RF, RF) -> Union(SUP, "failed")
++ kovacic(a_0,a_1,a_2) returns either "failed" or P(u) such that
++ \spad{$e^\left(\frac{-a_1}{2a_2}\right) e^\int u$} is a solution of
++ \spad{$a_2 y'' + a_1 y' + a0 y = 0$}
++ whenever \spad{\text{factor}(P u = 0)}. 
++ The equation must be already irreducible over the rational functions.
kovacic: (RF, RF, RF, UP -> Factored UP) -> Union(SUP, "failed")
++ kovacic(a_0,a_1,a_2,ezfactor) returns either "failed" or P(u) such
++ that \spad{$e^\left(\frac{-a_1}{2a_2}\right) e^\int u$} is a solution of
++ \spad{$a_2 y'' + a_1 y' + a0 y = 0$}
++ whenever \spad{\text{factor}(P u = 0)}. 
++ The equation must be already irreducible over the rational functions.
++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
++ not necessarily into irreducibles.

Impl ==> add
import RationalRicDE(F, UP)

case2 : (RF, LF, UP -> Factored UP) -> Union(SUP, "failed")
cannotCase2?: LF -> Boolean

diff case2 : (RF, LF, UP -> Factored UP) -> Union(SUP, "failed")
kovacic(a0, a1, a2) ==

-- it is assumed here that a2 y'' + a1 y' + a0 y is already irreducible
-- over the rational functions, i.e. that the associated Riccati equation
-- does NOT have rational solutions (so we don't check case 1 of Kovacic's
-- algorithm)
-- currently only check case 2, not 3
kovacic(a0, a1, a2, ezfactor) ==

-- transform first the equation to the form y'' = r y
-- which makes the Galois group unimodular
-- this does not change irreducibility over the rational functions
-- the following is split into 5 lines in order to save a couple of
-- hours of compile time.
r:RF := a1**2
r := r + 2 * a2 * differentiate a1
r := r - 2 * a1 * differentiate a2
r := r - 4 * a0 * a2
r := r / (4 * a2**2)
lf := factors squareFree denom r
case2(r, lf, ezfactor)
-- this is case 2 of Kovacic's algorithm, i.e. look for a solution
-- of the associated Riccati equation in a quadratic extension
-- lf is the squarefree factorisation of denom(r) and is used to
-- check the necessary condition
case2(r, lf, ezfactor) ==
cannotCase2? lf => "failed"
-- build the symmetric square of the operator L = y'' - r y
-- which is L2 = y'''' - 4 r y' - 2 r' y
12:LODO := monomial(1, 3) - monomial(4*r, 1) - 2 * differentiate(r)::LODO
-- no solution in this case if L2 has no rational solution
empty?(sol := ricDsolve(l2, ezfactor)) => "failed"
-- otherwise the defining polynomial for an algebraic solution
-- of the Riccati equation associated with L is
-- u^2 - b u + (1/2 b' + 1/2 b^2 - r) = 0
-- where b is a rational solution of the Ricatti of L2
b := first sol
monomial(1, 2)$SUP - monomial(b, 1)$SUP
+ ((differentiate(b) + b**2 - 2 * r) / (2::RF))::SUP
-- checks the necessary condition for case 2
-- returns true if case 2 cannot have solutions
-- the necessary condition is that there is either a factor with
-- exponent 2 or odd exponent > 2
cannotCase2? lf ==
for rec in lf repeat
  rec.exponent = 2 or (odd?(rec.exponent) and rec.exponent > 2) =>
  return false
true

— KOVACIC.dotabb —

"KOVACIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=KOVACIC"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"KOVACIC" -> "ACF"
Chapter 13

Chapter L

package LAPLACE LaplaceTransform

— LaplaceTransform.input —

)set break resume
)sys rm -f LaplaceTransform.output
)spool LaplaceTransform.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LaplaceTransform
--R
--R LaplaceTransform(R: Join(EuclideanDomain,OrderedSet,CharacteristicZero,RetractableTo(Integer),LinearlyEvaluable)...
--R Abbreviation for LaplaceTransform is LAPLACE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LAPLACE
--R
--R----------------------------- Operations ---------------------------------
--R laplace : (F,Symbol,Symbol) -> F
--R
--E 1

)spool
)lisp (bye)


— LaplaceTransform.help —
LaplaceTransform examples
====================================================================
This package computes the forward Laplace Transform.
See Also:
- )show LaplaceTransform

LaplaceTransform (LAPLACE)

Exports:
laplace

--- package LAPLACE LaplaceTransform ---

LaplaceTransform(R, F): Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
  RetractableTo Integer, LinearlyExplicitRingOver Integer)
  F : Join(TranscendentalFunctionCategory, PrimitiveFunctionCategory,
  AlgebraicallyClosedFunctionSpace R)
  SE ==> Symbol
  PI ==> PositiveInteger
\[\text{N} \Rightarrow \text{NonNegativeInteger}\]
\[\text{K} \Rightarrow \text{Kernel F}\]
\[\text{OFE} \Rightarrow \text{OrderedCompletion F}\]
\[\text{EQ} \Rightarrow \text{Equation OFE}\]

\[\text{ALGOP} \Rightarrow "\%alg"\]
\[\text{SPECIALDIFF} \Rightarrow "\%specialDiff"\]

Exports \(\Rightarrow\) with
\[\text{laplace: (F, SE, SE)} \rightarrow F\]
\[\text{++ \text{laplace}(f, t, s) returns the Laplace transform of } \text{\texttt{spad}}(f(t))\]
\[\text{++ using } \text{\texttt{spad}}(s) \text{ as the new variable.}\]
\[\text{++ This is } \text{\texttt{integral}}(\exp(-s*t) \ast f(t), t = 0..\%plusInfinity).\]
\[\text{++ Returns the formal object } \text{\texttt{spad}}(\text{\texttt{laplace}(f, t, s)}) \text{ if it cannot}\]
\[\text{++ compute the transform.}\]

Implementation \(\Rightarrow\) add
\[\text{import IntegrationTools(R, F)}\]
\[\text{import ElementaryIntegration(R, F)}\]
\[\text{import PatternMatchIntegration(R, F)}\]
\[\text{import PowerSeriesLimitPackage(R, F)}\]
\[\text{import FunctionSpaceIntegration(R, F)}\]
\[\text{import TrigonometricManipulations(R, F)}\]

\[\text{locallaplace : (F, SE, F, SE, F)} \rightarrow F\]
\[\text{lapkernel : (F, SE, F, F)} \rightarrow \text{Union(F, "failed")}\]
\[\text{intlaplace : (F, F, F, SE, F)} \rightarrow \text{Union(F, "failed")}\]
\[\text{isLinear : (F, SE)} \rightarrow \text{Union(Record(const:F, nconst:F), "failed")}\]
\[\text{mkPlus} : F \rightarrow \text{Union(List F, "failed")}\]
\[\text{dvlap} : \text{List F, SE)} \rightarrow F\]
\[\text{tdenom} : (F, F)} \rightarrow \text{Union(F, "failed")}\]
\[\text{atn} : (F, SE)} \rightarrow \text{Union(Record(coef:F, deg:PI), "failed")}\]
\[\text{aexp} : (F, SE)} \rightarrow \text{Union(Record(coef:F, coefi:F, coef0:F), "failed")}\]
\[\text{algebraic?} : (F, SE)} \rightarrow \text{Boolean}\]

\[\text{oplap := operator("laplace"::Symbol, 3)$BasicOperator}\]
\[\text{laplace(f,t,s)} \Rightarrow \text{localaplace(complexElementary(f,t),t,t::F,s,s::F)}\]

\[\text{-- returns true if the highest kernel of f is algebraic over something}\]
\[\text{algebraic?(f, t)} \Rightarrow\]
\[\text{l := varselect(kernels f, t)}\]
\[\text{m:N := reduce(max, [height k for k in l], 0)$List(N)}\]
\[\text{for k in l repeat}\]
\[\text{height k = m and has?(operator k, ALGOP)} \Rightarrow \text{return true}\]
\[\text{false}\]

\[\text{-- differentiate a kernel of the form laplace(1.1,1.2,1.3) w.r.t x.}\]
\[\text{-- note that x is not necessarily 1.3}\]
\[\text{-- if x = 1.3, then there is no use recomputing the laplace transform,}\]
-- it will remain formal anyways

dvlap(l, x) ==
  l1 := first l
  l2 := second l
  x = (v := retract(l3 := third l)@SE) => - oplap(l2 * l1, l2, l3)
  e := exp(- 13 * l2)
  locallaplace(differentiate(e * l1, x) / e, retract(l2)@SE, l2, v, l3)

-- returns \([b, c]\) iff \(f = c * t + b\)
-- and \(b\) and \(c\) do not involve \(t\)

isLinear(f, t) ==
  ff := univariate(f, kernel(t)@K)
  ((d := retractIfCan(denom ff)@Union(F, "failed")) case "failed")
  or (degree(numer ff) > 1) => "failed"

  freeOf?(b := coefficient(numer ff, 0) / d, t) and
  freeOf?(c := coefficient(numer ff, 1) / d, t) => \([b, c]\)
  "failed"

-- returns \([a, n]\) iff \(f = a * t^n\)

atn(f, t) ==
  if ((v := isExpt f) case Record(var:K, exponent:Integer)) then
    w := v::Record(var:K, exponent:Integer)
    (w.exponent > 0) and
    ((vv := symbolIfCan(w.var)) case SE) and (vv::SE = t) =>
      return [1, w.exponent::PI]
  (u := isTimes f) case List(F) =>
    c:F := 1
    d:N := 0
    for g in u::List(F) repeat
      if (rec := atn(g, t)) case Record(coef:F, deg:PI) then
        r := rec::Record(coef:F, deg:PI)
        c := c * r.coef
        d := d + r.deg
      else c := c * g
      zero? d => "failed"
      [c, d::PI]
  "failed"

-- returns \([a, c, b]\) iff \(f = a * \exp(c * t + b)\)
-- and \(b\) and \(c\) do not involve \(t\)

aexp(f, t) ==
  is?(f, "exp"::SE) =>
    (v := isLinear(first argument(retract(f)@K), t)) case "failed" =>
      "failed"
    [1, v.nconst, v.const]
  (u := isTimes f) case List(F) =>
    c:F := 1
    c1 := c0 := 0$F
    for g in u::List(F) repeat
      if (r := aexp(g, t)) case Record(coef:F, coef1:F, coef0:F) then
rec := r::Record(coef:F, coef1:F, coef0:F)
c := c * rec.coef
c0 := c0 + rec.coef0
c1 := c1 + rec.coef1
else c := c * g
zero? c0 and zero? c1 => "failed"
[c, c1, c0]
if (v := isPower f) case Record(val:F, exponent:Integer) then
  w := v::Record(val:F, exponent:Integer)
  (w.exponent ^= 1) and
  ((r := aexp(w.val, t)) case Record(coef:F, coef1:F, coef0:F)) =>
  rec := r::Record(coef:F, coef1:F, coef0:F)
  return [rec.coef ** w.exponent, w.exponent * rec.coef1, w.exponent * rec.coef0]
"failed"

mkPlus f ==
(u := isPlus numer f) case "failed" => "failed"
d := denom f
[p / d for p in u::List(SparseMultivariatePolynomial(R, K))]
-- returns g if f = g/t
tdenom(f, t) ==
(denom f exquo numer t) case "failed" => "failed"
(t * f)

intlaplace(f, ss, g, v, vv) ==
  is?(g, oplap) or ((i := integrate(g, v)) case List(F)) => "failed"
  (u:=limit(i::F,equation(vv::OFE,plusInfinity()$OFE)$EQ)) case OFE =>
  (l := limit(i::F, equation(vv::OFE, ss::OFE)$EQ)) case OFE =>
  retractIfCan(u::OFE - l::OFE)@Union(F, "failed")
"failed"
"failed"

lapkernel(f, t, tt, ss) ==
  (k := retractIfCan(f)@Union(K, "failed")) case "failed" => "failed"
  empty?(arg := argument(k::K)) => "failed"
  is?(op := operator k, "%diff"::SE) =>
    not( #arg = 3) => "failed"
    not(is?(arg.3, t)) => "failed"
    fint := eval(arg.1, arg.2, tt)
    s := name operator (kernels(ss).1)
    ss * locallaplace(fint, t, tt, s, ss) - eval(fint, tt = 0)
  not (empty?(rest arg)) => "failed"
  member?(t, variables(a := first(arg) / tt)) => "failed"
  is?(op := operator k, "Si"::SE) => atan(a / ss) / ss
  is?(op, "Ci"::SE) => log((ss**2 + a**2) / a**2) / (2 * ss)
  is?(op, "Ei"::SE) => log((ss + a) / a) / ss
  -- digamma (or Gamma) needs SpecialFunctionCategory
  -- which we do not have here
-- is?(op, "log"::SE) => (digamma(1) - log(a) - log(ss)) / ss
"failed"

-- Below we try to apply one of the textbook rules for computing
-- Laplace transforms, either reducing problem to simpler cases
-- or using one of known base cases
locallaplace(f, t, tt, s, ss) ==
  zero? f => 0
  one? f => inv ss
  (f = 1) => inv ss

-- laplace(f(t)/t,t,s)
-- = integrate(laplace(f(t),t,v), v = s..%plusInfinity)
(x := tdenom(f, tt)) case F =>
  g := locallaplace(x::F, t, tt, vv := new()$SE, vvv := vv::F)
  (x := intlaplace(f, ss, g, vv, vvv)) case F => x::F
  oplap(f, tt, ss)

-- Use linearity
(u := mkPlus f) case List(F) =>
  +/[locallaplace(g, t, tt, s, ss) for g in u::List(F)]
(rec := splitConstant(f, t)).const ^= 1 =>
  rec.const * locallaplace(rec.nconst, t, tt, s, ss)

-- laplace(t^n*f(t),t,s) = (-1)^n*D(laplace(f(t),t,s), s, n))
(v := atn(f, t)) case Record(coef:F, deg:PI) =>
  vv := v::Record(coef:F, deg:PI)
  is?(la := locallaplace(vv.coef, t, tt, s, ss), oplap) => oplap(f,tt,ss)
  (-1$Integer)**(vv.deg) * differentiate(la, s, vv.deg)

-- Complex shift rule
(w := aexp(f, t)) case Record(coef:F, coef1:F, coef0:F) =>
  ww := w::Record(coef:F, coef1:F, coef0:F)
  exp(ww.coef0) * locallaplace(ww.coef,tt,ss,ss - ww.coef1)

-- Try base cases
(x := lapkernel(f, t, tt, ss)) case F => x::F

-- -- The following does not seem to help computing transforms, but
-- -- quite frequently leads to loops, so I (wh) disabled it for now
-- -- last chance option: try to use the fact that
-- -- laplace(f(t),t,s) = s laplace(g(t),t,s) - g(0) where dg/dt = f(t)
-- -- elem?(int := lfintegrate(f, t)) and (rint := retractIfCan int) case F =>
-- -- fint := rint :: F
-- -- to avoid infinite loops, we don't call laplace recursively
-- -- if the integral has no new logs and f is an algebraic function
-- -- empty?(logpart int) and algebraic?(f, t) => oplap(fint, tt, ss)
-- ss * locallaplace(fint, t, tt, ss) - eval(fint, tt = 0)
  oplap(f, tt, ss)
setProperty(clap, SPECIALDIFF, dvlap@((List F, SE) -> F) pretend None)

---

--- LAPLACE.dotabb ---

"LAPLACE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LAPLACE"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"LAPLACE" -> "ACFS"

---

package LAZM3PK LazardSetSolvingPackage

--- LazardSetSolvingPackage.input ---

)set break resume
)sys rm -f LazardSetSolvingPackage.output
)spool LazardSetSolvingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 37
R := Integer
--R
--R (1) Integer
--R
--E 1

--S 2 of 37
ls : List Symbol := [b1,x,y,z,t,v,u,w]
--R
--R (2) [b1,x,y,z,t,v,u,w]
--R
--E 2

--S 3 of 37
V := OVAR(ls)
--R
--R (3) OrderedVariableList([b1,x,y,z,t,v,u,w])
E := IndexedExponents V

P := NSMP(R, V)

b1: P := 'b1

x: P := 'x

y: P := 'y

z: P := 'z
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 9

---S 10 of 37
t: P := 't
---R
---R (10) t
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 10

---S 11 of 37
u: P := 'u
---R
---R (11) u
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 11

---S 12 of 37
v: P := 'v
---R
---R (12) v
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 12

---S 13 of 37
w: P := 'w
---R
---R (13) w
---IType: NewSparseMultivariatePolynomial(Integer,...
---E 13

---S 14 of 37
T := REGSET(R,E,V,P)
---R
---R
---R (14)
---R RegularTriangularSet(Integer,IndexedExponents(OrderedVariableList([b1,x,y,z,t
---R ,v,u,w]),OrderedVariableList([b1,x,y,z,t,v,u,w]),NewSparseMultivariatePolyno
---R mial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w])))
---R Type: Domain
---E 14

---S 15 of 37
p0 := b1 + y + z - t - w
---R
---R
p1 := 2*z*u + 2*y*v + 2*t*w - 2*w**2 - w - 1

p2 := 3*z*u**2 + 3*y*v**2 - 3*t*w**2 + 3*w**3 + 3*w**2 - t + 4*w

p3 := 6*x*z*v - 6*t*w**2 + 6*w**3 - 3*t*w + 6*w**2 - t + 4*w

p4 := 4*z*u**3+ 4*y*v**3+ 4*t*w**3- 4*w**4 - 6*w**3+ 4*t*w -10*w**2 - w - 1

p5 := 8*x*z*u*v +8*t*w**3 -8*w**4 +4*t*w**2 -12*w**3 +4*t*w -14*w**2 -3*w -1
**Package Lazm3pk LazardSetsSolvingPackage**

\[
p_6 := 12xzv^2 + 12tuv^3 - 12uv^4 + 12tuv^2 - 18uv^3 + 8tvw - 14uv^2 - w - 1
\]

\[
(21) \quad 12xz + (12w + 12w + 8w)t - 12w - 18w - 14w - w - 1
\]

--Type: NewSparseMultivariatePolynomial(Integer,...

--E 21

--S 22 of 37

\[
p_7 := -24tuv^3 + 24uv^4 - 24tuv^2 + 36uv^3 - 8tvw + 26uv^2 + 7vw + 1
\]

\[
(22) \quad (- 24w - 24w - 8w)t + 24w + 36w + 26w + 7w + 1
\]

--Type: NewSparseMultivariatePolynomial(Integer,...

--E 22

--S 23 of 37

\[
lp := [p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7]
\]

\[
(23) \quad [b_1 + y + z - t - w, 2v y + 2uw + 2w t - 2w - w - 1,
\]

\[
3v y + 3uw + (- 3w - 1)t + 3w + 3w + 4w,
\]

\[
6v z x + (- 6w - 3w - 1)t + 6w + 6w + 4w,
\]

\[
3v z x + (3w + 3w + 1)t + 3w + 3w + 4w,
\]

\[
4v y + 4uw + (4w + 4w)t - 4w - 6w - 10w - w - 1,
\]

\[
8uv z x + (8w + 4w + 4w)t - 8w - 12w - 14w - 3w - 1,
\]

\[
12v z x + (12w + 12w + 8w)t - 12w - 18w - 14w - w - 1,
\]

\[
(- 24w - 24w - 8w)t + 24w + 36w + 26w + 7w + 1
\]

--Type: List(NewSparseMultivariatePolynomial(Integer,...

--E 23

--S 24 of 37

\[
lts := \text{zeroSetSplit}(lp, false)\]

--R

\[
(24) \quad \{(w + 1, u, v, t + 1, b_1 + y + z + 2), \{w + 1, v, t + 1, z, b_1 + y + 2\},
\]

\[
\{w + 1, t + 1, z, y, b_1 + 2\}, \{w + 1, v - u, t + 1, y + z, x, b_1 + 2\},
\]

\[
\{w + 1, u, t + 1, y, x, b_1 + z + 2\},
\]

\[
5432
\]

\[
\{144w + 216w + 96w + 6w - 11w - 1,
\]

--R
(12w + 9w + 1)u - 72w - 108w - 9w - 3w,
(12w + 9w + 1)v + 36w + 54w + 18w,
(24w + 24w + 8w)t - 24w - 36w - 26w - 7w - 1,

(12u v - 12u )z + (12w v + 12w + 4)t + (3w - 5)v + 36w + 42w + 6w
+ 16w,

2v y + 2u z + 2w t - 2w - w - 1,

6v z x + (- 6w - 3w - 1)t + 6w + 6w + 4w, b1 + y + z - t - w}
exedExponents(OrderedVariableList([b1,x,y,z,t,v,u,w])),OrderedVariableList([b1,x,y,z,t,v,u,w]),NewSparseMultivariatePolynomial(Integer,OrderedVariableList([b1,x,y,z,t,v,u,w])))

Type: Domain

zeroSetSplit(lp,false)$pack

5 4 3 2 4 3 2
{144w + 216v + 96w + 6w - 11w - 1, u - 24w - 36w - 14w + w + 1,
  4 3 2
3v - 48w - 60w - 10w + 8w + 2, t - 24w - 36w - 14w - w + 1,
  4 3 2
486z - 2772w - 4662w - 2055w + 30w + 127,
  4 3 2
2916y - 22752w - 30312w - 8220w + 2064w + 1561,
  4 3 2
356x - 3696w - 4536w - 968w + 822w + 371,
  4 3 2
2916b1 - 30600w - 46692w - 20274w - 8076w + 593}

Type: List(SquareFreeRegularTriangularSet(Integer,...

f0 := (w - v) ** 2 + (u - t) ** 2 - 1

2 2 2 2
(t - 2u t + v - 2w v + u + w - 1

Type: NewSparseMultivariatePolynomial(Integer,...

f1 := t ** 2 - v ** 3

2 3
(t - v

Type: NewSparseMultivariatePolynomial(Integer,...
f2 := 2 * t * (w - v) + 3 * v ** 2 * (u - t)

--R
--R
--R 2
--R (31) (- 3v - 2v + 2w)t + 3u v
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 31

--S 32 of 37
f3 := (3 * z * v ** 2 - 1) * (2 * z * t - 1)

--R
--R
--R 2
--R (32) 6v t z + (- 2t - 3v )z + 1
--IType: NewSparseMultivariatePolynomial(Integer,...
--E 32

--S 33 of 37
lf := [f0, f1, f2, f3]

--R
--R
--R (33) 6v t z + (- 2t - 3v )z + 1
--IType: List(NewSparseMultivariatePolynomial(Integer,...
--E 33

--S 34 of 37
zeroSetSplit(lf,true)$T

--R
--R
--R (34) 6v t z + (- 2t - 3v )z + 1
--IType: List(NewSparseMultivariatePolynomial(Integer,...
--E 33
--R - 10125w - 4800w + 2501w + 4968w - 1587
--R *
--R v
--R +
--R 3 2 2 6 5 4 3 2
--R (1944w - 108w )u + 972w + 3024w - 1080w + 496w + 1116w
--R ,
--R 2 2 2 2 2
--R (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
--R ]
--IType: List(RegularTriangularSet(Integer, IndexedExponents(...
--E 34

--S 35 of 37
zeroSetSplit(lf,false)$T
--R
--R
--R (35)
--R [
--R {
--R 6 3 2 4
--R 729u + (- 1458w + 729w - 4158w - 1685)u
--R +
--R 6 5 4 3 2 2 8
--R (729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u + 729w
--R +
--R 7 6 5 4 3 2
--R 216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529
--R ,
--R
--R 4 3 2 2 6 5
--R 2187u + (- 4374w - 972w - 12474w - 2868)u + 2187w - 1944w
--R +
--R 4 3 2
--R - 10125w - 4800w + 2501w + 4968w - 1587
--R *
--R v
--R +
--R 3 2 2 6 5 4 3 2
--R (1944w - 108w )u + 972w + 3024w - 1080w + 496w + 1116w
--R ,
--R 2 2 2 2 2
--R (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
--R ,
--R 4 3 2
--R {27w + 4w - 54w - 36w + 23, u, (12w + 2)v - 9w - 2w + 9,
--R 2 2
--R 6t - 2v - 3w + 2w + 3, 2t z - 1}
--R ,
\[
\begin{align*}
\{ & 59049w + 91854w - 45198w + 145152w + 63549w + 60922w + 21420, \\
& 3148448266904w - 18316865522574w + 23676956746098w \\
+ & 6657857188965w + 8904703998546w + 3890631403260 \\
* & 2 \\
\} \\
\begin{align*}
& 2 \\
& u \\
& 5 \\
& 4 \\
& 3 \\
& 94262810316408w - 82887296576616w + 8980183148784w \\
& 2 \\
& 28141734167208w + 38070359425432w + 16003865949120 \\
& (243w + 36w + 85)v + (- 81u - 162w + 36w + 154w + 72)v - 72w + 4w, \\
& (3v + 2v - 2w)t - 3u v, ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1 \\
& 2 \\
& 6t - 2v - 3w + 2w + 3, 3v z - 1 \\
\end{align*}
\]
--E 35

--S 36 of 37
zeroSetSplit(lf,false)$pack
--R
--R (36)
--R [ 
--R 6 \\
--R 5 \\
--R 4 \\
--R 3 \\
--R 2 \\
--R 729u + (- 1458w + 729w - 4158w - 1685)u \\
--R + \\
--R 6 \\
--R 5 \\
--R 4 \\
--R 3 \\
--R 2 \\
--R 2 \\
--R 8 \\
--R (729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)v + 729w \\
--R + \\
--R 7 \\
--R 6 \\
--R 5 \\
--R 4 \\
--R 3 \\
--R 2 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 \\
--R 8 \\
--R 2 

\begin{verbatim}
--R 4 3 2 2 6 5
--R 2187u + (- 4374 w - 972 w - 12474 w - 2868)u + 2187w - 1944 w
--R +
--R 4 3 2
--R - 10125 w - 4800 w + 2501 w + 4968 w - 1587
--R *
--R +
--R 3 2 2 6 5 4 3 2
--R (1944 w - 108 w )u + 972 w + 3024 w - 1080 w + 496 w + 1116 w
--R ,
--R 2 2 2 2 2
--R (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
--R ,
--R 2
--R 641237934604288z
--R +
--R (78614584763904 w + 26785578742272)u + 236143618655616w
--R +
--R 70221988585728
--R *
--R v
--R +
--R (358520253138432 w + 101922133759488)u + 142598803536000w
--R +
--R 54166419595008
--R *
--R z
--R +
--R (32655103844499 w - 44224572465882)u v
--R +
--R (43213900115457 w - 32432039102070)u
--R }
--R ,
--R 4 3 2
--R {27w + 4w - 54w - 36w + 23, u, 218v - 162w + 3w + 160w + 153,
--R 2 3 2
--R 109t - 27w - 54w + 63w + 80, 1744z + (- 1458 w + 27w + 1440w + 505)t}
--R ,
--R 4 3 2
--R {27w + 4w - 54w - 36w + 23, u, 218v - 162w + 3w + 160w + 153,
--R 2 3 2
--R 109t - 27w - 54w + 63w + 80, 1308z + 162w - 3w - 814 w - 153}
\end{verbatim}
--R ,
--R --R 4 3 2 2 2
--R {729w + 972w - 1026w + 1684w + 765, 81u + 72w + 16w - 72,
--R 3 2
--R 702v - 162w - 225w + 40w - 99,
--R 3 2
--R 11336t + (324w - 603w - 1718w - 1557)u,
--R --R 2
--R 595003968z
--R +
--R 3 2
--R (-- 963325386w - 898607682w + 1516286466w - 3239166186)u
--R +
--R 3 2
--R - 1579048992w - 1796454288w + 2428328160w - 4368496024
--R *
--R z
--R +
--R 3 2
--R (971313306w + 9678670317w - 16726834476w + 28144233593)u
--R }
--R ]
--R --R Type: List(SquareFreeRegularTriangularSet(Integer,...
--E 36

--S 37 of 37
)show LazardSetSolvingPackage
--R
--R LazardSetSolvingPackage(R: GcdDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategoryType)
--R Abbreviation for LazardSetSolvingPackage is LAZM3PK
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LAZM3PK
--R
--R ------------------------------- Operations -------------------------------
--R normalizeIfCan : ST -> ST
--R zeroSetSplit : (List(P),Boolean) -> List(ST)
--R
--E 37
)spool
)lisp (bye)
A package for solving polynomial systems by means of Lazard triangular sets. This package provides two operations. One for solving in the sense of the regular zeros, and the other for solving in the sense of the Zariski closure. Both produce square-free regular sets. Moreover, the decompositions do not contain any redundant component. However, only zero-dimensional regular sets are normalized, since normalization may be time consuming in positive dimension.

The decomposition process is that of


The LazardSetSolvingPackage package constructor solves polynomial systems by means of Lazard triangular sets. However one condition is relaxed: Regular triangular sets whose saturated ideals have positive dimension are not necessarily normalized.

The decompositions are computed in two steps. First the algorithm of Moreno Maza (implemented in the RegularTriangularSet domain constructor) is called. Then the resulting decompositions are converted into lists of square-free regular triangular sets and the redundant components are removed. Moreover, zero-dimensional regular triangular sets are normalized.

Note that the way of understanding triangular decompositions is detailed in the example of the RegularTriangularSet constructor.

The LazardSetSolvingPackage constructor takes six arguments. The first one, R, is the coefficient ring of the polynomials; it must belong to the category GcdDomain. The second one, E, is the exponent monoid of the polynomials; it must belong to the category OrderedAbelianMonoidSup. The third one, V, is the ordered set of variables; it must belong to the category OrderedSet. The fourth one is the polynomial ring; it must belong to the category RecursivePolynomialCategory(R,E,V). The fifth one is a domain of the category RegularTriangularSetCategory(R,E,V,P) and the last one is a domain of the category SquareFreeRegularTriangularSetCategory(R,E,V,P). The abbreviation for LazardSetSolvingPackage is LAZM3PK.

For the purpose of solving zero-dimensional algebraic systems, see also LexTriangularPackage and ZeroDimensionalSolvePackage. These packages are easier to call than LAZM3PK. Moreover, the ZeroDimensionalSolvePackage package provides operations to compute either the complex roots or the real roots.

We illustrate now the use of the LazardSetSolvingPackage package constructor with two examples (Butcher and Vermeer).
Define the coefficient ring.

\[ R := \text{Integer} \]

Define the list of variables,

\[ \text{ls} : \text{List Symbol} := [b1, x, y, z, t, v, u, w] \]

and make it an ordered set:

\[ V := \text{OVAR(ls)} \]

then define the exponent monoid.

\[ E := \text{IndexedExponents V} \]

Define the polynomial ring.

\[ P := \text{NSMP}(R, V) \]

Let the variables be polynomial.

\[ b1: P := 'b1 \]
\[ x: P := 'x \]
\[ y: P := 'y \]
\[ z: P := 'z \]
\[ t: P := 't \]
\[ u: P := 'u \]
\[ v: P := 'v \]
\[ w: P := 'w \]

Now call the \{\tt RegularTriangularSet\} domain constructor.

\[ T := \text{REGSET}(R, E, V, P) \]

Define a polynomial system (the Butcher example).

\[ p0 := b1 + y + z - t - w \]
\[ p1 := 2z*u + 2y*v + 2t*w - 2*w**2 - w - 1 \]
\[
\begin{align*}
\p_2 & := 3z^2u^2 + 3y^2v^2 - 3t^2w^2 + 3w^3 + 3w^2 - t + 4w \\
\p_3 & := 6x^2zv - 6t^2w^2 + 6w^3 - 3t^2w + 6w^2 - t + 4w \\
\p_4 & := 4z^2u^3 + 4y^2v^3 + 4t^2w^3 - 4w^4 - 6w^3 + 4t^2w - 10w^2 - w - 1 \\
\p_5 & := 8x^2zuv + 8t^2w^3 - 8w^4 + 4t^2w^2 - 12t^2w + 7w^2 - 3w - 1 \\
\p_6 & := 12x^2zuv^2 + 12t^2w^3 - 12w^4 + 12t^2w^2 - 18w^3 + 4t^2w - 14w^2 - w - 1 \\
\p_7 & := -24t^2w^3 + 24w^4 - 24t^2w^2 + 36w^3 - 8t^2w + 26w^2 + 7w + 1 \\
\end{align*}
\]

\[
\lp := \{p_0, p_1, p_2, p_3, p_4, p_5, p_6, p_7\}
\]

First of all, let us solve this system in the sense of Lazard by means of the REGSET constructor:

\[
\begin{align*}
\text{lts} & := \text{zeroSetSplit}(\lp, \text{false})$
\end{align*}
\]
We can get the dimensions of each component of a decomposition as follows.

\[
\text{[coHeight(ts) for ts in lts]} \\
[3,3,3,2,2,0]
\]

The first five sets have a simple shape. However, the last one, which has dimension zero, can be simplified by using Lazard triangular sets.

Thus we call the \texttt{SquareFreeRegularTriangularSet} domain constructor,

\[
\text{ST := SREGSET(R,E,V,P)}
\]

and set the \texttt{LAZM3PK} package constructor to our situation,

\[
\text{pack := LAZM3PK(R,E,V,P,T,ST)}
\]

We are ready to solve the system by means of Lazard triangular sets:

\[
\text{zeroSetSplit(lp,false)$pack} \\
\{
w + 1,t + 1,z,y,b1 + 2\}, \{w + 1,v,t + 1,z,b1 + y + 2\}, \\
\{w + 1,u,v,t + 1,b1 + y + z + 2\}, \{w + 1,v - u,t + 1,y + z,x,b1 + 2\}, \\
\{w + 1,u,t + 1,y,x,b1 + z + 2\}, \\
\{144w + 216w + 96w + 6w - 11w - 1, u - 24w - 36w - 14w + w + 1, \\
486z - 2772w - 4662w - 2055w + 30w + 127, \\
2916y - 22752w - 30312w - 8220w + 2064w + 1561, \\
356x - 3696w - 4536w - 968w + 822w + 371, \\
2916b1 - 30600w - 46692w - 20274w - 8076w + 593\}
\]

We see the sixth triangular set is nicer now: each one of its polynomials has a constant initial.

We follow with the Vermeer example. The ordering is the usual one for this system.
Define the polynomial system.

\[ f_0 := (w - v)^2 + (u - t)^2 - 1 \]

\[ t - 2u t + v - 2w v + u + w - 1 \]

\[ f_1 := t^2 - v^3 \]

\[ t - v \]

\[ f_2 := 2 * t * (w - v) + 3 * v^2 * (u - t) \]

\[ (- 3v - 2v + 2w)t + 3u v \]

\[ f_3 := (3 * z * v^2 - 1) * (2 * z * t - 1) \]

\[ 6v t z + (- 2t - 3v)z + 1 \]

\[ l_f := [f_0, f_1, f_2, f_3] \]

First of all, let us solve this system in the sense of Kalkbrener by means of the REGSET constructor:

\[ \text{zeroSetSplit}(l_f, \text{true}) \]

\[
\begin{align*}
6 &
3 &
2 &
4 &
729u &+ (-1458w + 729w - 4158w - 1685)u + \\
6 &
5 &
4 &
3 &
2 &
8 &
(729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u + 729w + \\
7 &
6 &
5 &
4 &
3 &
2 &
216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529 &.
\end{align*}
\]

\[
4 &
3 &
2 &
2 &
6 &
5 &
2187u &+ (-4374w - 972w - 12474w - 2868)u + 2187w - 1944w +
\]

We have obtained one regular chain (i.e. regular triangular set) with dimension 1. This set is in fact a characteristic set of the (radical of) the ideal generated by the input system lf. Thus we have only the generic points of the variety associated with lf (for the elimination ordering given by ls).

So let us get now a full description of this variety.

Hence, we solve this system in the sense of Lazard by means of the REGSET constructor:

```plaintext
zeroSetSplit(lf,false)$T
```

\[
\begin{align*}
4 & 3 & 2 \\
-10125w & -4800w + 2501w + 4968w - 1587 \\
* & v \\
+ & 3 & 2 & 2 & 6 & 5 & 4 & 3 & 2 \\
(1944w - 108w )u & + 972w + 3024w - 1080w + 496w + 1116w \\
, & 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\
(3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1}
\end{align*}
\]

We retrieve our regular chain of dimension 1 and we get three regular chains of dimension 0 corresponding to the degenerated cases. We want now to simplify these zero-dimensional regular chains by using Lazard triangular sets. Moreover, this will allow us to prove that the above decomposition has no redundant component.

Generally, decompositions computed by the REGSET constructor do not have redundant components. However, to be sure that no redundant component occurs one needs to use the SREGSET or LAZM3PK constructors.

So let us solve the input system in the sense of Lazard by means of the LAZM3PK constructor:
zeroSetSplit(1f, false)$pack
[
  4
  3 2
  6 3 2 4
  729u + (-1458w + 729w - 4158w - 1685)u +
  6
  5 4 3 2
  8
  (729w - 1458w - 2619w - 4892w - 297w + 5814w + 427)u + 729w +
  7
  6 5 4 3 2
  216w - 2900w - 2376w + 3870w + 4072w - 1188w - 1656w + 529

  4
  3 2
  2187u + (-4374w - 972w - 12474w - 2868)u + 2187w - 1944w +
  4
  3 2
  - 10125w - 4800w + 2501w + 4968w - 1587

  3
  2 2 6 5
  (1944w - 108w )u + 972w + 3024w - 1080w + 496w + 1116w

  2
  2 2 2 2
  (3v + 2v - 2w)t - 3u v , ((4v - 4w)t - 6u v )z + (2t + 3v )z - 1

  2
  2 2 2
  (81w + 18w + 28, 729w - 1890w - 533, 81v + (-162w + 27)v - 72w - 112,
  11881t + (972w + 2997)u v + (-11440w - 11536)u,

  2
  641237934604288z

  +
  78614584763904w + 26785578742272)u + 236143618655616w

  +
  70221988585728

  *
  v

  +
  (358520253138432w + 1019221373759488)u + 142598803536000w

  +
  54166419595008

  *
  z

  +
  (32655103844499w - 44224572465882)u v

+
Due to square-free factorization, we obtained now four zero-dimensional regular chains. Moreover, each of them is normalized (the initials are constant). Note that these zero-dimensional components may be investigated further with the ZeroDimensionalSolvePackage package constructor.

See also:
- )show LazardSetSolvingPackage
- )show ZeroDimensionalSolvePackage
LazardSetSolvingPackage (LAZM3PK)

Exports:
  normalizeIfCan  zeroSetSplit

--- package LAZM3PK LazardSetSolvingPackage ---

)abbrev package LAZM3PK LazardSetSolvingPackage
++ Author: Marc Moreno Maza
++ Date Created: 10/02/1998
++ Date Last Updated: 12/16/1998
++ References:
  ++ [1] D. LAZARD "A new method for solving algebraic systems of
++ Description:
  ++ A package for solving polynomial systems by means of Lazard triangular
  ++ sets. This package provides two operations. One for solving in the sense
  ++ of the regular zeros, and the other for solving in the sense of
  ++ the Zariski closure. Both produce square-free regular sets.
  ++ Moreover, the decompositions do not contain any redundant component.
  ++ However, only zero-dimensional regular sets are normalized, since
  ++ normalization may be time consuming in positive dimension.
  ++ The decomposition process is that of [2].

LazardSetSolvingPackage(R,E,V,P,TS,ST): Exports == Implementation where

  R : GcdDomain
  E : OrderedAbelianMonoidSup
  V : OrderedSet
  P : RecursivePolynomialCategory(R,E,V)
TS: RegularTriangularSetCategory(R,E,V,P)
ST: SquareFreeRegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
K ==> Fraction R
LP ==> List P
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Split ==> List TS
--KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
--EntryGcd ==> List PWT
--HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
--KeyInvSet ==> Record(arg1: P, arg3: TS)
--EntryInvSet ==> List TS
--HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,ST)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,ST)
normalizpack ==> NormalizationPackage(R,E,V,P,ST)

Exports == with

  normalizeIfCan: ST -> ST
    ++ \axiom{normalizeIfCan(ts)} returns \axiom{ts} in an normalized shape
    ++ if \axiom{ts} is zero-dimensional.
  zeroSetSplit: (LP, B) -> List ST
    ++ \axiom{zeroSetSplit(lp,clos?)} has the same specifications as
    ++ zeroSetSplit(lp,clos?) from RegularTriangularSetCategory.

Implementation == add

  convert(st: ST): TS ==
    ts: TS := empty()
    lp: LP := members(st)$ST
    lp := sort(infRittWu?,lp)
    for p in lp repeat
      ts := internalAugment(p,ts)$TS
    ts

  squareFree(ts: TS): List ST ==
    empty? ts => [empty()$ST]
    lp: LP := members(ts)$TS
    lp := sort(infRittWu?,lp)
    newts: ST := empty()$ST
    toSee: List ST := [newts]
    toSave: List ST
    for p in lp repeat
toSave := []
while (not empty? toSee) repeat
  us := first toSee; toSee := rest toSee
  lpwt := stoseSquareFreePart(p,us)$regsetgcdpack
  for pwt in lpwt repeat
    newus := internalAugment(pwt.val,pwt.tower)$ST
    toSave := cons(newus,toSave)
  toSee := toSave

normalizeIfCan(ts: ST): ST ==
  empty? ts => ts
  lp: LP := members(ts)$ST
  lp := sort(infrittwu?,lp)
  p: P := first lp
  not univariate?(p)$polsetpack => ts
  lp := rest lp
  newts: ST := empty()$ST
  newts := internalAugment(p,newts)$ST
  while (not empty? lp) repeat
    p := first lp
    lv := variables(p)
    for v in lv repeat
      v = mvar(p) => "leave"
      not algebraic?(v,newts) => return internalAugment(lp,newts)$ST
    lp := rest lp
    p := normalizedAssociate(p,newts)$normalizpack
    newts := internalAugment(p,newts)$ST
  newts

zeroSetSplit(lp:List(P), clos?:B): List ST ==
  -- if clos? then SOLVE in the closure sense
  toSee: Split := zeroSetSplit(lp, clos?)$TS
  toSave: List ST := []
  for ts in toSee repeat
    toSave := concat(squareFree(ts),toSave)
  toSave := removeSuperfluousQuasiComponents(toSave)$quasicomppack
  [normalizeIfCan(ts) for ts in toSave]
PACKAGE LEADCDET LEADINGCOEFDETERMINATION

1443

package LEADCDET LeadingCoefDetermination
— LeadingCoefDetermination.input —
)set break resume
)sys rm -f LeadingCoefDetermination.output
)spool LeadingCoefDetermination.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LeadingCoefDetermination
--R
--R LeadingCoefDetermination(OV: OrderedSet,E: OrderedAbelianMonoidSup,Z: EuclideanDomain,P: PolynomialCateg
--R Abbreviation for LeadingCoefDetermination is LEADCDET
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LEADCDET
--R
--R------------------------------- Operations ---------------------------------R distFact : (Z,List(SparseUnivariatePolynomial(Z)),Record(contp: Z,factors: List(Record(irr: P,pow: Integ
--R polCase : (Z,NonNegativeInteger,List(Z)) -> Boolean
--R
--E 1
)spool
)lisp (bye)

———— LeadingCoefDetermination.help —
====================================================================
LeadingCoefDetermination examples
====================================================================
Package for leading coefficient determination in the lifting step.
Package working for every R euclidean with property "F".
See Also:
o )show LeadingCoefDetermination

———-


LeadingCoefDetermination (LEADCDET)

Exports:
distFact  polCase

— package LEADCDET LeadingCoefDetermination —

)abbrev package LEADCDET LeadingCoefDetermination
++ Author : P.Gianni, May 1990
++ Description:
++ Package for leading coefficient determination in the lifting step.
++ Package working for every R euclidean with property "F".

LeadingCoefDetermination(OV,E,Z,P) : C == T
where

OV  : OrderedSet
E   : OrderedAbelianMonoidSup
Z   : EuclideanDomain
BP  ==> SparseUnivariatePolynomial Z
P   : PolynomialCategory(Z,E,OV)
NNI ==> NonNegativeInteger
LeadFact ==> Record(polfac:List(P),correct:Z,corrfact:List(BP))
ParFact ==> Record(irr:P,pow:Integer)
FinalFact ==> Record(contp:Z,factors:List(ParFact))

C == with

polCase : (Z,NNI,List(Z)) -> Boolean
  ++ polCase(contprod, numFacts, evallcs), where contprod is the
  ++ product of the content of the leading coefficient of
  ++ the polynomial to be factored with the content of the
  ++ evaluated polynomial, numFacts is the number of factors
  ++ of the leadingCoefficient, and evallcs is the list of
  ++ the evaluated factors of the leadingCoefficient, returns
  ++ true if the factors of the leading Coefficient can be
  ++ distributed with this valuation.
distFact : (Z,List(BP),FinalFact,List(Z),List(OV),List(Z)) ->
  Union(LeadFact,"failed")
++ distFact(contm,unilist,plead,vl,lvar,lval), where contm is
++ the content of the evaluated polynomial, unilist is the list
++ of factors of the evaluated polynomial, plead is the complete
++ factorization of the leading coefficient, vl is the list
++ of factors of the leading coefficient evaluated, lvar is the
++ list of variables, lval is the list of values, returns a record
++ giving the list of leading coefficients to impose on the univariate
++ factors,

T == add
   distribute: (Z,List(BP),List(P),List(Z),List(OV),List(Z)) -> LeadFact
   checkpow : (Z,Z) -> NNI

polCase(d:Z,nk:NNI,lval:List(Z)):Boolean ==
   -- d is the product of the content lc m (case polynomial)
   -- and the cont of the polynomial evaluated
   q:Z
   distlist:List(Z) := [d]
   for i in 1..nk repeat
      q := unitNormal(lval.i).canonical
      for j in 0..(i-1)::NNI repeat
         y := distlist.((i-j)::NNI)
         while y^=1 repeat
            y := gcd(y,q)
            q := q quo y
         if q=1 then return false
      distlist := append(distlist,[q])
      true

checkpow(a:Z,b:Z) : NonNegativeInteger ==
   qt: Union(Z,"failed")
   for i in 0.. repeat
      qt:= b exquo a
      if qt case "failed" then return i
   b:=qt::Z
   distribute(contm:Z,unilist:List(BP),pl:List(P),vl:List(Z),
              lvar:List(OV),lval:List(Z)): LeadFact ==
   d,lcp : Z
   nf:NNI:=#unilist
   for i in 1..nf repeat
      lcp := leadingCoefficient (unilist.i)
      d:= gcd(lcp,vl.i)
      pl.i := (lcp quo d)*pl.i
      d := vl.i quo d
      unilist.i := d*unilist.i
      contm := contm quo d
      if contm ^=1 then for i in 1..nf repeat pl.i := contm*pl.i
   [pl,contm,unilist]$LeadFact
distFact(contm: Z, unilist: List(BP), plead: FinalFact, vl: List(Z), lvar: List(OV), lval: List(Z)): Union(LeadFact, "failed") ==

h: NonNegativeInteger

c, d : Z

lpol: List(P):=[[]
lexp: List(Integer):=[[]

nf: NNI := #unilist

vl := reverse vl -- lpol and vl reversed so test from right to left

for fpl in plead.factors repeat

lpol:=[fpl.irr,:lpol]
lexp:=[fpl.pow,:lexp]

vlp: List(Z):= [i$Z for i in 1..nf]
aux: List(P):= [i$P for i in 1..nf]

for i in 1..nf repeat

c := contm*leadingCoefficient unilist.i

c=1 or c=-1 => "next i"

for k in 1..(# lpol) repeat

lexp.k=0 => "next factor"

h:= checkpow(vl.k,c)

if h ^=0 then

if h>lexp.k then return "failed"

lexp.k:=lexp.k-h

aux.i := aux.i*(lpol.k ** h)

d:= vl.k**h

vlp.i:= vlp.i*d

c:= c quo d

if contm=1 then vlp.i:=c

for k in 1..(# lpol) repeat if lexp.k ^= 0 then return "failed"

contm =1 => [[vlp.i*aux.i for i in 1..nf], 1, unilist]$LeadFact
distribute(contm, unilist, aux, vlp, lvar, lval)

——

— LEADCDET.dotabb —

"LEADCDET" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LEADCDET"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LEADCDET" -> "PFECAT"

——

package LEXTRIPK LexTriangularPackage

— LexTriangularPackage.input —
set break resume
set message test on
set message auto off
clear all

-- S 1 of 23
R := Integer
-- R
-- R (1) Integer
-- R

-- E 1

-- S 2 of 23
ls : List Symbol := [a,b,c,d,e,f]
-- R
-- R
-- R (2) [a,b,c,d,e,f]
-- R

-- E 2

-- S 3 of 23
V := OVAR(ls)
-- R
-- R
-- R (3) OrderedVariableList([a,b,c,d,e,f])
-- R

-- E 3

-- S 4 of 23
P := NSMP(R, V)
-- R
-- R
-- R (4)
-- R NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
-- R

-- E 4

-- S 5 of 23
p1: P := a*b*c*d*e*f - 1
-- R
-- R
-- R (5) f e d c b a - 1
-- R Type: NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
-- E 5

-- S 6 of 23
p2: P := a*b*c*d*e +a*b*c*d*f +a*b*c*e*f +a*b*d*e*f +a*c*d*e*f +b*c*d*e*f
-- R
\[ ((e + f)d + f e)c + f e d c) a + f e d c b \]

---

```
---R
---R (6) (((e + f)d + f e)c + f e d c) a + f e d c b
---RTypem NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
---E 6
---S 7 of 23
```

```
p3: P := a*b*c*d + a*b*c*f + a*b*e*f + a*d*e*f + b*c*d*e + c*d*e*f
---R
---R (7) (((d + f)c + f e)b + f e d)a + e d c b + f e d c
---RTypem NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
---E 7
---S 8 of 23
```

```
p4: P := a*b*c + a*b*f + a*e*f + b*c*d + c*d*e + d*e*f
---R
---R (8) ((c + f)b + f e)a + d c b + e d c + f e d
---RTypem NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
---E 8
---S 9 of 23
```

```
p5: P := a*b + a*f + b*c + c*d + d*e + e*f
---R
---R (9) (b + f)a + c b + d c + e d + f e
---RTypem NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
---E 9
---S 10 of 23
```

```
p6: P := a + b + c + d + e + f
---R
---R (10) a + b + c + d + e + f
---RTypem NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f]))
---E 10
---S 11 of 23
```

```
lp := [p1, p2, p3, p4, p5, p6]
---R
---R (11)
---R [f e d c b a - 1, ((((e + f)d + f e)c + f e d b + f e d c)a + f e d c b,
---RTypem List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))
---E 11
---S 12 of 23
```
lextripack := LEXTRIPK(R,ls)
--R
--R
--R (12) LexTriangularPackage(Integer,[a,b,c,d,e,f])
--R Type: Domain
--E 12

--S 13 of 23
lg := groebner(lp)$lextripack
--R
--R
--R (13)
--R [a + b + c + d + e + f,
--R
--R 2
--R 3968379498283200b + 15873517993132800f b + 3968379498283200d
--R +
--R 3 5
--R 4 4
--R 15873517993132800 d + 3968379498283200 f e - 15873517993132800 f e
--R +
--R 5 3
--R 6 2
--R 23810276989699200 f e + (206355733910726400f e + 230166010900425600) e
--R +
--R 43
--R 37
--R - 729705987316687f + 1863667496667205421f
--R +
--R 31
--R 25
--R 291674853771731104461f + 365285994691106921745f
--R +
--R 19
--R 13
--R 549961185828911895f - 365048404038768439269f
--R +
--R 7
--R 292382820431504027669f - 2271689867631865497f
--R *
--R e
--R +
--R 44
--R 38
--R - 39888126426455399f + 10187423878429609997f
--R +
--R 32
--R 26
--R 1594377523424314053637f + 1994739308439916238065f
--R +
--R 20
--R 14
--R 1596840088052642815f - 1993494118301162145413f
--R +
--R 8
--R 2
--R - 1596049742289689815053f - 11488171330159667449f
--R ,
--R
\[ (23810276989699200c - 23810276989699200f)b + 23810276989699200c \]
\[ + \]
\[ 71430830969097600f c - 23810276989699200d - 95241107958796800f d \]
\[ + \]
\[ 3 5 \]
\[ 4 4 \]
\[ 5 3 \]
\[ - 55557312975964800f e + 174608697924460800f e - 174608697924460800f e \]
\[ + \]
\[ \]
\[ 6 \]
\[ 2 \]
\[ (- 2428648252949318400e - 2611193709870345600)e \]
\[ + \]
\[ 43 \]
\[ 37 \]
\[ 8305444561289527f - 212120871519454564f \]
\[ + \]
\[ 31 \]
\[ 25 \]
\[ - 331981583093451385381f - 4157691646261657136445f \]
\[ + \]
\[ 19 \]
\[ 13 \]
\[ - 6072721607510764095f + 4154986709036460221649f \]
\[ + \]
\[ 7 \]
\[ 3327761311138587096749f + 25885340608290841637f \]
\[ * \]
\[ e \]
\[ + \]
\[ 44 \]
\[ 38 \]
\[ 45815897629010329f - 117013765582151891207f \]
\[ + \]
\[ 32 \]
\[ 26 \]
\[ - 1831316684970865074187f - 22909971239649297438915f \]
\[ + \]
\[ 20 \]
\[ 14 \]
\[ - 16133250761305157265f + 2289730585763178256623f \]
\[ + \]
\[ 8 \]
\[ 2 \]
\[ 18329944781867242497923f + 130258531002020420699f \]
\[ , \]
\[ (7936758996566400d - 7936758996566400f)b - 7936758996566400df \]
\[ + \]
\[ 3 5 \]
\[ 4 4 \]
\[ 5 3 \]
\[ - 7936758996566400fe + 23810276989699200fe - 23810276989699200fe \]
\[ + \]
\[ 6 \]
\[ 2 \]
\[ (- 337312257354072000e - 36905929340337600e) \]
\[ + \]
\[ 43 \]
\[ 37 \]
\[ 1176345388640471f - 3004383582891473073f \]
\[ + \]
--R 31 25
--R 470203502707246105653f - 588858183402644348088f
--R +
--R 19 13
--R 856939308623513535f + 588472674223405263777f
--R +
--R 7
--R 471313241958371103517f + 3659742549078552381f
--R *
--R e
--R +
--R 44 38 32
--R 6423170513956901f - 164047721370368083f - 25674191652275877463f
--R +
--R 26 20
--R - 3211938090825682172335f - 2330490332697587485f
--R +
--R 14 8
--R 3210100109444754864587f + 2569858315395162617847f
--R +
--R 2
--R 18326089487427735751f
--R ,
--R 3 5
--R (11905138494849600e - 11905138494849600f)b - 3968379498283200f e
--R +
--R 4 4 5 3
--R 15873517993132800f e - 27778656487982400f e
--R +
--R 6 2
--R (- 208339923659868000f - 240086959646133600)e
--R +
--R 43 37
--R 786029984751110f - 2007519008182245250f
--R +
--R 31 25
--R - 314188062908073807090f - 393423667537929575250f
--R +
--R 19 13
--R - 550329120654394950f + 393196408728889612770f
--R +
--R 7
--R 314892372799176495730f + 240938651514668530f
--R *
--R e
--R +
--R 44 38 32
--R 4177638546747827f - 10669685294602576381f - 1669852980419949524601f
--R +
--R 26 20
--R - 2089077057287904170745f - 156989976358027879f
--R +
--R 14 8
--R 2087864026859015573349f + 1671496085945199677969f
--R +
--R 2
--R 11940257226216280177f
--R ,
--R 6
--R (11905138494849600f - 11905138494849600)b - 15873517993132800f e
--R +
--R 3 4 4 3
--R 3968379498283200f e - 3968379498283200f e
--R +
--R 11 5 2
--R (- 686529653202993600f - 607162063237329600f )e
--R +
--R 42 36 30
--R 65144531306704f - 166381280901088652f - 26033434502470283472f
--R +
--R 24 18
--R - 31696259583860650140f + 971492093167581360f
--R +
--R 12 6
--R 32220085033691389548f + 25526177666070529808f + 138603268355749244
--R *
--R +
--R 43 37 31
--R 167620036074811f - 428102417974791473f - 66997243801231679313f
--R +
--R 25 19
--R - 83426716722148750485f + 203673895369980765f
--R +
--R 13 7
--R 83523056326010432457f + 66996789640238066937f + 4785928655549587901f
--R ,
--R 3 2 2 45
--R 801692827936c + 2405078483808f c - 2405078483808f c - 13752945467f
--R +
--R 39 33 27
--R 35125117815561f + 5496946957826433f + 6834659447749117f
--R +
--R 21 15 9
--R - 44484880462461f - 6873406230093057f - 5450844938762633f
--R +
--R 3
(--R 1216586044571f
--R ,
--R 2
--R (23810276989699200d - 23810276989699200f)c + 23810276989699200d
--R +
--R 3
--R 5 4
--R 71430830969097600f d + 7936758996566400f e - 31747035986265600f e
--R +
--R 5
--R 3
--R 6
--R 2
--R 31747035986265600f e + (404774708824886400f + 396837949828320000)e
--R +
--R 43
--R 37
--R - 1247372229446701f + 318578565459621203f
--R +
--R 31
--R 25
--R 498594866849974751463f + 624542545845791047935f
--R +
--R 19
--R 13
--R 93108575769682885f - 62415066358241706387f
--R +
--R 7
--R - 499881859388360475647f - 3926885313819527351f
--R *
--R e
--R +
--R 44
--R 38
--R - 7026011547118141f + 17944427051950691243f
--R +
--R 32
--R 26
--R 2808383522593986603543f + 3513624142354807530135f
--R +
--R 20
--R 14
--R 2860757006705537685f - 3511356735642190737267f
--R +
--R 8
--R 2
--R - 2811332494697103819887f - 20315011631522847311f
--R ,
--R (7936758996566400e - 7936758996566400f)c
--R +
--R 43
--R 37
--R - 4418748183673f + 11285568707456559f + 176599861729451019f
--R +
--R 25
--R 19
--R 2173749283622606155f - 55788292195402895f
--R +
--R 13
--R 7
--R - 2215291421788292951f - 1718142665347430851f + 30256569458230237f
--R *
CHAPTER 13. CHAPTER L

\[\begin{align*}
  &- R \quad e \\
  &- R \quad + \quad 44 \quad 38 \quad 32 \\
  &- R \quad 4418748183673f \quad - 11285668707456559f \quad - 1765998617294451019f \\
  &- R \quad + \quad 26 \quad 20 \quad 14 \\
  &- R \quad - 2173749283622606155f \quad + 55788292195402895f \quad + 2215291421788292951f \\
  &- R \quad + \quad 8 \quad 2 \\
  &- R \quad 1718142665347430851f \quad - 30256569458230237f \\
  &- R \quad , \\
  &- R \quad 6 \quad 43 \\
  &- R \quad (72152354514240f \quad - 72152354514240)c \quad + 40950859449f \\
  &- R \quad + \quad 37 \quad 31 \quad 25 \\
  &- R \quad - 104588980990367f \quad - 1636727395575307f \quad - 20268523416527355f \\
  &- R \quad + \quad 19 \quad 13 \quad 7 \\
  &- R \quad 442205002259535f \quad + 20576059935789063f \quad + 15997133796970563f \\
  &- R \quad + \quad 275099892785581f \\
  &- R \quad , \\
  &- R \quad 3 \quad 2 \quad 2 \\
  &- R \quad 1984189749141600d \quad + 5952569247424800f \quad d \quad - 5952569247424800f \quad d \\
  &- R \quad + \quad 4 \quad 5 \quad 5 \quad 4 \quad 3 \\
  &- R \quad - 3968379498283200f \quad e \quad + 15873517993132800f \quad e \quad + 1785770774227400e \\
  &- R \quad + \quad 7 \quad 2 \\
  &- R \quad ( - 148814231185620000f \quad - 162703559429611200f)e \\
  &- R \quad + \quad 44 \quad 38 \\
  &- R \quad - 390000914678878f \quad + 996062704593756434f \\
  &- R \quad + \quad 32 \quad 26 \\
  &- R \quad 155886323972034823914f \quad + 194745956143985421330f \\
  &- R \quad + \quad 20 \quad 14 \\
  &- R \quad 62050775955574430f \quad - 194596512653299068786f \\
  &- R \quad + \quad 8 \quad 2 \\
  &- R \quad - 155796897940756922666f \quad - 1036375759077320978f \\
  &- R \quad * \\
  &- R \quad e \\
  &- R \quad + \quad 45 \quad 39 \quad 33 \\
  &- R \quad - 374998630035991f \quad + 957747106595453993f \quad + 149889155566764891693f \\
  &- R \quad + 
\end{align*}\]
--R  27  21
--R 187154171443494641685f - 127129015426348065f
--R +
--R  15  9  3
--R - 187241532343115040417f - 149719985567976534037f - 836654081239648061f
--R ,
--R
--R  (5952569247424800e - 5952569247424800f)d - 3968379498283200f e
--R +
--R  4  4  5  3
--R  9920948745708000f e - 3968379498283200f e
--R +
--R  6  2
--R (- 148814231185620000f - 150798420934761600)e
--R +
--R  43  37
--R  492558110242653f - 1257992359608074599f
--R +
--R  31  25
--R - 196883094539368513959f - 246562115745735428056f
--R +
--R  19  13
--R - 325698701993885505f + 24641776988365180111f
--R +
--R  7
--R  197327352068200652911f + 1523373796389332143f
--R *
--R e
--R +
--R  44  38  32
--R  2679481081803026f - 6843392695421906608f - 1071020459642646913578f
--R +
--R  26  20
--R - 1339789169692041240060f - 852746750910750210f
--R +
--R  14  8
--R  1339105101917878401312f + 1071900289758712984762f
--R +
--R  2
--R  7555239072072727756f
--R ,
--R  6  2  5
--R (11905138494849600f - 11905138494849600)d - 7936758996566400f e
--R +
--R  3  4  4  3
--R  31747035986265600f e - 31747035986265600f e
--R +
--R  11  5  2
\[
\begin{align*}
&\text{--R} & 104889507084213371570f & + 167117997269207870f \\
&\text{--R} & + & 12 & 6 \\
&\text{--R} & - 104793725781390615514f & - 83842685189903180394f & - 569978796672974242f \\
&\text{--R} & , \\
&\text{--R} & 6 & 3 \\
&\text{--R} & (25438330117200f + 25438330117200f) & + 76314990351600f + 76314990351600f \\
&\text{--R} & + & 7 & 2 \\
&\text{--R} & (76314990351600f + 76314990351600f) & + 460507167940725f \\
&\text{--R} & + & 44 & 38 & 32 \\
&\text{--R} & - 1594966552735f & + 4075343370415745f & + 637527159231148925f \\
&\text{--R} & + & 26 & 20 & 14 \\
&\text{--R} & 797521176113606525f & + 530440941097175f & - 797160527306433145f \\
&\text{--R} & + & 8 & 2 \\
&\text{--R} & - 638132320196044965f & - 4510507167940725f \\
&\text{--R} & * & e \\
&\text{--R} & + & 45 & 39 & 33 \\
&\text{--R} & - 6036376800443f & + 15416903421476909f & + 241280764619230449f \\
&\text{--R} & + & 27 & 21 & 15 \\
&\text{--R} & 3017679923028013705f & + 1422320037411955f & - 3016650402417843941f \\
&\text{--R} & + & 9 & 3 \\
&\text{--R} & - 2414249368183033161f & - 16561862361763873f \\
&\text{--R} & , \\
&\text{--R} & 12 & 2 \\
&\text{--R} & (1387545279120f - 1387545279120f) & + 6168756279120f \\
&\text{--R} & + & 43 & 37 & 31 \\
&\text{--R} & 4321823003f & - 11037922310209f & - 1727510711947989f \\
&\text{--R} & + & 25 & 19 & 13 \\
&\text{--R} & - 2165150991154425f & - 5114342560755f & + 2162682824948601f \\
&\text{--R} & + & 7 \\
&\text{--R} & 1732620732685741f & + 13506088516033f \\
&\text{--R} & * & e \\
&\text{--R} & + & 44 & 38 & 32 \\
&\text{--R} & 24177661775f & - 61749727185325f & - 9664106795754225f \\
&\text{--R} & + 
\end{align*}
\]
\[
\text{Type: List(NewSparseMultivariatePolynomial(Integer,OrderedVariableList([a,b,c,d,e,f])))}
\]

\[
\text{\texttt{lexTriangular}}(\texttt{lg}, \texttt{false})$\texttt{lextripack}
\]

\[
(14)
\]

\[
\begin{align*}
&\{f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1, \\
&2, 5, 3, 4, 4, 3, 3, 5, 2
\end{align*}
\]

\[
\begin{align*}
&d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f, \\
&2, 5, 3, 4, 4, 3, 5, 2
\end{align*}
\]

\[
\begin{align*}
&3b + 2f e - 5f e - 5f e + 10f e - 4e + 7f, \\
&2, 5, 3, 4, 4, 3, 5, 2
\end{align*}
\]

\[
\begin{align*}
&a - f e + 3f e - 3f e + 4f e + 3e - 3f
\end{align*}
\]

\[
\begin{align*}
&6, 2, 2, 2
\end{align*}
\]

\[
\begin{align*}
&\{f - 1, e - f, d - f, c + 4f c + f, (c - f)b - f c - 5f ,a + b + c + 3f}, \\
&6, 2, 2
\end{align*}
\]

\[
\begin{align*}
&\{f - 1, e - f, d - f, c - f, b + 4f b + f ,a + b + 4f}, \\
&6, 2, 2
\end{align*}
\]

\[
\begin{align*}
&\{f - 1, e - f, d + 4f d + f , (d - f)c - f d - 5f ,b - f, a + c + d + 3f}, \\
&36, 30, 24, 18, 12, 6
\end{align*}
\]

\[
\begin{align*}
&\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \\
&12, 2
\end{align*}
\]

\[
\begin{align*}
&(161718564f - 161718564)e \\
&+ 31, 25
\end{align*}
\]

\[
\begin{align*}
&- 504205f + 1287737951f + 201539391380f + 253982817368f \\
&+ 7
\end{align*}
\]

\[
\begin{align*}
&201940704665f + 1574134601f \\
&* e \\
&+ 32, 26
\end{align*}
\]

\[
\begin{align*}
&- 2818405f + 7198203911f + 1126548149060f + 1416530563364f
\end{align*}
\]
\[
\begin{align*}
\text{PACKAGE LEXTRIPK LEXTRIANGULARPACKAGE} & \\
8 & 2 \\
1127377589345f + 7988820725f & \\
, & \\
6 & 25 34 \\
(693772639560f - 693772639560)d - 462515093040f e + 1850060372160f e & \\
+ & \\
43 11 52 \\
- 1850060372160f e + (-24513299931120f - 23588269745040f)e & \\
+ & \\
30 24 18 \\
- 890810428f + 227518104754f + 355937263869776f & \\
+ & \\
12 6 \\
413736880104344f + 34284930487996f + 3704966481878 & \\
* & \\
e & \\
+ & \\
31 25 19 \\
- 4163798003f + 10634395752169f + 1664161760192806f & \\
+ & \\
13 7 \\
2079424391370694f + 1668153650635921f + 1092427438973f & \\
, & \\
6 & 31 25 \\
(12614047992f - 12614047992)c - 7246825f + 18508536599f & \\
+ & \\
19 13 7 \\
2896249516034f + 3581539649666f + 2796477571739f - 48094301893f & \\
, & \\
6 & 25 34 \\
(693772639560f - 693772639560)b - 925030186080f e + 2312575465200f e & \\
+ & \\
43 11 52 \\
- 2312575465200f e + (-40007555547960f - 35382404617560f)e & \\
+ & \\
30 24 18 \\
- 3781280823f + 9657492291789f + 1511158913397906f & \\
+ & \\
12 6 \\
1837290892286154f + 1487216006594361f + 8077238712093 & \\
* & \\
e & \\
+ & \\
31 25 19 \\
- 9736390478f + 24866827916734f + 389149681905296f & \\
+ & \\
13 7
\end{align*}
\]
\[4872556418871424f + 3904047887269606f + 27890075838538f\]
\[a + b + c + d + e + f\]
\[62222\]
\{f - 1, e + 4f e + f, (e - f) d - f e - 5f, c - f, b - f, a + d + e + 3f\}
\[\text{Type: List(RegularChain(Integer, [a,b,c,d,e,f]))}\]

\[lts := \text{lexTriangular(lg, true)} \cdot \text{lextripack}\]
\[(15)\]
\[[\]
\[6652433425\]
\{f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,\]
\[25344352\]
\[3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,\]
\[25344352\]
\[3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f,\]
\[25344352\]
\[a - f e + 3f e - 3f e + 4f e + 3e - 3f\]
\[,\]
\[6622\]
\{f - 1, e - f, d - f, c + 4f c + f, b + c + 4f, a - f\},
\[622\]
\{f - 1, e - f, d - f, c - f, b + 4f b + f, a + b + 4f\},
\[622\]
\{f - 1, e - f, d + 4f d + f, c + d + 4f, b - f, a - f\},
\[36302418126\]
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1,\]
\[2\]
\[1387545279120e\]
\[+\]
\[312519\]
\[4321823003f - 11037922310209f - 1727506390124986f\]
\[+\]
\[137\]
\[2 - 2176188913464634f - 1732620732685741f - 13506088516033f\]
\[*\]
\[e\]
\[+\]
\[322620\]
\[24177661775f - 61749727185325f - 9664082618092450f\]
\[+\]
\[1482\]
\[12152237485813570f - 9672870290826025f - 68544102808525f\]
--R
--R
--R 1387545279120d
--R +
--R 30 24 18
--R - 1128983050f + 2883434331830f + 451234998755840f
--R +
--R 12 6
--R 562426491685760f + 447129055314890f - 165557857270
--R *
--R e
--R +
--R 31 25 19
--R - 1816935351f + 4640452214013f + 726247129626942f
--R +
--R 13 7
--R 912871801716798f + 726583262666877f + 4909358645961f
--R ,
--R ,
--R 1387545279120c + 778171189f - 1987468196267f - 310993566954378f
--R +
--R 13 7
--R - 383262822316802f - 30035488637543f + 5289595037041f
--R ,
--R ,
--R 1387545279120b
--R +
--R 30 24 18
--R 1128983050f - 2883434331830f - 451234998755840f
--R +
--R 12 6
--R - 562426491685760f - 447129055314890f + 165557857270
--R *
--R e
--R +
--R 31 25 19
--R - 3283058841f + 8384938292463f + 1312252817452422f
--R +
--R 13 7
--R 1646579934064638f + 1306372958656407f + 4694680112151f
--R ,
--R ,
--R 1387545279120a + 1387545279120e + 4321823003f - 11037922310209f
--R +
--R 19 13 7
--R - 1727506390124986f - 2176188913464634f - 1732620732685741f
--R +
--R - 13506088516033f
\[
\begin{align*}
\{f - 1, e + 4f, e + f, d + e + 4f, c - f, b - f, a - f\} \\
\text{Type: List(RegularChain(Integer, [a, b, c, d, e, f]))}
\end{align*}
\]

\[
\begin{align*}
\text{[init}(p) \text{ for } p \text{ in } (ts :: \text{List(P)}) \text{ for } ts \text{ in lts}] \\
\text{-R} \\
\text{-R} (16) \\
\text{-R} [[1,3,1,3,1,1], [1,1,1,1,1,1], [1,1,1,1,1], [1,1,1,1,1]] \\
\text{-R} [1387545279120, 1387545279120, 1387545279120, 1387545279120, 1387545279120, 1] \\
\text{-R} [1,1,1,1,1,1] \\
\text{-RTypem: List(List(NewSparseMultivariatePolynomial(Integer, OrderedVariableList([a, b, c, d, e, f])))])}
\end{align*}
\]

\[
\begin{align*}
squareFreeLexTriangular(lg, true) \$\text{lextripack} \\
\text{-R} \\
\text{-R} (17) \\
\text{-R} [ \\
\text{-R} 6 \ 6 \ 5 \ 2 \ 4 \ 3 \ 3 \ 4 \ 2 \ 5 \\
\text{-R} \{f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1, \\
\text{-R} 2 \ 5 \ 3 \ 4 \ 4 \ 3 \ 5 \ 2 \\
\text{-R} 3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f, \\
\text{-R} 2 \ 5 \ 3 \ 4 \ 4 \ 3 \ 5 \ 2 \\
\text{-R} 3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f, \\
\text{-R} 2 \ 5 \ 3 \ 4 \ 4 \ 3 \ 5 \ 2 \\
\text{-R} a - f e + 3f e - 3f e + 4f e + 3e - 3f} \\
\text{-R} , \\
\text{-R} 6 \ 2 \ 2 \\
\text{-R} \{f - 1, e - f, d - f, c + 4f c + f ,b + c + 4f, a - f\}, \\
\text{-R} 6 \ 2 \ 2 \\
\text{-R} \{f - 1, e - f, d - f, c - f,b + 4f b + f ,a + b + 4f\}, \\
\text{-R} 6 \ 2 \ 2 \\
\text{-R} \{f - 1, e - f, d + 4f d + f ,c + d + 4f,b - f, a - f\}, \\
\text{-R} 36 \ 30 \ 24 \ 18 \ 12 \ 6 \\
\text{-R} \{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \\
\text{-R} 2 \\
\text{-R} 1387545279120e \\
\text{-R} + \\
\text{-R} 4321823003f - 11037922310209f - 1727506390124986f \\
\text{-R} + \\
\text{-R} 13 \ 7
\end{align*}
\]
--R - 2176188913464634f - 1732620732685741f - 13506088516033f
--R *
--R e
--R +
--R 32  26  20
--R 24177661775f - 61749727185325f - 9664082618092450f
--R +
--R  14  8  2
--R - 12152237485813570f - 9672870290826025f - 68544102808525f
--R ,
--R +
--R  1387545279120d
--R +
--R 30  24  18
--R - 1128983050f + 2883434331830f + 451234998755840f
--R +
--R  12  6
--R 562426491685760f + 447129055314890f - 165557857270
--R *
--R e
--R +
--R 31  25  19
--R - 1816935351f + 4640452214013f + 726247129626942f
--R +
--R  13  7
--R 912871801716798f + 726583262666877f + 4909358645961f
--R ,
--R +
--R 31  25  19
--R 1387545279120c + 778171189f - 1987468196267f - 310993556954378f
--R +
--R  13  7
--R - 383262822316802f - 300335488637543f + 5289595037041f
--R ,
--R +
--R 1387545279120b
--R +
--R 30  24  18
--R 1128983050f - 2883434331830f - 451234998755840f
--R +
--R 12  6
--R 562426491685760f - 447129055314890f + 165557857270
--R *
--R e
--R +
--R 31  25  19
--R - 3283058841f + 8384938292463f + 1312252817452422f
--R +
--R  13  7
--R 1646579934064638f + 1306372958656407f + 4694680112151f
((18) 156
Type: PositiveInteger

(19) [a, b, c, d, e, f, %A]
Type: List(Symbol)

(20) ZeroDimensionalSolvePackage(Integer, [a, b, c, d, e, f], [a, b, c, d, e, f, %A])
Type: Domain

(21)
[complexRoots = ? - 13? + 49,
coordinates =
\[ \begin{align*} &\text{coordinates } = \\
&\quad [35a + 3\%A + 19\%A, 35b + \%A + 18\%A, 35c - 2\%A - \%A, \\
&\quad 35d - 3\%A - 19\%A, 35e - \%A - 18\%A, 35f + 2\%A + \%A] \\
&\quad ] \\
&\quad , \\
&\quad [ \\
&\quad \text{complexRoots } = \\
&\quad 8 \quad 7 \quad 6 \quad 5 \quad 4 \quad 3 \quad 2 \\
&\quad , \\
&\quad \text{coordinates } = \\
&\quad [ \\
&\quad \quad 43054532a + 33782\%A - 546673\%A + 3127348\%A - 6927123\%A \\
&\quad + \\
&\quad \quad 3 \quad 2 \\
&\quad \quad 4365212\%A - 25086957\%A + 39582814\%A - 107313172 \\
&\quad , \\
&\quad ] \\
&\quad , \\
&\quad [ \\
&\quad \quad 43054532b - 33782\%A + 546673\%A - 3127348\%A + 6927123\%A \\
&\quad + \\
&\quad \quad 3 \quad 2 \\
&\quad \quad - 4365212\%A + 25086957\%A - 39582814\%A + 107313172 \\
&\quad , \\
&\quad ] \\
&\quad , \\
&\quad [ \\
&\quad \quad 21527266c - 22306\%A + 263139\%A - 1166076\%A + 1821805\%A \\
&\quad + \\
&\quad \quad 3 \quad 2 \\
&\quad \quad - 2892788\%A + 10322663\%A - 9026596\%A + 12950740 \\
&\quad , \\
&\quad ] \\
&\quad , \\
&\quad [ \\
&\quad \quad 43054532d + 22306\%A - 263139\%A + 1166076\%A - 1821805\%A \\
&\quad + \\
&\quad \quad 7 \quad 6 \quad 5 \quad 4 \\
\end{align*} \]
complexRoots =
8 7 6 5 4 3 2
,
coordinates =
[
7 6 5 4
43064532a + 33782%A + 546673%A + 3127348%A + 6927123%A
+
3 2
4365212%A + 25086957%A + 39582814%A + 107313172
,
7 6 5 4
43064532b - 33782%A - 546673%A - 3127348%A - 6927123%A
+
3 2
- 4365212%A - 25086957%A - 39582814%A - 107313172
,
7 6 5 4
21527266c - 22306%A - 263139%A - 1166076%A - 1821805%A
+
3 2
- 2892788%A - 10322663%A - 9026596%A - 12950740
,
`package Lextripk Lextraangularpackage`

```r
--R 43054532d + 22306%A + 263139%A + 1166076%A + 1821805%A
--R +
--R 3 2
--R 2892788%A + 10322663%A + 30553862%A + 12950740
--R ,
--R
--R 7 6 5 4
--R 43054532e - 22306%A - 263139%A - 1166076%A - 1821805%A
--R +
--R 3 2
--R - 2892788%A - 10322663%A - 30553862%A - 12950740
--R ,
--R
--R 7 6 5 4
--R 21527266f + 22306%A + 263139%A + 1166076%A + 1821805%A
--R +
--R 3 2
--R 2892788%A + 10322663%A + 9026596%A + 12950740
--R ]
--R]
--R ]
--R
--R 4 2
--R ]complexRoots= ? - ? + 1,
--R 3 3 3 3
--R coordinates= [a - %A,b + %A - %A,c + %A ,d + %A,e - %A + %A,f - %A ]]
--R ,
--R
--R 8 6 4 2
--R coordinates =
--R 7 5 3 3 3 3
--R [4a - 2%A - 7%A - 20%A - 22%A, 4b + 2%A + 7%A + 20%A + 22%A,
--R 7 5 3 3 3 3
--R 4c + %A + 3%A + 10%A + 10%A, 4d + %A + 3%A + 10%A + 6%A,
--R 7 5 3 3 3 3
--R 4e - %A - 3%A - 10%A - 6%A, 4f - %A - 3%A - 10%A - 10%A]
--R]
--R]
--R
--R 4 3 2
--R coordinates =
--R 3 2 3 2
--R [30a - %A - 5%A - 30%A - 6, 6b + %A + 5%A + 24%A + 6,
--R 3 2 3 2
--R 30c - %A - 5%A - 6, 30d - %A - 5%A - 30%A - 6,
--R 3 2 3 2
```
--R \[\{\begin{array}{>{\scriptstyle}{l}}
30e - %A - 5%A - 30%A - 6, \\
30f - %A - 5%A - 30%A - 6
\end{array}\}\]

coordinates =
\[
\begin{array}{>{\scriptstyle}{l}}
3 & 2 & 3 & 2 \\
\{30a - %A + 5%A - 30%A + 6, \\
6b + %A - 5%A + 24%A - 6,
\end{array}
\]

coordinates =
\[
\begin{array}{>{\scriptstyle}{l}}
3 & 2 & 3 & 2 \\
\{30c - %A + 5%A + 6, \\
30d - %A - 5%A + 30%A + 6,
\end{array}
\]

coordinates =
\[
\begin{array}{>{\scriptstyle}{l}}
3 & 2 & 3 & 2 \\
\{30e - %A - 5%A - 30%A - 6, \\
30f - %A - 5%A - 30%A - 6.
\end{array}
\]
coordinates = \\
\[ 7 6 5 4 3 2 \\
\] \\
\[ 2a + 2%A + 9%A + 18%A + 19%A + 4%A - 10%A - 2%A + 4, \\
7 6 5 4 3 2 \\
\]
\[ 2b + 2%A + 9%A + 18%A + 19%A + 4%A - 10%A - 4%A + 4, \\
7 6 5 4 3 \\
\]
\[ 2c - %A - 4%A - 8%A - 9%A - 4%A - 2%A - 4, \\
7 6 5 4 3 \\
\]
\[ 2d + %A + 4%A + 8%A + 9%A + 4%A + 2%A + 4, \\
7 6 5 4 3 2 \\
\]
\[ 2e - 2%A - 9%A - 18%A - 19%A - 4%A + 10%A + 4%A - 4, \\
7 6 5 4 3 \\
\]
\[ 2f - 2%A - 9%A - 18%A - 19%A - 4%A + 10%A + 2%A - 4] \\
\]

complexRoots = \\
\[ 7 6 5 4 3 2 \\
\]
\[ 8 7 6 5 4 3 2 \\
\]
\]
\[ 7 6 5 4 3 2 \\
\]
\[ 1408a - 19%A - 200%A - 912%A - 2216%A - 4544%A - 6784%A \\
\]
\[ + \\
\]
\[ - 6976%A - 1792 \\
\]
\[ 7 6 5 4 3 2 \\
\]
\[ 1408b - 37%A - 408%A - 1952%A - 5024%A - 10368%A - 16768%A \\
\]
\[ + \\
\]
\[ - 17920%A - 5120 \\
\]
\[ 7 6 5 4 3 2 \\
\]
\[ 1408c + 37%A + 408%A + 1952%A + 5024%A + 10368%A + 16768%A \\
\]
\[ + \\
\]
\[ 17920%A + 5120 \\
\]
\[ 7 6 5 4 3 2 \\
\]
\[ 1408d + 19%A + 200%A + 912%A + 2216%A + 4544%A + 6784%A \\
\]
\[ + \\
\]
\[ 6976%A + 1792 \\
\]
\[ 2e + %A, 2f - %A] \\
\]
---R ]
---R ,
---R ---R 8 6 4 2
---R [complexRoots= ? + 4? + 12? + 16? + 4,
---R ---R coordinates =
---R 7 5 3 7 5 3
---R [4a - %A - 3%A - 10%A - 6%A, 4b - %A - 3%A - 10%A - 10%A,
---R 7 5 3 7 5 3
---R 4c - 2%A - 7%A - 20%A - 22%A, 4d + 2%A + 7%A + 20%A + 22%A,
---R 7 5 3 7 5 3
---R 4e + %A + 3%A + 10%A + 10%A, 4f + %A + 3%A + 10%A + 6%A]}
---R ]
---R ,
---R [complexRoots= ? + 16? - 96? + 256? + 256,
---R ---R coordinates =
---R 7 5 3
---R [512a - %A - 12%A + 176%A - 448%A,
---R 7 5 3
---R 128b - %A - 16%A + 96%A - 256%A,
---R 7 5 3
---R 128c + %A + 16%A - 96%A + 256%A,
---R 7 5 3
---R 512d + %A + 12%A - 176%A + 448%A, 2e + %A, 2f - %A]
---R ]
---R ,
---R [ complexRoots =
---R 8 7 6 5 4 3 2
---R ,
---R ,
---R coordinates =
---R [ 7 6 5 4 3 2
---R [1408a - 19%A + 200%A - 912%A + 2216%A = 4544%A + 6784%A
---R +
---R - 6976%A + 1792
---R ,
---R 7 6 5 4 3 2
---R [1408b - 37%A + 408%A - 1952%A + 5024%A - 10368%A + 16768%A
---R +
---R - 17920%A + 5120
---R ,
---R ---R 8 6 4 2
---R [complexRoots= ? + 4? + 12? + 16? + 4,
\[
\begin{align*}
&1408c + 37\%A - 408\%A + 1952\%A - 5024\%A + 10368\%A - 16768\%A \\
&+ 17920\%A - 5120 \\
&1408d + 19\%A - 200\%A + 912\%A - 2216\%A + 4544\%A - 6784\%A \\
&+ 6976\%A - 1792 \\
&2e + \%A, 2f - \%A
\end{align*}
\]

\[
\begin{align*}
\text{coordinates} = \\
&\begin{cases}
    7 & 6 & 5 & 4 & 3 & 2 \\
    2a + 2\%A - 9\%A + 18\%A - 19\%A + 4\%A + 10\%A - 2\%A - 4 \\
    7 & 6 & 5 & 4 & 3 & 2 \\
    2b + 2\%A - 9\%A + 18\%A - 19\%A + 4\%A + 10\%A - 4\%A - 4 \\
    7 & 6 & 5 & 4 & 3 & 2 \\
    2c - \%A + 4\%A - 8\%A + 9\%A - 4\%A - 2\%A + 4 \\
    7 & 6 & 5 & 4 & 3 & 2 \\
    2d + \%A - 4\%A + 8\%A - 9\%A + 4\%A + 2\%A - 4 \\
    7 & 6 & 5 & 4 & 3 & 2 \\
    2e - 2\%A + 9\%A - 18\%A + 19\%A - 4\%A - 10\%A + 4\%A + 4 \\
    7 & 6 & 5 & 4 & 3 & 2 \\
    2f - 2\%A + 9\%A - 18\%A + 19\%A - 4\%A - 10\%A + 2\%A + 4
\end{cases} \\
\]
\]

\[
\begin{align*}
\text{complexRoots}= & \ ? + 12? + 144, \\
\text{coordinates} = \\
&\begin{cases}
    7 & 6 & 5 & 4 & 3 & 2 \\
    12a - \%A - 12, 12b - \%A - 12, 12c - \%A - 12, 12d - \%A - 12, \\
    2 & 2 \\
    6e + \%A + 3\%A + 12, 6f + \%A - 3\%A + 12
\end{cases} \\
\]
\]

\[
\begin{align*}
\text{complexRoots}= & \ ? + 6? + 30? + 36? + 36, \\
\text{coordinates} = \\
&\begin{cases}
    4 & 3 & 2
\end{cases}
\]


```plaintext
-- R 3 2 3 2
-- R [6a - %A - 5%A - 24%A - 6, 30b + %A + 5%A + 30%A + 6,
-- R 3 2 3 2
-- R 30c + %A + 5%A + 30%A + 6, 30d + %A + 5%A + 30%A + 6,
-- R 3 2 3 2
-- R 30e + %A + 5%A + 30%A + 6, 30f + %A + 5%A + 6]
-- R ]
-- R ,
-- R [complexRoots= ? - 6? + 30? - 36? + 36,
-- R coordinates =
-- R 3 2 3 2
-- R [6a - %A + 5%A - 24%A + 6, 30b + %A - 5%A + 30%A - 6,
-- R 3 2 3 2
-- R 30c + %A - 5%A + 30%A - 6, 30d + %A - 5%A + 30%A - 6,
-- R 3 2 3 2
-- R 30e + %A - 5%A + 30%A - 6, 30f + %A - 5%A - 6]
-- R ]
-- R ,
-- R [complexRoots= ? + 12? + 144,
-- R coordinates =
-- R 2 2 2 2
-- R [12a + %A + 12, 12b + %A + 12, 12c + %A + 12, 12d + %A + 12,
-- R 2 2
-- R 6e - %A + 3%A - 12, 6f - %A - 3%A - 12]
-- R ]
-- R ,
-- R [complexRoots= ? - 12,
-- R coordinates= [a - 1,b - 1,c - 1,d - 1,e + %A + 4,2f - %A + 4]]
-- R ,
-- R [complexRoots= ? + 6? + 6,
-- R coordinates= [a + %A + 5,b - 1,c - 1,d - 1,e - 1,f - %A - 1]]
-- R ,
-- R [complexRoots= ? - 6? + 6,
-- R coordinates= [a + %A - 5,b + 1,c + 1,d + 1,e + 1,f - %A + 1]]
-- R ,
-- R [complexRoots= ? - 12,
```
CHAPTER 13. CHAPTER L

--R coordinates= [a + 1,b + 1,c + 1,d + 1,2e + %A - 4,2f - %A - 4]
--R ,
--R
--R [complexRoots= ? + 6? + 30? + 36? + 36,
--R coordinates =
--R 3 2 3 2
--R [30a - %A - 5%A - 30%A - 6, 30b - %A - 5%A - 30%A - 6,
--R 3 2 3 2
--R 30c - %A - 5%A - 30%A - 6, 6d + %A + 5%A + 24%A + 6,
--R 3 2 3 2
--R 30e - %A - 5%A - 6, 30f - %A - 5%A - 30%A - 6]
--R ]
--R ,
--R
--R [complexRoots= ? - 6? + 30? - 36? + 36,
--R coordinates =
--R 3 2 3 2
--R [30a - %A + 5%A - 30%A + 6, 30b - %A + 5%A - 30%A + 6,
--R 3 2 3 2
--R 30c - %A + 5%A - 30%A + 6, 6d + %A - 5%A + 24%A - 6,
--R 3 2 3 2
--R 30e - %A + 5%A + 6, 30f - %A + 5%A - 30%A + 6]
--R ]
--R ,
--R
--R [complexRoots= ? + 6? + 6,
--R coordinates= [a + 1,b + 1,c + 1,d - %A - 5,e + %A + 1,f + 1]]
--R ,
--R
--R [complexRoots= ? - 6? + 6,
--R coordinates= [a - 1,b - 1,c - 1,d - %A + 5,e + %A - 1,f - 1]]
--R ]
--RType: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer))))
--E 21

concat [realSolve(ts)$zdpack for ts in lts]
--R
--R
--R (22)
--R [[%B1,%B1,%B1,%B5,- %B5 - 4%B1,%B1], [%B1,%B1,%B1,%B6,- %B6 - 4%B1,%B1],
--R [%B7,%B7,%B7,%B11,- %B11 - 4%B7], [%B7,%B7,%B7,%B7,%B12,- %B12 - 4%B7],
--R [%B8,%B8,%B8,%B9,- %B9 - 4%B8], [%B8,%B8,%B8,%B10,- %B10 - 4%B8],
--S 22 of 23
\[
\begin{align*}
&\frac{\%B13, \%B13, \%B17, - \%B17 - 4\%B13, \%B13, \%B13}{\%B13, \%B13, \%B18, - \%B18 - 4\%B13, \%B13, \%B13}, \\
&\frac{\%B14, \%B14, \%B15, - \%B15 - 4\%B14, \%B14, \%B14}{\%B14, \%B14, \%B16, - \%B16 - 4\%B14, \%B14, \%B14}, \\
&\%B19, \%B29, \\
&7865521 \quad 31 \quad 6696179241 \quad 25 \quad 25769893181 \quad 19 \quad 3003344760 \quad 2002229840 \quad 6006689520, \\
&\%B19 - \%B19 + \%B19 + \%B19 - \%B19 - \%B19, \\
&- \%B19 + \%B19 + \%B19 - \%B19 - \%B19, \\
&778171189 \quad 31 \quad 1987468196267 \quad 25 \quad 155496778477189 \quad 19 \quad 3003344760 \quad 2002229840 \quad 6006689520, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&191634111158401 \quad 13 \quad 30033548637543 \quad 7 \quad 755656433863 \quad 693772639560 \quad 6006689520, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&1094352947 \quad 31 \quad 2794979430821 \quad 25 \quad 21870802908737 \quad 19 \quad 462515093040 \quad 462515093040 \quad 231257546520, \\
&\%B19 - \%B19 \quad \%B19 - \%B19 \quad \%B19 - \%B19 \quad \%B19 - \%B19, \\
&91476663003591 \quad 13 \quad 145152550961823 \quad 7 \quad 1564893370717 \quad 693772639560 \quad 693772639560, \\
&\%B19 - \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&77085848840 \quad 154171697680 \quad 462515093040 \quad 6006689520, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&4321823003 \quad 31 \quad 180949546069 \quad 25 \quad 22746643920 \quad 863753195062493 \quad 19 \quad 1088094456732317 \quad 13, \\
&\%B29 - \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&863753195062493 \quad 19 \quad 1088094456732317 \quad 13 \quad 693772639560 \quad 693772639560 \quad 693772639560, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&1732620732685741 \quad 7 \quad 13506088516033 \quad 13 \quad 1387545279120 \quad 1387545279120 \quad 1387545279120, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&\%B19 + \%B19 + \%B19 + \%B19 + \%B19, \\
&\frac{\%B19, \%B30}{\frac{\%B19, \%B30,}{}}.
\end{align*}
\]
<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>7865521</td>
<td>31</td>
<td>6696179241</td>
<td>25</td>
<td>25769893181</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>600689520</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1975912990729</td>
<td>13</td>
<td>1048460696489</td>
<td>7</td>
<td>21252634831</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3003344760</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>778171189</td>
<td>31</td>
<td>1987468196267</td>
<td>25</td>
<td>155496778477189</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>1387545279120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>191631411158401</td>
<td>13</td>
<td>300335488637543</td>
<td>7</td>
<td>755656433863</td>
<td></td>
<td></td>
</tr>
<tr>
<td>693772639560</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1094352947</td>
<td>31</td>
<td>2794979430821</td>
<td>25</td>
<td>218708802908737</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>462515093040</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>91476663003591</td>
<td>13</td>
<td>145152550961823</td>
<td>7</td>
<td>1564893370717</td>
<td></td>
<td></td>
</tr>
<tr>
<td>77085848840</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4321823003</td>
<td>31</td>
<td>180949546069</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1387545279120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>863753195062493</td>
<td>19</td>
<td>1088094456732317</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>693772639560</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1732620732685741</td>
<td>7</td>
<td>13506088516033</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1387545279120</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7865521</td>
<td>31</td>
<td>6696179241</td>
<td>25</td>
<td>25769893181</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>600689520</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1975912990729</td>
<td>13</td>
<td>1048460696489</td>
<td>7</td>
<td>21252634831</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3003344760</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Note: The table contains numerical values and some operations indicated by the symbols + and -.]
<table>
<thead>
<tr>
<th>Operator</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>191631411158401 13 300335488637543 7 756656433863</td>
</tr>
<tr>
<td>-</td>
<td>693772639560 1387545279120 198220754160</td>
</tr>
<tr>
<td>+</td>
<td>1094352947 31 2794979430821 25 218708802908737 19</td>
</tr>
<tr>
<td>-</td>
<td>462515093040 693772639560 231257546520</td>
</tr>
<tr>
<td>+</td>
<td>1975912990729 13 1048460696489 7 21252634831</td>
</tr>
<tr>
<td>-</td>
<td>778171189 31 1987468196267 25 155496778477189 19</td>
</tr>
<tr>
<td>+</td>
<td>1387545279120 693772639560</td>
</tr>
<tr>
<td>+</td>
<td>863753195062493 19 1088094456732317 13</td>
</tr>
<tr>
<td>-</td>
<td>693772639560 1387545279120 693772639560</td>
</tr>
<tr>
<td>+</td>
<td>1732620732685741 7 13506088516033</td>
</tr>
<tr>
<td>-</td>
<td>1387545279120</td>
</tr>
<tr>
<td>+</td>
<td>4321823003 31 180949546069 25</td>
</tr>
<tr>
<td>-</td>
<td>1387545279120 22746643920</td>
</tr>
<tr>
<td>+</td>
<td>1975912990729 13 1048460696489 7 21252634831</td>
</tr>
<tr>
<td>-</td>
<td>778171189 31 1987468196267 25 155496778477189 19</td>
</tr>
<tr>
<td>+</td>
<td>1387545279120 693772639560</td>
</tr>
<tr>
<td>+</td>
<td>191631411158401 13 300335488637543 7 756656433863</td>
</tr>
</tbody>
</table>
### CHAPTER 13. CHAPTER L

<table>
<thead>
<tr>
<th>--R</th>
<th>( \frac{%B_{20} + %B_{20} - %B_{20}}{%B_{20} + %B_{20} - %B_{20}} )</th>
<th>( \frac{%B_{20} - %B_{20} + %B_{20}}{%B_{20} + %B_{20} + %B_{20}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>--R</td>
<td>( \frac{693772639560 + 1387545279120}{198220754160} )</td>
<td>( \frac{31}{2794979430821} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{25}{218708802908737} )</td>
<td>( \frac{19}{145152550961823} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{1387545279120 + 462515093040}{231257546520} )</td>
<td>( \frac{1987545279120 + 1387545279120}{198220754160} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{154171697680 + 462515093040}{231257546520} )</td>
<td>( \frac{462515093040 + 1387545279120}{231257546520} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{693772639560 + 1387545279120}{231257546520} )</td>
<td>( \frac{462515093040 + 1387545279120}{231257546520} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{1094352947 + 145152550961823}{218708802908737} )</td>
<td>( \frac{25}{218708802908737} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{19}{145152550961823} )</td>
<td>( \frac{19}{145152550961823} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{77085848840 + 154171697680}{231257546520} )</td>
<td>( \frac{462515093040 + 1387545279120}{231257546520} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{1094352947 + 145152550961823}{218708802908737} )</td>
<td>( \frac{25}{218708802908737} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{19}{145152550961823} )</td>
<td>( \frac{19}{145152550961823} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{77085848840 + 154171697680}{231257546520} )</td>
<td>( \frac{462515093040 + 1387545279120}{231257546520} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{1094352947 + 145152550961823}{218708802908737} )</td>
<td>( \frac{25}{218708802908737} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{19}{145152550961823} )</td>
<td>( \frac{19}{145152550961823} )</td>
</tr>
<tr>
<td>--R</td>
<td>( \frac{77085848840 + 154171697680}{231257546520} )</td>
<td>( \frac{462515093040 + 1387545279120}{231257546520} )</td>
</tr>
</tbody>
</table>

\[ \%B_{21}, \%B_{25}, \]

\[ \frac{7865521}{31} \frac{6696179241}{25} \frac{25769893181}{19} \]

\[ \frac{6006689620}{2002229840} \frac{49235160}{49235160} \]

\[ \frac{1975912990729}{13} \frac{1048460696489}{7} \frac{21252634831}{19} \]

\[ \frac{3003344760}{2002229840} \frac{6006689520}{6006689520} \]

\[ \frac{778171189}{31} \frac{1987468196267}{25} \frac{155496778477189}{19} \]

\[ \frac{1387545279120}{1387545279120} \frac{693772639560}{693772639560} \]

\[ \frac{191631411158401}{13} \frac{300335488637543}{7} \frac{755656438563}{19} \]

\[ \frac{693772639560}{1387545279120} \frac{198220754160}{198220754160} \]

\[ \frac{1094352947}{31} \frac{2794979430821}{25} \frac{218708802908737}{19} \]

\[ \frac{462515093040}{462515093040} \frac{231257546520}{231257546520} \]
\[ \%B22, \%B23, \]
\[ \%B22 + \%B21 + \%B22 \]
\[ 7865521 31 6696179241 25 25769893181 19 \]
\[ 6006689520 2002229840 49235160 \]
\[ 191631411158401 13 30033588637543 7 21252634831 \]
\[ 3003344760 2002229840 6006689520 \]
\[ 778171189 31 1987468196267 25 15549677847189 19 \]
\[ 693772639560 1387545279120 198220754160 \]
\[ 4321823003 31 180949546069 25 \]
\[ 863753195062493 19 1088094456732317 13 \]
\[
\begin{align*}
\text{--R} & \quad 1732620732685741 \quad 7 \quad 13506088516033 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 1387545279120 \quad 1387545279120 \\
\text{--R} & \quad \} \\
\text{--R} & \quad , \\
\text{--R} & \quad [\%B22, \%B24, \\
\text{--R} & \quad 7865521 \quad 31 \quad 6696179241 \quad 25 \quad 25769893181 \quad 19 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 6006689520 \quad 2002229840 \quad 49235160 \\
\text{--R} & \quad + \\
\text{--R} & \quad 1975912990729 \quad 13 \quad 1048460696489 \quad 7 \quad 21252634831 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 3003344760 \quad 2002229840 \quad 6006689520 \\
\text{--R} & \quad , \\
\text{--R} & \quad 778171189 \quad 31 \quad 1987468196267 \quad 25 \quad 155496778477189 \quad 19 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 1387545279120 \quad 1387545279120 \quad 693772639560 \\
\text{--R} & \quad + \\
\text{--R} & \quad 191631411158401 \quad 13 \quad 300335488637543 \quad 7 \quad 75656433863 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 693772639560 \quad 1387545279120 \quad 198220754160 \\
\text{--R} & \quad , \\
\text{--R} & \quad 1094352947 \quad 31 \quad 2794979430821 \quad 25 \quad 218708802908737 \quad 19 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 462515093040 \quad 462515093040 \quad 231257546520 \\
\text{--R} & \quad + \\
\text{--R} & \quad 91476663003591 \quad 13 \quad 145152550961823 \quad 7 \quad 1564893370717 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \quad - \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 77085848840 \quad 154171697680 \quad 462515093040 \\
\text{--R} & \quad , \\
\text{--R} & \quad 4321823003 \quad 31 \quad 180949546069 \quad 25 \\
\text{--R} & \quad \%B24 \quad - \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 1387545279120 \quad 22746643920 \\
\text{--R} & \quad + \\
\text{--R} & \quad 863753195062493 \quad 19 \quad 1088094456732317 \quad 13 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 693772639560 \quad 693772639560 \\
\text{--R} & \quad + \\
\text{--R} & \quad 1732620732685741 \quad 7 \quad 13506088516033 \\
\text{--R} & \quad \text{---} \quad \%B22 \quad + \quad \text{---} \quad \%B22 \\
\text{--R} & \quad 1387545279120 \quad 1387545279120 \\
\text{--R} & \quad ] \\
\text{--R} & \quad , \\
\text{--R} & \quad [\%B31, \%B35, - \%B35 - 4\%B31, \%B31, \%B31], \\
\text{--R} & \quad [\%B31, \%B36, - \%B36 - 4\%B31, \%B31, \%B31],
\end{align*}
\]
CHAPTER 13. CHAPTER L

---R [2B32,3B33,- 3B33 - 42B32,2B32,2B32,2B32],
---R Type: List(List(RealClosure(Fraction(Integer))))
---E 22

---S 23 of 23
)show LexTriangularPackage
---R
---R LexTriangularPackage(R: GcdDomain,ls: List(Symbol)) is a package constructor
---R Abbreviation for LexTriangularPackage is LEXTRIPK
---R This constructor is not exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for LEXTRIPK
---R
---R------------------------------- Operations --------------------------------
---R fglmIfCan : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> Union(List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),"failed")
---R groebner : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),
---R lexTriangular : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),Boolean) -> List(RegularChain(R,ls)),
---R squareFreeLexTriangular : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),Boolean) -> Boolean
---R zeroDimensional? : List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))) -> Boolean
---R zeroSetSplit : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),Boolean) -> List(RegularChain(R,ls)),
---R zeroSetSplit : (List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls))),Boolean) ->
---R
---E 23
)
)
)

---

LexTriangularPackage.help

====================================================================
LexTriangularPackage examples
====================================================================

A package for solving polynomial systems with finitely many solutions.

The decompositions are given by means of regular triangular sets. The computations use lexicographical Groebner bases. The main operations are lexTriangular and squareFreeLexTriangular. The second one provides decompositions by means of square-free regular triangular sets.

Both are based on the lexTriangular method described in
   D. LAZARD "Solving Zero-dimensional Algebraic Systems"

They differ from the algorithm described in
   M. MORENO MAZA and R. RIOBOO "Computations of gcd over algebraic towers of simple extensions"
by the fact that multiplicities of the roots are not kept. With the
squareFreeLexTriangular operation all multiplicities are removed.
With the other operation some multiplicities may remain. Both operations
admit an optional argument to produce normalized triangular sets.

The LexTriangularPackage package constructor provides an
implementation of the lexTriangular algorithm (D. Lazard "Solving
This algorithm decomposes a zero-dimensional variety into zero-sets of
regular triangular sets. Thus the input system must have a finite
number of complex solutions. Moreover, this system needs to be a
lexicographical Groebner basis.

This package takes two arguments: the coefficient-ring R of the
polynomials, which must be a GcdDomain and their set of variables
given by ls a List Symbol. The type of the input polynomials must be
NewSparseMultivariatePolynomial(R,V) where V is OrderedVariableList(ls).
The abbreviation for LexTriangularPackage is LEXTRIPK. The main operations
are lexTriangular and squareFreeLexTriangular. The later provide
decompositions by means of square-free regular triangular sets, built
with the SquareFreeRegularTriangularSet constructor, whereas the former
uses the RegularTriangularSet constructor. Note that these constructors
also implement another algorithm for solving algebraic systems by
means of regular triangular sets; in that case no computations of
Groebner bases are needed and the input system may have any dimension
(i.e. it may have an infinite number of solutions).

The implementation of the lexTriangular algorithm provided in the
LexTriangularPackage constructor differs from that reported in
"Computations of gcd over algebraic towers of simple extensions" by
Indeed, the squareFreeLexTriangular operation removes all multiplicities
of the solutions (i.e. the computed solutions are pairwise different) and
the lexTriangular operation may keep some multiplicities; this
later operation runs generally faster than the former.

The interest of the lexTriangular algorithm is due to the
following experimental remark. For some examples, a triangular
decomposition of a zero-dimensional variety can be computed faster via
a lexicographical Groebner basis computation than by using a direct method
(like that of SquareFreeRegularTriangularSet and RegularTriangularSet).
This happens typically when the total degree of the system relies
essentially on its smallest variable (like in the Katsura systems).
When this is not the case, the direct method may give better timings
(like in the Rose system).

Of course, the direct method can also be applied to a lexicographical
Groebner basis. However, the lexTriangular algorithm takes advantage
of the structure of this basis and avoids many unnecessary computations.
which are performed by the direct method.

For this purpose of solving algebraic systems with a finite number of solutions, see also the ZeroDimensionalSolvePackage. It allows to use both strategies (the lexTriangular algorithm and the direct method) for computing either the complex or real roots of a system.

Note that the way of understanding triangular decompositions is detailed in the example of the RegularTriangularSet constructor.

Since the LexTriangularPackage package constructor is limited to zero-dimensional systems, it provides a zeroDimensional? operation to check whether this requirement holds. There is also a groebner operation to compute the lexicographical Groebner basis of a set of polynomials with type NewSparseMultivariatePolynomial(R,V). The elimination ordering is that given by ls (the greatest variable being the first element of ls). This basis is computed by the FLGM algorithm (Faugere et al. "Efficient Computation of Zero-Dimensional Groebner Bases by Change of Ordering", J. of Symbol. Comput., 1993) implemented in the LinGroebnerPackage package constructor.

Once a lexicographical Groebner basis is computed, then one can call the operations lexTriangular and squareFreeLexTriangular. Note that these operations admit an optional argument to produce normalized triangular sets. There is also a zeroSetSplit operation which does all the job from the input system; an error is produced if this system is not zero-dimensional.

Let us illustrate the facilities of the LexTriangularPackage constructor by a famous example, the cyclic-6 root system.

Define the coefficient ring.

```plaintext
declare (R Integer)
  Type: Domain
```

Define the list of variables,

```plaintext
define (ls : List Symbol := [a,b,c,d,e,f])
  Type: List Symbol
```

and make it an ordered set.

```plaintext
define (V := OVAR(ls))
  Type: Domain
```

Define the polynomial ring.
Define the polynomials.

\[
p_1 := a \cdot b \cdot c \cdot d \cdot e \cdot f - 1 \\
\text{Type: NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

\[
p_2 := a \cdot b \cdot c \cdot d \cdot e + a \cdot b \cdot c \cdot e \cdot f + a \cdot b \cdot d \cdot e \cdot f + a \cdot c \cdot d \cdot e \cdot f + b \cdot c \cdot d \cdot e \cdot f \\
\text{((e + f)d + f e)c + f e d)a + f e d c b} \\
\text{Type: NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

\[
p_3 := a \cdot b \cdot c \cdot d + a \cdot b \cdot c \cdot f + a \cdot b \cdot e \cdot f + a \cdot d \cdot e \cdot f + b \cdot c \cdot d \cdot e + c \cdot d \cdot e \cdot f \\
\text{((d + f)c + f e)b + f e d)a + e d c b + f e d c} \\
\text{Type: NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

\[
p_4 := a \cdot b \cdot c + a \cdot b \cdot f + a \cdot e \cdot f + b \cdot c \cdot d + c \cdot d \cdot e + d \cdot e \cdot f \\
\text{((c + f)b + f e)a + d c b + e d c + f e d} \\
\text{Type: NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

\[
p_5 := a \cdot b + a \cdot f + b \cdot c + c \cdot d + d \cdot e + e \cdot f \\
\text{(b + f)a + c b + d c + e d + f e} \\
\text{Type: NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

\[
p_6 := a + b + c + d + e + f \\
\text{a + b + c + d + e + f} \\
\text{Type: NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

\[
lp := [p_1, p_2, p_3, p_4, p_5, p_6] \\
[f \cdot d \cdot e \cdot c \cdot b \cdot a - 1, ((e + f)d + f e)c + f e d c + f e d c b, \\
(((d + f)c + f e)b + f e d a + e d c b + f e d c, \\
(((c + f)b + f e)a + d c b + e d c + f e d, \\
(b + f)a + c b + d c + e d + f e, a + b + c + d + e + f) \\
\text{Type: List NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])}
\]

Now call LEXTRIPK.

\[
lextripack := LEXTRIPK(R, ls) \\
\text{LexTriangularPackage(Integer, [a,b,c,d,e,f])} \\
\text{Type: Domain}
\]
CHAPTER 13. CHAPTER L

Compute the lexicographical Groebner basis of the system. This may take between 5 minutes and one hour, depending on your machine.

\[
\begin{align*}
lg := \text{groebner(lp) \text{ lextripack}} \\
[a + b + c + d + e + f, \\
\quad 2] \\
3968379498283200b + 15873517993132800f b + 3968379498283200d + 15873517993132800f d + 3968379498283200f e - 15873517993132800f e + \\
536 6 \quad 2 \\
23810276989699200f e + (206355733910726400f + 230166010900425600)e + 43 \quad 37 \\
- 729705987316687f + 1863667496867205421f + 31 \quad 25 \\
291674853771731104461f + 365285994691106921745f + 19 \quad 13 \\
549961185828911895f - 365048404038768439269f + 7 \\
- 292382820431504027669f - 2271898467631865497f * e + 44 \quad 38 \\
- 3988812642545399f + 10187423878429609997f + 32 \quad 26 \\
1594377523424314053637f + 1994739308439916238065f + 20 \quad 14 \\
1596840088052642815f - 1993494118301162145413f + 8 \quad 2 \\
- 1596049742289689815053f - 11488171330159667449f , \\
2 \\
(23810276989699200c - 23810276989699200f)b + 23810276989699200c + 2 \\
71430830969097600f c - 23810276989699200d - 95241107958796800f d + 35 \quad 44 \\
- 55557312975964800f e + 174608697924460800f e - 174608697924460800f e + \\
\end{align*}
\]
\begin{align*}
\text{6} & \quad \text{2} \\
\quad & \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quarten
\[ + 44 \quad 38 \quad 32 \\
6423170513956901f - 1640477213703648083f - 2567419165227528774463f \\
+ 26 \quad 20 \\
- 32119380825662172335f - 233049032697587485f \\
+ 14 \quad 8 \\
3210100109444754864587f + 2569858315395162617847f \\
+ 18326089487427735751f \\
, 3 \quad 5 \\
\left(1190513849489600e - 1190513849489600f\right)b - 3968379498233200f e \\
+ 44 \quad 5 \quad 3 \\
15873517993132800f e - 27778656487982400f e \\
+ 18326089487427735751f 31.25 \\
- 31418806290873807090f - 393423667537929575250f \\
+ 19 \quad 13 \\
- 55032912065439495f + 393196408728889612770f \\
+ 31 \quad 7 \\
314892372799176495730f + 2409386515146668530f \\
* e \\
+ 44 \quad 38 \quad 32 \\
4177638546747827f - 10669685294602576381f - 1669852980419949524601f \\
+ 26 \quad 20 \\
- 2089077057287904170745f - 1569899763580278796f \\
+ 14 \quad 8 \\
2087864026859015573349f + 1671496085945199577969f \\
+ 2 \\
11940257226216280177f \\
, 6 \quad 2 \quad 5 \\
\left(1190513849489600f - 1190513849489600\right)b - 15873517993132800f e \]
\[\begin{align*}
&+ 3^4 4^3 3^9 8^3 9^6 2^8 3^{20} 0^0 0^0 f^e - 3^9 8^3 7^9 4^9 8^3 2^8 3^{20} 0^0 0^0 f^e \\
&+ 11^5 2^2 (-6^8 5^2 6^5 5^3 2^0 2^9 9^8 3^6 0^0 f^e - 6^0 7^1 6^2 2^0 6^3 2^3 7^3 2^9 6^0 0^0 f^e) \\
&+ 42^3 6^6 5^1 4^5 3^1 3^9 6^7 0^4 f^e - 1^6 6^3 8^1 2^8 0^9 0^1 0^8 8^6 6^5 2^7 f^e - 2^6 0^3 3^4 4^5 0^2 4^7 0^2 2^8 3^4 7^2 f^e \\
&+ 2^4 5^2 4^2 9^8 6^5 2^9 8^3 6^5 0^1 4^0 f^e + 9^7 1^4 9^2 0^3 9^3 1^6 7^5 6^8 1^3 6^0 f^e \\
&+ 1^2^1 1^2 6^5 8^6 2^0 0^4 8^5 5^0 0^3 6^3 9^1 3^8 9^8 5^9 4^8 f^e + 2^5 5^2 6^1 7^7 7^6 6^6 0^7 0^5 2^9 8^0 8^0 f^e + 1^3 8^6 0^3 2^6 8^3 5^5 6^7 4^9 2^4 4^4 \\
&\times f^e \\
&+ 4^3 3^7 3^1 1^6 7^6 2^0 0^3 6^7 4^1 8^1 7^9 1^4 7^1 3^7 f^e - 4^2 8^1 0^2 4^1 7^9 7^9 1^4 7^3 f^e - 6^6 9^9 7^2 4^3 8^0 1^2 2^3 1^6 7^9 3^1 f^e \\
&+ 2^5 5^2 3^6 7^7 8^3 6^1 7^2 1^4 8^7 5^0 4^8 5^8 f^e + 2^0 3^6 7^3 8^9 5^3 6^9 9^8 0^7 6^5 f^e \\
&+ 1^3 8^3 5^2 3^0 6^5 3^2 6^0 1^0 4^3 2^4 8^5 7^7 f^e + 6^6 9^9 5^7 8^9 6^4 0^2 3^8 0^6 6^9 9^3 f^e + 4^7 8^5 9^2 8^5 5^5 4^9 5^8 7^9 0^1 f^e, \\
&\times 3^2 2^2 2^4 5^1 4^5 f^e + 2^4 0^5 0^7 8^4 8^3 8^8 0^3 8^0 8^0 f^e c - 2^4 0^5 0^7 8^4 8^3 8^8 0^3 8^0 8^0 f^e c - 1^3 7^5 2^9 4^5 4^6 f^e \\
&+ 3^9 3^3 3^2 7^7 f^e + 5^4 9^6 9^4 6^9 5^7 8^2 6^4 3^3 f^e + 6^8 3^4 6^5 9^4 4^7 7^4 9^1 1^7 f^e \\
&+ 2^1 2^1 1^5 5^5 6^0 f^e - 4^4 4^8 4^8 8^0 4^6 2^4 6^1 f^e - 6^8 7^3 4^0 6^2 3^0 0^9 3^0 5^7 f^e - 5^4 5^0 8^4 4^9 3^8 7^6 2^6 3^3 f^e \\
&+ 3^1 2^1 6^5 8^6 0^4 4^5 7^1 f^e, \\
&(2^3 8^1 0^2 7^6 9^8 9^6 9^9 2^0 0^0 d - 2^3 8^1 0^2 7^6 9^8 9^6 9^9 2^0 0^0 f^e) c + 2^3 8^1 0^2 7^6 9^8 9^6 9^9 2^0 0^0 d \\
&+ 3^5 5^4 4^4 7^1 4^3 0^8 3^0 9^6 9^0 9^7 6^0 0^0 f^e d + 7^9 3^6 7^5 8^8 9^9 6^5 6^6 4^0 0^0 f^e e - 3^1 7^4 0^7 3^5 9^8 6^2 6^5 6^0 0^0 f^e \\
&+ 5^3 6^2 3^1 7^4 0^7 3^5 9^8 6^2 6^5 6^0 0^0 f^e + (4^0 4^7 7^4 0^7 8^8 8^2 4^8 8^8 6^4 0^0 f^e + 3^9 6^8 3^7 9^4 9^8 2^8 3^2 0^0 0^0) e \\
&+ 4^3 3^7 2^2 4^7 3^9 4^6 7^0 0^1 f^e + 3^1 8^5 7^8 5^5 4^9 6^6 2^1 2^0 3^f^e
\end{align*}\]
\begin{align*}
&+ 31 \quad \quad \quad \quad 25 \\
&498594866849974751463f + 62454254584579104793f \\
&+ 19 \quad \quad \quad \quad 13 \\
&9310857557696828885f - 624150663582417063387f \\
&+ 7 \\
&- 49988185938360475647f - 392688531381952351f \\
&\times e \\
&+ 44 \quad \quad \quad \quad 38 \\
&- 7026011547118141f + 1794442705195069123f \\
&+ 32 \quad \quad \quad \quad 26 \\
&280838352259398660354f + 3513624142354807530135f \\
&+ 20 \quad \quad \quad \quad 14 \\
&2860750706705637685f - 3511356736564219073267f \\
&+ 8 \quad \quad \quad \quad 2 \\
&- 281132494697103819887f - 2031501163152284731f \\
&\times (7936758996566400e - 7936758996566400f)c \\
&+ 43 \quad \quad \quad \quad 37 \quad \quad \quad \quad 31 \\
&- 4418748183673f + 112855687074566559f + 1765998617294451019f \\
&+ 25 \quad \quad \quad \quad 19 \\
&2173749283622606155f - 5578292195402995f \\
&+ 13 \quad \quad \quad \quad 7 \\
&- 221529142178292951f - 17181426665347430851f + 30256569458230237f \\
&\times e \\
&+ 44 \quad \quad \quad \quad 38 \quad \quad \quad \quad 32 \\
&4418748183673f - 112855687074566559f - 1765998617294451019f \\
&+ \quad 26 \quad \quad \quad \quad 20 \quad \quad \quad \quad 14 \\
&- 2173749283622606155f + 5578292195402995f + 221529142178292951f \\
&+ 8 \quad \quad \quad \quad 2 \\
&17181426665347430851f - 30256569458230237f \\
&\times (72152354514240f - 72152354514240c) + 40950859449f \\
&+ \quad 37 \quad \quad \quad \quad 31 \quad \quad \quad \quad 25
\end{align*}
\[ + \quad 31 \quad 25 \\
- 196883094539368513959f - 246562115745735428055f \\
+ \quad 19 \quad 13 \\
- 325698701993885505f + 246417769883651808111f \\
+ \quad 7 \\
19732735206820065291f + 1523373796389332143f \\
* e \\
+ \quad 44 \quad 38 \quad 32 \\
2679481081803026f - 6843392695421906608f - 107102045964264913578f \\
+ \quad 26 \quad 20 \\
- 1339789169692041240060f - 852746750910750210f \\
+ \quad 14 \quad 8 \\
1339105101971878401312f + 1071900289758712984762f \\
+ \quad 2 \\
75552390720727756f \\
, \quad 6 \quad 2 \quad 5 \\
(11905138494849600f - 11905138494849600)d - 7936758996566400f e \\
+ \quad 3 \quad 4 \quad 3 \\
31747035986266600f e - 31747035986266600f e \\
+ \quad 11 \quad 5 \quad 2 \\
(- 420649226818019200f - 404774708824886400f )e \\
+ \quad 42 \quad 36 \quad 30 \\
15336187600889f - 39169739565161107f - 6127176127489690827f \\
+ \quad 24 \quad 18 \\
- 7217708742310509615f + 538628483890722736f \\
+ \quad 12 \quad 6 \\
7508604353843507643f + 5886160769782607203f + 63576108396535879 \\
* e \\
+ \quad 43 \quad 37 \quad 31 \\
71737781770666f - 18321886207557938f - 28672874271132276078f \\
+ \quad 25 \quad 19 \\
- 35625223686939812010f + 164831339634084390f \\
+ \]
\[
\begin{align*}
13 & \quad 7 \\
35724160423073052642f & + 28627022578664910622f & + 187459987029680506f \\
6 & \quad 5 & \quad 24 \\
1322793166094400e & - 3968379498238200f \ e & + 3968379498283200f \ e \\
+ & \quad 33 \\
- 5291172664377600e & \ e & + \\
10 & \quad 42 \\
(- 230166010900425600f & - 226197631402142400f \ e) & + \\
47 & \quad 41 \\
- 152375364610443885f & + 3891666626064854890415f & + \\
35 & \quad 29 \\
60906097841360558987335f & + 76167367934608798697275f & + \\
23 & \quad 17 \\
27855066785995181125f & - 7614952817052723145495f & + \\
11 & \quad 5 \\
- 60933629892463517546975f & - 411415071682002547795f & \ e \\
+ & \quad e \\
42 & \quad 36 & \quad 30 \\
- 209493533143822f & + 535045979490560586f & + 8373794796497353146f & + \\
24 & \quad 18 \\
104889507084213371570f & + 167117997269207870f & + \\
12 & \quad 6 \\
- 104793725781390615514f & - 83842685189903180394f & - 569978796672974242f & + \\
+ & \quad 3 \\
(25438330117200f & + 25438330117200f \ e) & + \\
7 & \quad 2 \\
(76314990351600f & + 76314990351600f \ e) & + \\
44 & \quad 38 & \quad 32 \\
- 1594966552735f & + 4073543370415745f & + 637527159231148925f & + \\
26 & \quad 20 & \quad 14 \\
797521176113606525f & + 530440941097175f & - 797160527306433145f & + \\
+ & \quad 8 & \quad 2 \\
- 638132320196044965f & - 4510507167940725f & \ e
\end{align*}
\]
Type: List NewSparseMultivariatePolynomial(Integer, OrderedVariableList [a,b,c,d,e,f])

Apply lexTriangular to compute a decomposition into regular triangular sets. This should not take more than 5 seconds.

```
lexTriangular(lg, false) \$lextripack
[
   6   6   5   2   4   3   3   4   2   5
   f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,
   2   5   3   4   4   3   5   2
   3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
   2   5   3   4   4   3   5   2
   3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f,
   12   2
   1387545279120 f - 1387545279120 e

   43   37   31
   4321823003 f - 11037922310209 f - 1727510711947989 f
   +
   25   19   13
   - 12090487758628245 f - 8787672733575 f + 12083693383005045 f
   +
   7
   1732620732685741 f + 13506088516033 f
   *
   e

   44   38   32
   24177661775 f - 61749727185325 f - 9664106795754225 f
   +
   26   20   14
   - 12090487758628245 f - 8787672733575 f + 12083693383005045 f
   +
   8
   9672870290826025 f + 68544102808525 f
   ,
   48   42   36   30   18   12   6
   f - 2554 f - 399710 f - 499722 f + 499722 f + 399710 f + 2554 f - 1]
```
\[
\begin{align*}
2 & \quad 5 & \quad 3 & \quad 4 & \quad 4 & \quad 3 & \quad 5 & \quad 2 \\
\{f - 1, e - f, d - f, c + 4f, c + f, (c - f)b + f, c - 5f, a + b + c + 3f\}, \\
6 & \quad 2 & \quad 2 & \quad 2 \\
\{f - 1, e - f, d - f, c - f, b + 4f, b + f, a + b + 4f\}, \\
6 & \quad 2 & \quad 2 & \quad 2 \\
\{f - 1, e - f, d + 4f, d + f, (d - f)c - f, d - 5f, b - f, a + c + d + 3f\}, \\
36 & \quad 30 & \quad 24 & \quad 18 & \quad 12 & \quad 6 \\
\{f - 2554f^2 - 399709f^3 - 502276f^4 - 399709f^5 - 2554f^6 + 1, \\
12 & \quad 2 \\
(161718564f^6 - 161718564)\}e \\
+ 31 & \quad 25 & \quad 19 & \quad 13 \\
- 504205f + 1287737951f + 201539391380f + 253982817368f + 7 \\
201940704665f + 1574134601f * e + 32 & \quad 26 & \quad 20 & \quad 14 \\
- 2818405f + 7198203911f + 1126548149060f + 141653056364f + 8 \\
1127377589345f + 7988820725f + 6 & \quad 2 & \quad 5 & \quad 3 & \quad 4 \\
(693772639560f - 693772639560)d - 462515093040f e + 1850060372160f e + 4 & \quad 3 & \quad 11 & \quad 5 & \quad 2 \\
- 1850060372160f e + (- 24513299931120f - 23588269745040f )e + 30 & \quad 24 & \quad 18 \\
- 890810428f + 2275181044754f + 355937263869776f + 12 & \quad 6 \\
413736880104344f + 342849304487996f + 3704966481878 * e + 31 & \quad 25 & \quad 19 \\
- 4163798003f + 10634395752169f + 1664161760192806f + 13 & \quad 7 \\
2079424391370694f + 1668153650635921f + 10924274392693f + 6 & \quad 31 & \quad 25 \\
(12614047992f - 12614047992)c - 7246825f + 18508536599f
\end{align*}
\]
\[ + 19 \quad 13 \quad 7 \\
2896249516034f + 3581539649666f + 2796477571739f - 48094301893f \\
, 6 \quad 2 \quad 5 \quad 3 \quad 4 \\
(693772639560f - 693772639560)b - 925030186080f e + 2312575465200f e \\
+ 4 \quad 3 \quad 11 \quad 5 \quad 2 \\
- 2312575465200f e + (- 40007555547960f - 35382404617560f )e \\
+ 30 \quad 24 \quad 18 \\
- 3781280823f + 9657492291789f + 1511158913397906f \\
+ 12 \quad 6 \\
1837290892286154f + 1487216006594361f + 8077238712093 \\
* e \\
+ 31 \quad 25 \quad 19 \\
- 9736390478f + 24866827916734f + 3891495681905296f \\
+ 13 \quad 7 \\
4872556418871424f + 3904047887269606f + 27890075838538f \\
, a + b + c + d + e + f \\
, 6 \quad 2 \quad 2 \quad 2 \\
\{ f - 1, e + 4f e + f, (e - f) d - f e - 5f, c - f, b - f, a + d + e + 3f \} \]

Type: List RegularChain(Integer, [a, b, c, d, e, f])

Note that the first set of the decomposition is normalized (all initials are integer numbers) but not the second one (normalized triangular sets are defined in the description of the NormalizedTriangularSetCategory constructor).

So apply now lexTriangular to produce normalized triangular sets.

\[ lts := lexTriangular(lg, true)$lextripack \]

\[
\begin{align*}
\{ f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1, & \\
3d + f e - 4f e + 4f e - 2f e - 2f e + 2f, c + f, & \\
3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f, & \\
a - f e + 3f e - 3f e + 4f e + 3e - 3f & \\
\}
\]

\[
\begin{align*}
\{ f - 1, e - f, d - f, c + 4f c + f, b + c + 4f, a - f, & \\
\}
\]
\{f - 1, e - f, d - f, c - f, b + 4f, b + f, a + b + 4f, \}
\{f - 1, e - f, d + 4f, d + f, c + d + 4f, b - f, a - f, \}
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \}
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \}
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \}
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \}
\{f - 2554f - 399709f - 502276f - 399709f - 2554f + 1, \}
We check that all initials are constant.

We check that all initials are constant.

Note that each triangular set in lts is a lexicographical Groebner basis. Recall that a point belongs to the variety associated with lp if and only if it belongs to that associated with one triangular set ts in lts.

By running the squareFreeLexTriangular operation, we retrieve the above decomposition.

```
squareFreeLexTriangular(lg,true)$lextripack
```

```
[6 6 5 2 4 3 3 4 2 5]
```

```
{f + 1, e - 3f e + 3f e - 4f e + 3f e - 3f e - 1,
 2 5 3 4 4 3 5 2
3d + f e - 4f e + 4f e - 2f e - 2e + 2f, c + f,
 2 5 3 4 4 3 5 2
3b + 2f e - 5f e + 5f e - 10f e - 4e + 7f,
 2 5 3 4 4 3 5 2
}
\[a - f e + 3f e - 3f e + 4f e + 3e - 3f\]
\[\{f - 1, e - f, d - f, c + 4f, c + f, b + c + 4f, a - f\},\]
\[\{f - 1, e - f, d - f, c - f, b + 4f, b + f, a + b + 4f\},\]
\[\{f - 1, e - f, d + 4f, d + f, c + d + 4f, b - f, a - f\},\]
\[\{f - 255f - 399709f - 502276f - 399709f - 2554f + 1,\]
\[1387545279120e + \]
\[4321823003f - 11037922310209f - 1727506390124986f + \]
\[- 2176188913464634f - 1732620732685741f - 13506088516033f * e + \]
\[24177661775f - 61749727185325f - 9664082618092450f + \]
\[- 12152237485813570f - 9672870290826025f - 68544102808525f ,\]
\[1387545279120d + \]
\[- 1128983050f + 2883434331830f + 451234998755840f + \]
\[562426491685760f + 447129055314890f - 165557857270 * e + \]
\[- 1816935351f + 4640452214013f + 726247129626942f + \]
\[912871801716798f + 726583262666877f + 4909358645961f ,\]
\[1387545279120c + 778171189f - 1987468196267f - 310993556954378f + \]
\[- 383262822316802f - 300335488637543f + 5289595037041f ,\]
\[1387545279120b\]
Thus the solutions given by \texttt{lts} are pairwise different.

We count them as follows.

\[
reduce(+,[\text{degree(ts)} \text{ for ts in } \texttt{lts}])
\]

156

\text{Type: PositiveInteger}

We can investigate the triangular decomposition \texttt{lts} by using the \texttt{ZeroDimensionalSolvePackage}.

This requires to add an extra variable (smaller than the others) as follows.

\texttt{ls2 : List Symbol := concat(ls,new()$Symbol)}

\text{[a,b,c,d,e,f,%A]}

\text{Type: List Symbol}

Then we call the package.
zdpack := ZDSOLVE(R,ls,ls2)
(20) ZeroDimensionalSolvePackage(Integer,[a,b,c,d,e,f],[a,b,c,d,e,f,%A])
Type: Domain

We compute a univariate representation of the variety associated with the input system as follows.

concat [univariateSolve(ts)$zdpack for ts in lts]
[
  4 2
  [complexRoots= ? - 13? + 49,
     coordinates =
     3 3 3 3
     [7a + %A - 6%A, 21b + %A + %A, 21c - 2%A + 19%A, 7d - %A + 6%A,
      3 3
      21e - %A - %A, 21f + 2%A - 19%A]
  ,
  4 2
  [complexRoots= ? + 11? + 49,
     coordinates =
     3 3 3 3
     [35a + 3%A + 19%A, 35b + %A + 18%A, 35c - 2%A - %A,
      3 3
      35d - 3%A - 19%A, 35e - %A - 18%A, 35f + 2%A + %A]
  ],
  [complexRoots =
    8 7 6 5 4 3 2
    ,
    coordinates =
    7 6 5 4
    43054532a + 33782%A - 546673%A + 3127348%A - 6927123%A +
    3 2
    4365212%A - 25086957%A + 39582814%A - 107313172
    ,
    7 6 5 4
    43054532b - 33782%A + 546673%A - 3127348%A + 6927123%A +
    3 2
    - 4365212%A + 25086957%A - 39582814%A + 107313172
    ,
    7 6 5 4
    21527266c - 22306%A + 263139%A - 1166076%A + 1821805%A +
\[\begin{align*}
3 & - 289278\% A + 10322663\% A - 9026596\% A + 12950740 \\
7 & - 6 & 5 & 4 & 43054532d + 22306\% A - 263139\% A + 1166076\% A - 1821805\% A \\
3 & 2 & 289278\% A - 10322663\% A + 30553862\% A - 12950740 \\
7 & - 6 & 5 & 4 & 43054532e - 22306\% A + 263139\% A - 1166076\% A + 1821805\% A \\
3 & 2 & - 289278\% A + 10322663\% A - 30553862\% A + 12950740 \\
7 & - 6 & 5 & 4 & 21527266f + 22306\% A - 263139\% A + 1166076\% A - 1821805\% A \\
3 & 2 & 289278\% A - 10322663\% A + 9026596\% A - 12950740 \\
\end{align*}\]

```
complexRoots =
\[\begin{align*}
coordinates =
\[\begin{align*}
43054532a & + 33782\% A + 546673\% A + 3127348\% A + 6927123\% A \\
3 & 2 & 4365212\% A & + 25086957\% A & + 39582814\% A & + 107313172 \\
43054532b & - 33782\% A & - 546673\% A & - 3127348\% A & - 6927123\% A \\
3 & 2 & - 4365212\% A & - 25086957\% A & - 39582814\% A & - 107313172 \\
21527266c & - 22306\% A & - 263139\% A & - 1166076\% A & - 1821805\% A \\
3 & 2 & - 289278\% A & - 10322663\% A & - 9026596\% A & - 12950740 \\
43054532d & + 22306\% A & + 263139\% A & + 1166076\% A & + 1821805\% A
\end{align*}\]
```
\[ + \begin{array}{cccc}
3 & 2 & 2892788\%A & + 10322663\%A & + 30553862\%A & + 12950740 \\
7 & 6 & 5 & 4 & 43054532e & - 22306\%A & - 263139\%A & - 1166076\%A & - 1821805\%A \\
+ & 3 & 2 & 2892788\%A & - 10322663\%A & - 30553862\%A & - 12950740 \\
7 & 6 & 5 & 4 & 21527266f & + 22306\%A & + 263139\%A & + 1166076\%A & + 1821805\%A \\
+ & 3 & 2 & 2892788\%A & + 10322663\%A & + 9026596\%A & + 12950740 \\
\end{array}\]

\[
\text{complexRoots} = ? - ? + 1,
\]

\[
\text{coordinates} = \begin{bmatrix}
\text{a} - \%A, \text{b} + \%A - \%A, \text{c} + \%A, \text{d} + \%A, \text{e} - \%A + \%A, \text{f} - \%A
\end{bmatrix}
\]

\[
\begin{array}{cccc}
4 & 2 & \text{complexRoots} = \text{?} + 4? + 12? + 16? + 4, \\
\end{array}
\]

\[
\text{coordinates} = \begin{bmatrix}
4\text{a} - 2\%A - 7\%A - 20\%A - 22\%A, 4\text{b} + 2\%A + 7\%A + 20\%A + 22\%A, \\
4\text{c} + 3\%A + 10\%A + 10\%A, 4\text{d} + 3\%A + 10\%A + 6\%A, \\
4\text{e} - 3\%A - 10\%A - 6\%A, 4\text{f} - 3\%A - 10\%A - 10\%a
\end{bmatrix}
\]

\[
\begin{array}{cccc}
4 & 3 & 2 & \text{complexRoots} = \text{?} + 6? + 30? + 36? + 36, \\
\end{array}
\]

\[
\text{coordinates} = \begin{bmatrix}
30\text{a} - \%A - 5\%A - 30\%a - 6, 6\text{b} + \%A + 5\%a + 24\%a + 6, \\
30\text{c} - \%a - 5\%A - 6, 30\text{d} - \%A - 5\%A - 30\%A - 6, \\
30\text{e} - \%a - 5\%A - 30\%a - 6, 30\text{f} - \%A - 5\%a - 30\%a - 6
\end{bmatrix}
\]

\[
\begin{array}{cccc}
4 & 3 & 2 & \text{complexRoots} = \text{?} - 6? + 30? - 36? + 36, \\
\end{array}
\]

\[
\text{coordinates} = \begin{bmatrix}
30\text{a} - \%a + 5\%A - 30\%a + 6, 6\text{b} + \%A - 5\%a - 24\%a - 6,
\end{bmatrix}
\]
3 \cdot 2
30c - \%A + 5\%A + 6, 30d - \%A + 5\%A - 30\%A + 6,
3 \cdot 2
30e - \%A + 5\%A - 30\%A + 6, 30f - \%A + 5\%A - 30\%A + 6]

2
[complexRoots= \frac{a}{A} + 6, \frac{b}{A} + 6,
coordinates= [a + 1, b - \%A - 5, c + \%A + 1, d + 1, e + 1, f + 1]]

2
[complexRoots= \frac{a}{A} - 6, \frac{b}{A} + 6,
coordinates= [a - 1, b - \%A + 5, c + \%A - 1, d - 1, e - 1, f - 1]]

4 \cdot 3 \cdot 2
[complexRoots= \frac{a}{A} + 6\%A + 30\%A + 36\%A + 36,
coordinates =
6a + \%A + 5\%A + 24\%A + 6, 30b - \%A - 5\%A - 6,
3 \cdot 2
30c - \%A - 5\%A - 30\%A - 6, 30d - \%A - 5\%A - 30\%A - 6,
3 \cdot 2

4 \cdot 3 \cdot 2
[complexRoots= \frac{a}{A} - 6\%A + 30\%A - 36\%A + 36,
coordinates =
6a + \%A - 5\%A + 24\%A - 6, 30b - \%A + 5\%A + 6,
3 \cdot 2
30c - \%A + 5\%A - 30\%A + 6, 30d - \%A + 5\%A - 30\%A + 6,
3 \cdot 2
30e - \%A + 5\%A - 30\%A + 6, 30f - \%A + 5\%A - 30\%A + 6]
\[
6c + \%A + 5\%A + 24\%A + 6, 30d - \%A - 5\%A - 6, \\
3 2 3 2
\]
\[
\]
\[
\]
\[
\text{coordinates} = \\
\]
\[
\text{complexRoots} = ? - 6? + 30? - 36? + 36, \\
\text{coordinates} = \\
3 2 3 2
\]
\[
30a - \%A + 5\%A - 30\%A + 6, 30b - \%A + 5\%A - 30\%A + 6, \\
3 2 3 2
\]
\[
6c + \%A - 5\%A + 24\%A - 6, 30d - \%A + 5\%A + 6, \\
3 2 3 2
\]
\[
30e - \%A + 5\%A - 30\%A + 6, 30f - \%A + 5\%A - 30\%A + 6]
\]
\[
\]
\[
\text{coordinates} = \\
\]
\[
\text{complexRoots} = ? + 6? + 6, \\
\text{coordinates} = [a + 1,b + 1,c - \%A - 5,d + \%A + 1,e + 1,f + 1]
\]
\[
\]
\[
\text{complexRoots} = ? - 6? + 6, \\
\text{coordinates} = [a - 1,b - 1,c - \%A + 5,d + \%A - 1,e - 1,f - 1]
\]
\[
\]
\[
\text{coordinates} = \\
7 6 5 4 3 2
\]
\[
2a + 2\%A + 9\%A + 18\%A + 19\%A + 4\%A - 10\%A - 2\%A + 4, \\
7 6 5 4 3 2
\]
\[
2b + 2\%A + 9\%A + 18\%A + 19\%A + 4\%A - 10\%A - 4\%A + 4, \\
7 6 5 4 3
\]
\[
2c - \%A - 4\%A - 8\%A - 9\%A - 4\%A - 2\%A - 4, \\
7 6 5 4 3
\]
\[
2d + \%A + 4\%A + 8\%A + 9\%A + 4\%A + 2\%A + 4, \\
7 6 5 4 3
\]
\[
2e - 2\%A - 9\%A - 18\%A - 19\%A - 4\%A + 10\%A + 4\%A - 4, \\
7 6 5 4 3
\]
\[
\]
\[
\]
\[
\text{complexRoots} = \\
\]
\[
\text{coordinates} = \\
7 6 5 4 3 2
\]
\[
1408a - 19\%A - 200\%A - 912\%A - 2216\%A - 4544\%A - 6784\%A
\]
\[
\begin{align*}
1408b & \quad - 37A - 408A - 1952A - 5024A - 10368A - 16768A \\
17920A & \quad - 5120 \\
1408c & \quad + 37A + 408A + 1952A + 5024A + 10368A + 16768A \\
17920A & \quad + 5120 \\
1408d & \quad + 19A + 200A + 912A + 2216A + 4544A + 6784A \\
6976A & \quad + 1792 \\
2e & \quad + %A, 2f - %A
\end{align*}
\]

\[
\begin{align*}
\text{coordinates} &= 7 5 3 7 5 3 \\
4a - %A - 3A - 10A - 6A, & 4b - %A - 3A - 10A - 10A, \\
4c - 2A - 7A - 20A - 22A, & 4d + 2A + 7A + 20A + 22A, \\
4e + %A + 3A + 10A + 10A, & 4f + %A + 3A + 10A + 6A
\end{align*}
\]

\[
\begin{align*}
\text{coordinates} &= 7 5 3 \\
512a - %A - 12A + 176A - 448A, & 128b - %A - 16A + 96A - 256A, \\
128c + %A + 16A - 96A + 256A, & 512d + %A + 12A - 176A + 448A, 2e + %A, 2f - %A
\end{align*}
\]

\[
\begin{align*}
\text{coordinates} &= 8 7 6 5 4 3 2 \\
? - 12A & \quad + 64A - 192A + 432A - 768A + 1024A - 768A + 256
\end{align*}
\]
coordinates =
[
    7  6  5  4  3  2
1408a – 19%A + 200%A - 912%A + 2216%A - 4544%A + 6784%A
+ - 6976%A + 1792
, 
7  6  5  4  3  2
1408b – 37%A + 408%A - 1952%A + 5024%A - 10368%A + 16768%A
+ - 17920%A + 5120
, 
7  6  5  4  3  2
1408c + 37%A - 408%A + 1952%A - 5024%A + 10368%A - 16768%A
+ 17920%A - 5120
, 
7  6  5  4  3  2
1408d + 19%A - 200%A + 912%A - 2216%A + 4544%A - 6784%A
+ 6976%A - 1792
, 
2e + %A, 2f - %A]
]

coordinates =
7  6  5  4  3  2
[2a + 2%A - 9%A + 18%A - 19%A + 4%A + 10%A - 2%A - 4,
7  6  5  4  3  2
2b + 2%A - 9%A + 18%A - 19%A + 4%A + 10%A - 4%A - 4,
7  6  5  4  3  2
2c - %A + 4%A - 8%A + 9%A - 4%A - 2%A + 4,
7  6  5  4  3  2
2d + %A - 4%A + 8%A - 9%A + 4%A + 2%A - 4,
7  6  5  4  3  2
2e - 2%A + 9%A - 18%A + 19%A - 4%A - 10%A + 4%A + 4,
7  6  5  4  3  2
2f - 2%A + 9%A - 18%A + 19%A - 4%A - 10%A + 2%A + 4]
]

[complexRoots= ? + 12? + 144,
coordinates =
2  2  2  2
[12a - %A - 12, 12b - %A - 12, 12c - %A - 12, 12d - %A - 12,
2  2
6e + %A + 3%A + 12, 6f + %A - 3%A + 12]  
]
[complexRoots= \( \pm 6\gamma^2 \pm 30\gamma \pm 36 \),
coordinates =
\[
\begin{align*}
  & 3 & 2 & 0 & 0 & 0 & 0 \\
  6a & - \%A & - 5\%A & - 24\%A & - 6, & 30b & + \%A & + 5\%A & + 30\%A & + 6, \\
  3 & 2 & 3 & 2 & 3 & 0 & 0 & 0 & 0 & 0 \\
  30c & + \%A & + 5\%A & + 30\%A & + 6, & 30d & + \%A & + 5\%A & + 30\%A & + 6, \\
  3 & 2 & 3 & 2 & 3 & 0 & 0 & 0 & 0 & 0 \\
  30e & + \%A & + 5\%A & + 30\%A & + 6, & 30f & + \%A & + 5\%A & + 6 \\
\end{align*}
\]
]

[complexRoots= \( \pm 6\gamma^2 \pm 30\gamma \pm 36 \),
coordinates =
\[
\begin{align*}
  & 3 & 2 & 0 & 0 & 0 & 0 \\
  3 & 2 & 3 & 2 & 3 & 0 & 0 & 0 & 0 & 0 \\
  30c & + \%A & - 5\%A & + 30\%A & - 6, & 30d & - \%A & - 5\%A & + 30\%A & - 6, \\
  3 & 2 & 3 & 2 & 3 & 0 & 0 & 0 & 0 & 0 \\
  30e & + \%A & - 5\%A & + 30\%A & - 6, & 30f & - \%A & - 5\%A & + 6 \\
\end{align*}
\]
]

[complexRoots= \( \pm 12\gamma^2 \pm 144 \),
coordinates =
\[
\begin{align*}
  & 2 & 2 & 2 & 2 & 2 \\
  12a & + \%A & + 12, & 12b & + \%A & + 12, & 12c & + \%A & + 12, & 12d & + \%A & + 12, \\
  2 & 2 & 2 & 2 & 2 & 2 \\
  6e & - \%A & + 3\%A & - 12, & 6f & - \%A & - 3\%A & - 12 \\
\end{align*}
\]
]

[complexRoots= \( \pm 12 \),
coordinates= [a - 1,b - 1,c - 1,d - 1,e - 1,f - \%A - 3]]}

[complexRoots= \( \pm 6\gamma^2 \pm 6 \),
coordinates= [a + \%A + 5,b - 1,c - 1,d - 1,e - 1,f - \%A - 1]]}

[complexRoots= \( \pm 6\gamma^2 \pm 6 \),
coordinates= [a + \%A - 5,b + 1,c + 1,d + 1,e + 1,f - \%A + 1]]}

[complexRoots= \( \pm 12 \),
coordinates= [a + 1,b + 1,c + 1,d + 1,e - 1,f - \%A - 1]]}
coordinates =
   
   \[
   \begin{array}{ccc}
   3 & 2 & 3 & 2 \\
   \text{30a} - \%A - 5\%A - 30\%A - 6, & \text{30b} - \%A - 5\%A - 30\%A - 6, \\
   \text{30c} - \%A - 5\%A - 30\%A - 6, & \text{6d} + \%A + 5\%A + 24\%A + 6, \\
   \text{30e} - \%A - 5\%A - 6, & \text{30f} - \%A - 5\%A - 30\%A - 6 \\
   \end{array}
   \]

[complexRoots= \( ? - 6\%A + 30\%A - 36\%A + 36, \\
   \text{coordinates } =
   
   \begin{array}{ccc}
   3 & 2 & 3 & 2 \\
   \text{30a} - \%A + 5\%A - 30\%A + 6, & \text{30b} - \%A + 5\%A - 30\%A + 6, \\
   \text{30c} - \%A + 5\%A - 30\%A + 6, & \text{6d} + \%A - 5\%A + 24\%A - 6, \\
   \text{30e} - \%A + 5\%A + 6, & \text{30f} - \%A + 5\%A - 30\%A + 6 \\
   \end{array}
   \]

[complexRoots= \( ? + 6\%A + 6, \\
   \text{coordinates } = [\text{a} + 1, \text{b} + 1, \text{c} + 1, \text{d} - \%A - 5, \text{e} + \%A + 1, \text{f} + 1]
   
   \]

[complexRoots= \( ? - 6\%A + 6, \\
   \text{coordinates } = [\text{a} - 1, \text{b} - 1, \text{c} - 1, \text{d} - \%A + 5, \text{e} + \%A - 1, \text{f} - 1]
   
   \]

Type: List Record(complexRoots: SparseUnivariatePolynomial Integer, 
   \text{coordinates } : List Polynomial Integer)

Since the univariateSolve operation may split a regular set, it 
returns a list. This explains the use of concat.

Look at the last item of the result. It consists of two parts. For 
any complex root \( ? \) of the univariate polynomial in the first part, we 
get a tuple of univariate polynomials (in a, ...,f respectively) by 
replacing \( \%A \) by \( ? \) in the second part. Each of these tuples t 
describes a point of the variety associated with lp by equaling to 
zero the polynomials in t.

Note that the way of reading these univariate representations is 
explained also in the example illustrating the ZeroDimensionalSolvePackage 
constructor.

Now, we compute the points of the variety with real coordinates.

concat [realSolve(ts)$zdpack for ts in lts]
   
   \[
   \begin{array}{cccc}
   \text{[\%B1,\%B1,\%B1,\%B5, - \%B5 - 4\%B1,\%B1], [\%B1,\%B1,\%B1,\%B6, - \%B6 - 4\%B1,\%B1],} \\
   \end{array}
   \]
[\%B7,\%B7,\%B7,\%B7,\%B11,- \%B11 - 4\%B7], [\%B7,\%B7,\%B7,\%B7,\%B12,- \%B12 - 4\%B7],
[\%B8,\%B8,\%B8,\%B8,\%B9,- \%B9 - 4\%B8], [\%B8,\%B8,\%B8,\%B8,\%B10,- \%B10 - 4\%B8],
[\%B13,\%B13,\%B17,- \%B17 - 4\%B13,\%B13,\%B13],
[\%B13,\%B13,\%B18,- \%B18 - 4\%B13,\%B13,\%B13],
[\%B14,\%B14,\%B15,- \%B15 - 4\%B14,\%B14,\%B14],
[\%B14,\%B14,\%B16,- \%B16 - 4\%B14,\%B14,\%B14],
[\%B19, \%B29, 
7865521 31 6696179241 25 25769893181 19
- \%B19 - \%B19 - \%B19 6006689520 2002229840 49236160
+ 1975912990729 13 1048460696489 7 21252634831
\%B19 \%B19 \%B19
3003344760 2002229840 6006689520
- 778171189 31 1987468196267 25 15549677847189 19
\%B19 \%B19 \%B19
1387545279120 1387545279120 693772639560
+ 191634111158401 13 300335488637543 7 75566433863
\%B19 \%B19 \%B19
693772639560 1387545279120 198220754160
- 1094352947 31 2794979430821 25 21870802908737 19
\%B19 \%B19 \%B19
462515093040 462515093040 231257546520
+ 91476663003591 13 145152550961823 7 1564893370717
\%B19 \%B19 \%B19
77085848840 154171697680 462515093040
- 4321823003 31 180949646069 25
\%B19 \%B19 \%B19
1387545279120 22746643920
+ 863753195062493 19 1088094456732317 13
\%B19 \%B19 \%B19
693772639560 693772639560
+ 1732620732685741 7 13506088516033
\%B19 \%B19 \%B19
1387545279120 1387545279120
\%B19, \%B30,
7865521 31 6696179241 25 25769893181 19
\%B19 \%B19 \%B19
6006689520 2002229840 49236160
+ 1975912990729 13 1048460696489 7 21252634831
\[
\begin{align*}
\text{PACKET LEXTRIPK LEXTRIANGULARPACKAGE} & \quad 1511 \\
\text{3003344760} & \quad 2002229840 & \quad 6006689520 \\
\text{778171189} & \quad 31 & \quad 1987469196267 & \quad 25 & \quad 155496778477189 & \quad 19 \\
\text{1387545279120} & \quad 1387545279120 & \quad 693772639560 \\
+ & \quad 1916314111538401 & \quad 13 & \quad 30033548637543 & \quad 7 & \quad 755656433863 \\
\text{693772639560} & \quad 1387545279120 & \quad 198220754160 \\
\text{1094352947} & \quad 31 & \quad 2794979430821 & \quad 25 & \quad 218708802908737 & \quad 19 \\
\text{462515093040} & \quad 154171697680 & \quad 462515093040 \\
+ & \quad 9147666303591 & \quad 13 & \quad 145152550961823 & \quad 7 & \quad 1564893370717 \\
\text{77085848840} & \quad 154171697680 & \quad 462515093040 \\
\text{863753195062493} & \quad 19 & \quad 1088094456732317 & \quad 13 \\
\text{693772639560} & \quad 693772639560 \\
+ & \quad 1732627268451741 & \quad 7 & \quad 13506088516033 \\
\text{1387545279120} & \quad 1387545279120 \\
\text{7865521} & \quad 31 & \quad 6669179241 & \quad 25 & \quad 2576993181 & \quad 19 \\
\text{6006689520} & \quad 2002229840 & \quad 49235160 \\
+ & \quad 1975912990729 & \quad 13 & \quad 1048460696489 & \quad 7 & \quad 21252634831 \\
\text{3003344760} & \quad 2002229840 & \quad 6006689520 \\
\text{778171189} & \quad 31 & \quad 1987469196267 & \quad 25 & \quad 155496778477189 & \quad 19 \\
\text{1387545279120} & \quad 1387545279120 & \quad 693772639560 \\
+ & \quad 191631411158401 & \quad 13 & \quad 30033548637543 & \quad 7 & \quad 755656433863 \\
\text{693772639560} & \quad 1387545279120 & \quad 198220754160 \\
\text{1094352947} & \quad 31 & \quad 2794979430821 & \quad 25 & \quad 218708802908737 & \quad 19 \\
\end{align*}
\]
\[
\begin{align*}
\text{CHAPTER 13. \textit{CHAP}\textit{TER} L} \\
\begin{array}{|c|c|c|c|}
\hline
462515093040 & 462515093040 & 231257546520 \\
\hline
91476663003591 & 13 & 145152550961823 & 7 & 1564893370717 \\
- & \text{\%B20} & - & \text{\%B20} & - & \text{\%B20} \\
77085848840 & 154171697680 & 462515093040 \\
\hline
4321823003 & 31 & 180949546069 & 25 \\
- \text{\%B27} & - & \text{\%B20} & + & \text{\%B20} \\
1387545279120 & 22746643920 \\
\hline
863753195062493 & 19 & 1088094456732317 & 13 \\
- & \text{\%B20} & + & \text{\%B20} \\
693772639560 & 693772639560 \\
\hline
1732620732685741 & 7 & 13506088516033 & 19 \\
- & \text{\%B20} & + & \text{\%B20} \\
1387545279120 & 1387545279120 \\
\hline
7865521 & 31 & 6696179241 & 25 & 25769893181 & 19 \\
- & \text{\%B20} & - & \text{\%B20} & - & \text{\%B20} \\
600689520 & 2002229840 & 49235160 \\
\hline
1975912990729 & 13 & 1048460696489 & 7 & 21252634831 \\
- & \text{\%B20} & - & \text{\%B20} & - & \text{\%B20} \\
3003344760 & 2002229840 & 600689520 \\
\hline
778171189 & 31 & 1987468196267 & 25 & 155496778477189 & 19 \\
- & \text{\%B20} & + & \text{\%B20} & + & \text{\%B20} \\
1387545279120 & 1387545279120 & 693772639560 \\
\hline
191631411158401 & 13 & 300335488637543 & 7 & 75566433863 \\
- & \text{\%B20} & + & \text{\%B20} & - & \text{\%B20} \\
693772639560 & 1387545279120 & 198220754160 \\
\hline
1094352947 & 31 & 2794979430821 & 25 & 218708802908737 & 19 \\
- & \text{\%B20} & - & \text{\%B20} & - & \text{\%B20} \\
462515093040 & 462515093040 & 231257546520 \\
\hline
91476663003591 & 13 & 145152550961823 & 7 & 1564893370717 \\
- & \text{\%B20} & - & \text{\%B20} & - & \text{\%B20} \\
77085848840 & 154171697680 & 462515093040 \\
\hline
4321823003 & 31 & 180949546069 & 25 \\
- \text{\%B28} & - & \text{\%B20} & + & \text{\%B20} \\
1387545279120 & 22746643920 \\
\hline
863753195062493 & 19 & 1088094456732317 & 13 \\
\end{array}
\end{align*}
\]
<table>
<thead>
<tr>
<th>Package</th>
<th>LexTriPK</th>
<th>LexTriangularPackage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%B20</td>
<td>+</td>
<td>%B20</td>
</tr>
<tr>
<td>693772639560</td>
<td>693772639560</td>
<td></td>
</tr>
<tr>
<td>%B20</td>
<td>1732620732685741</td>
<td>7 13506088516033</td>
</tr>
<tr>
<td>1387545279120</td>
<td>1387545279120</td>
<td></td>
</tr>
<tr>
<td>%B21, %B25, 7865521</td>
<td>6696179241</td>
<td>25 25769893181</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>6006689520</td>
<td>2002229840</td>
<td>49235160</td>
</tr>
<tr>
<td>%B21, %B25, 19759129990729</td>
<td>13 1048460696489</td>
<td>7 21252634831</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>3003344760</td>
<td>2002229840</td>
<td>6006689520</td>
</tr>
<tr>
<td>%B21, %B25, 19163141158401</td>
<td>13 300335488637543</td>
<td>7 755656433863</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>693772639560</td>
<td>1387545279120</td>
<td>693772639560</td>
</tr>
<tr>
<td>%B21, %B25, 1094352947</td>
<td>31 2794979430821</td>
<td>25 218708802908737</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>462515093040</td>
<td>462515093040</td>
<td>231257546520</td>
</tr>
<tr>
<td>%B21, %B25, 91476663003591</td>
<td>13 145152550961823</td>
<td>7 1564893370717</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>770858488840</td>
<td>154171697680</td>
<td>462515093040</td>
</tr>
<tr>
<td>%B21, %B25, 4321823003</td>
<td>31 180949546069</td>
<td>25</td>
</tr>
<tr>
<td>%B25</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>1387545279120</td>
<td>22746643920</td>
<td></td>
</tr>
<tr>
<td>%B21, %B25, 863753195062493</td>
<td>19 108809456732317</td>
<td>13</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>693772639560</td>
<td>693772639560</td>
<td></td>
</tr>
<tr>
<td>%B21, %B25, 1732620732685741</td>
<td>7 13506088516033</td>
<td></td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>1387545279120</td>
<td>1387545279120</td>
<td></td>
</tr>
<tr>
<td>%B21, %B25, 7865521</td>
<td>6696179241</td>
<td>25 25769893181</td>
</tr>
<tr>
<td>%B21</td>
<td>-</td>
<td>%B21</td>
</tr>
<tr>
<td>6006689520</td>
<td>2002229840</td>
<td>49235160</td>
</tr>
</tbody>
</table>
\[\begin{align*}
\text{1094352947} & \quad 31 \quad 2794979430821 \quad 25 \quad 218708802908737 \quad 19 \\
\text{462515093040} & \quad 462515093040 \quad 231257546520 \\
+ & \quad 91476663003591 \quad 13 \quad 145152550961823 \quad 7 \quad 1564893370717 \\
- & \quad \%B22 \quad - \quad \%B22 \quad - \quad \%B22 \\
\text{77085848840} & \quad 154171697680 \quad 462515093040 \\
\text{13} & \quad 180949546069 \quad 25 \\
- & \quad \%B23 \quad - \quad \%B22 \quad + \quad \%B22 \\
\text{1387545279120} & \quad 22746643920 \\
+ & \quad 863753195062493 \quad 19 \quad 1088094456732137 \\
- & \quad \%B22 \quad + \quad \%B22 \\
\text{693772639560} & \quad 693772639560 \\
+ & \quad 1732620732685741 \quad 7 \quad 13506086516033 \\
- & \quad \%B22 \quad + \quad \%B22 \\
\text{1387545279120} & \quad 1387545279120 \\
\text{7865521} & \quad 31 \quad 6696179241 \quad 25 \quad 25769893181 \quad 19 \\
- & \quad \%B22 \quad - \quad \%B22 \quad - \quad \%B22 \\
\text{6006689520} & \quad 2002229840 \quad 49235160 \\
+ & \quad 1975912990729 \quad 13 \quad 1048460696489 \quad 7 \quad 21252634831 \\
- & \quad \%B22 \quad - \quad \%B22 \quad - \quad \%B22 \\
\text{3003344760} & \quad 2002229840 \quad 6006689520 \\
, & \quad 778171189 \quad 31 \quad 1987468196267 \quad 25 \quad 15549778477189 \\
- & \quad \%B22 \quad + \quad \%B22 \quad + \quad \%B22 \\
\text{1387545279120} & \quad 1387545279120 \quad 693772639560 \\
+ & \quad 191631411158401 \quad 13 \quad 30033548637543 \quad 7 \quad 75566433863 \\
- & \quad \%B22 \quad + \quad \%B22 \quad - \quad \%B22 \\
\text{693772639560} & \quad 1387545279120 \quad 198220754160 \\
, & \quad 1094352947 \quad 31 \quad 2794979430821 \quad 25 \quad 218708802908737 \quad 19 \\
- & \quad \%B22 \quad - \quad \%B22 \quad - \quad \%B22 \\
\text{462515093040} & \quad 462515093040 \quad 231257546520 \\
+ & \quad 91476663003591 \quad 13 \quad 145152550961823 \quad 7 \quad 1564893370717 \\
- & \quad \%B22 \quad - \quad \%B22 \quad - \quad \%B22 \\
\text{77085848840} & \quad 154171697680 \quad 462515093040 \\
, & \quad 4321823003 \quad 31 \quad 180949546069 \quad 25 \\
- & \quad \%B24 \quad - \quad \%B22 \quad + \quad \%B22 \\
\text{1387545279120} & \quad 22746643920 \\
\end{align*}\]
We obtain 24 points given by lists of elements in the RealClosure of Fraction of R. In each list, the first value corresponds to the indeterminate $f$, the second to $e$ and so on.

See Also:
- `help RegularChain`
- `help RegularTriangularSet`
- `help SquareFreeRegularTriangularSet`
- `help ZeroDimensionalSolvePackage`
- `help NewSparseMultivariatePolynomial`
- `help LinGroebnerPackage`
- `help NormalizedTriangularSetCategory`
- `help RealClosure`
- `help Fraction`
- `show LexTriangularPackage`
LexTriangularPackage (LEXTRIPK)

Exports:

fglmIfCan groebner lexTriangular
squareFreeLexTriangular zeroDimensional? zeroSetSplit

package LEXTRIPK LexTriangularPackage

LexTriangularPackage(R,ls): Exports == Implementation where

R: GcdDomain
ls: List Symbol
V => OrderedVariableList ls
E ==> IndexedExponents V
P ==> NewSparseMultivariatePolynomial(R,V)
TS ==> RegularChain(R,ls)
ST ==> SquareFreeRegularTriangularSet(R,E,V,P)
Q1 ==> Polynomial R
PS ==> GeneralPolynomialSet(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
K ==> Fraction R
LP ==> List P
BWTS ==> Record(val : Boolean, tower : TS)
LpWTS ==> Record(val : (List P), tower : TS)
BWST ==> Record(val : Boolean, tower : ST)
LpWST ==> Record(val : (List P), tower : ST)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppackTS ==> QuasiComponentPackage(R,E,V,P,TS)
regsetgcdpackTS ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)
normalizpackTS ==> NormalizationPackage(R,E,V,P,TS)
quasicomppackST ==> QuasiComponentPackage(R,E,V,P,ST)
regsetgcdpackST ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,ST)
normalizpackST ==> NormalizationPackage(R,E,V,P,ST)

Exports == with

  zeroDimensional?: LP -> B
  ++ \axiom{zeroDimensional?(lp)} returns true iff
  ++ \axiom{lp} generates a zero-dimensional ideal
  ++ w.r.t. the variables involved in \axiom{lp}.
  fglmIfCan: LP -> Union(LP, "failed")
  ++ \axiom{fglmIfCan(lp)} returns the lexicographical Groebner
  ++ basis of \axiom{lp} by using the FGLM strategy,
  ++ if \axiom{zeroDimensional?(lp)} holds .
  groebner: LP -> LP
  ++ \axiom{groebner(lp)} returns the lexicographical Groebner
  ++ basis of \axiom{lp}. If \axiom{lp} generates a zero-dimensional
  ++ ideal then the FGLM strategy is used, otherwise
  ++ the Sugar strategy is used.
  lexTriangular: (LP, B) -> List TS
  ++ \axiom{lexTriangular(base, norm?)} decomposes the variety
  ++ associated with \axiom{base} into regular chains.
  ++ Thus a point belongs to this variety iff it is a regular
  ++ zero of a regular set in the output.
  ++ Note that \axiom{base} needs to be a lexicographical Groebner basis
  ++ of a zero-dimensional ideal. If \axiom{norm?} is \axiom{true}
  ++ then the regular sets are normalized.
  squareFreeLexTriangular: (LP, B) -> List ST
  ++ \axiom{squareFreeLexTriangular(base, norm?)} decomposes the variety
  ++ associated with \axiom{base} into square-free regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \texttt{axiom(base)} needs to be a lexicographical Groebner basis
++ of a zero-dimensional ideal. If \texttt{axiom(norm?)} is \texttt{axiom(true} 
++ then the regular sets are normalized.

\texttt{zeroSetSplit: (LP, B) -> List TS}
++ \texttt{axiom(zeroSetSplit(lp, norm?)} decomposes the variety
++ associated with \texttt{axiom(lp)} into regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \texttt{axiom(lp)} needs to generate a zero-dimensional ideal.
++ If \texttt{axiom(norm?)} is \texttt{axiom(true)} then the regular sets are normalized.

\texttt{zeroSetSplit: (LP, B) -> List ST}
++ \texttt{axiom(zeroSetSplit(lp, norm?)} decomposes the variety
++ associated with \texttt{axiom(lp)} into square-free regular chains.
++ Thus a point belongs to this variety iff it is a regular
++ zero of a regular set in in the output.
++ Note that \texttt{axiom(lp)} needs to generate a zero-dimensional ideal.
++ If \texttt{axiom(norm?)} is \texttt{axiom(true)} then the regular sets are normalized.

\texttt{Implementation == add

\texttt{trueVariables(lp: List(P)): List Symbol ==
  lv: List V := variables([lp]$PS)
  truels: List Symbol := []
  for s in ls repeat
    if member?(variable(s)::V, lv) then truels := cons(s,truels)
  reverse truels

\texttt{zeroDimensional?(lp:List(P)): Boolean ==
  truels: List Symbol := trueVariables(lp)
  fglmmpack := FGLMIfCanPackage(R,truels)
  lq1: List(Q1) := [p::Q1 for p in lp]
  zeroDimensional?(lq1)$fglmmpack

\texttt{fglmIfCan(lp:List(P)): Union(List(P), "failed") ==
  truels: List Symbol := trueVariables(lp)
  fglmmpack := FGLMIfCanPackage(R,truels)
  lq1: List(Q1) := [p::Q1 for p in lp]
  foo := fglmIfCan(lq1)$fglmmpack
  foo case "failed" => return("failed" :: Union(List(P), "failed"))
  lp := [retract(q1)$P for q1 in (foo :: List(Q1))]
  lp::Union(List(P), "failed")

\texttt{groebner(lp:List(P)): List(P) ==
  truels: List Symbol := trueVariables(lp)
  fglmmpack := FGLMIfCanPackage(R,truels)
  lq1: List(Q1) := [p::Q1 for p in lp]
  lq1 := groebner(lq1)$fglmmpack
  lp := [retract(q1)$P for q1 in lq1]
lexTriangular(base: List(P), norm?: Boolean): List(TS) ==
    base := sort(infRittWu?,base)
    base := remove(zero?, base)
    any?(ground?, base) => []
    ts: TS := empty()
    toSee: List LpWTS := [[base,ts]$LpWTS]
    toSave: List TS := []
    while not empty? toSee repeat
        lpwt := first toSee; toSee := rest toSee
        lp := lpwt.val; ts := lpwt.tower
        empty? lp => toSave := cons(ts, toSave)
        p := first lp; lp := rest lp; v := mvar(p)
        algebraic?(v,ts) =>
            error "lexTriangular$LEXTRIPK: should never happen !"
        norm? and zero? remainder(init(p),ts).polnum =>
            toSee := cons([lp, ts]$LpWTS, toSee)
        (not norm?) and zero? (initiallyReduce(init(p),ts)) =>
            toSee := cons([lp, ts]$LpWTS, toSee)
        lbwt: List BWTS := invertible?(init(p),ts)$TS
        while (not empty? lbwt) repeat
            bwt := first lbwt; lbwt := rest lbwt
            b := bwt.val; us := bwt.tower
            (not b) => toSee := cons([lp, us], toSave)
        lus: List TS
        if norm?
            then
                newp := normalizedAssociate(p,us)$normalizpackTS
                lus := [internalAugment(newp,us)$TS]
            else
                newp := p
                lus := augment(newp,us)$TS
        newlp := lp
        while (not empty? newlp) and (mvar(first newlp) = v) repeat
            newlp := rest newlp
        for us in lus repeat
            toSee := cons([newlp, us]$LpWTS, toSee)
    algebraicSort(toSave)$quasicomppackTS

zeroSetSplit(lp:List(P), norm?:B): List TS ==
    bar := fglmIfCan(lp)
    bar case "failed" =>
        error "zeroSetSplit$LEXTRIPK: #1 not zero-dimensional"
    lexTriangular(bar::(List P),norm?)

squareFreeLexTriangular(base: List(P), norm?: Boolean): List(ST) ==
    base := sort(infRittWu?,base)
    base := remove(zero?, base)
    any?(ground?, base) => []
    ts: ST := empty()
toSee: List LpWST := [[base,ts]$LpWST]
toSave: List ST := []
while not empty? toSee repeat
    lpwt := first toSee; toSee := rest toSee
    lp := lpwt.val; ts := lpwt.tower
    empty? lp => toSave := cons(ts, toSave)
    p := first lp; lp := rest lp; v := mvar(p)
    algebraic?(v,ts) =>
        error "lexTriangular$LEXTRIPK: should never happen !"
    norm? and zero? remainder(init(p),ts).polnum =>
        toSee := cons([lp, ts]$LpWST, toSee)
    (not norm?) and zero? (initiallyReduce(init(p),ts)) =>
        toSee := cons([lp, ts]$LpWST, toSee)
    lbwt: List BWST := invertible?(init(p),ts)$ST
    while (not empty? lbwt) repeat
        bwt := first lbwt; lbwt := rest lbwt
        b := bwt.val; us := bwt.tower
        (not b) => toSee := cons([lp, us], toSee)
    lus: List ST
    if norm?
        then
            newp := normalizedAssociate(p,us)$normalizpackST
            lus := augment(newp,us)$ST
        else
            lus := augment(p,us)$ST
    newlp := lp
    while (not empty? newlp) and (mvar(first newlp) = v) repeat
        newlp := rest newlp
    for us in lus repeat
        toSee := cons([newlp, us]$LpWST, toSee)
    algebraicSort(toSave)$quasicomppackST

zeroSetSplit(lp:List(P), norm?:B): List ST ==
    bar := fglmIfCan(lp)
    bar case "failed" =>
        error "zeroSetSplit$LEXTRIPK: #1 not zero-dimensional"
squareFreeLexTriangular(bar::(List P),norm?)

|—— LEXTRIPK.dotabb ——|

"LEXTRIPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LEXTRIPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"LEXTRIPK" -> "SFRTCAT"
package LINDEP LinearDependence

--- LinearDependence.input ---

)set break resume
)sys rm -f LinearDependence.output
)spool LinearDependence.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearDependence
--R
--R LinearDependence(S: IntegralDomain,R: LinearlyExplicitRingOver(S)) is a package constructor
--R Abbreviation for LinearDependence is LINDEP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LINDEP
--R
--R---------------------------------------- Operations ----------------------------------------
--R linearDependence : Vector(R) -> Union(Vector(S),"failed")
--R linearlyDependent? : Vector(R) -> Boolean
--R solveLinear : (Vector(R),R) -> Union(Vector(S),"failed") if S has FIELD
--R solveLinear : (Vector(R),R) -> Union(Vector(Fraction(S)),"failed") if not(has(S,Field))
--R
--E 1

)spool
)lisp (bye)

---

--- LinearDependence.help ---

LinearDependence examples

Test for linear dependence.

See Also:
- )show LinearDependence

---
LinearDependence (LINDEP)

Exports:
linearDependence  linearlyDependent?  solveLinear

— package LINDEP LinearDependence —

)abbrev package LINDEP LinearDependence
++ Author: Manuel Bronstein
++ Date Last Updated: 14 May 1991
++ Description:
++ Test for linear dependence.

LinearDependence(S, R): Exports == Implementation where
  S: IntegralDomain
  R: LinearlyExplicitRingOver S
  Q == Fraction S

Exports == with
  linearlyDependent?: Vector R -> Boolean
    ++ \spad{linearlyDependent?([v1,...,vn])} returns true if
    ++ the vi's are linearly dependent over S, false otherwise.
  linearDependence : Vector R -> Union(Vector S, "failed")
    ++ \spad{linearDependence([v1,...,vn])} returns \spad{[c1,...,cn]} if
    ++ \spad{c1*v1 + ... + cn*vn = 0} and not all the ci's are 0,
    ++ "failed" if the vi's are linearly independent over S.
  if S has Field then
    solveLinear: (Vector R, R) -> Union(Vector S, "failed")
      ++ \spad{solveLinear([v1,...,vn], u)} returns \spad{[c1,...,cn]}
      ++ such that \spad{c1*v1 + ... + cn*vn = u},
      ++ "failed" if no such ci's exist in S.
  else
    solveLinear: (Vector R, R) -> Union(Vector Q, "failed")
      ++ \spad{solveLinear([v1,...,vn], u)} returns \spad{[c1,...,cn]}
      ++ such that \spad{c1*v1 + ... + cn*vn = u},
      ++ "failed" if no such ci's exist in the quotient field of S.
Implementation ==> add
aNonZeroSolution: Matrix S -> Union(Vector S, "failed")

aNonZeroSolution m ==
every?(zero?, v := first nullSpace m) => "failed"
v
linearlyDependent? v ==
zero?(n := #v) => true
-- one? n => zero?(v(minIndex v))
(n = 1) => zero?(v(minIndex v))
positive? nullity reducedSystem transpose v

linearDependence v ==
zero?(n := #v) => empty()
-- one? n =>
(n = 1) =>
zero?(v(minIndex v)) => new(1, 1)
"failed"
aNonZeroSolution reducedSystem transpose v

if S has Field then
solveLinear(v:Vector R, c:R):Union(Vector S, "failed") ==
zero? c => new(#v, 0)
empty? v => "failed"
sys := reducedSystem(transpose v, new(1, c))
partialSolution(sys.mat, sys.vec)$LinearSystemMatrixPackage(S, Vector S, Vector S, Matrix S)

else
solveLinear(v:Vector R, c:R):Union(Vector Q, "failed") ==
zero? c => new(#v, 0)
empty? v => "failed"
sys := reducedSystem(transpose v, new(1, c))
partialSolution(map((z:S):Q+->z::Q, sys.mat),
$MatrixCategoryFunctions2(S, Vector S, Vector S, Matrix S, Q, Vector Q, Vector Q, Matrix Q),
map((z1:S):Q+-z1::Q, sys.vec)$VectorFunctions2(S, Q))$LinearSystemMatrixPackage(Q, Vector Q, Vector Q, Matrix Q)

— LINDEP.dotabb —
package LODOF LinearOrdinaryDifferentialOperatorFactorizer

-- LinearOrdinaryDifferentialOperatorFactorizer.input --

)set break resume
)sys rm -f LinearOrdinaryDifferentialOperatorFactorizer.output
)spool LinearOrdinaryDifferentialOperatorFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearOrdinaryDifferentialOperatorFactorizer
--R LinearOrdinaryDifferentialOperatorFactorizer(F: Join(Field,CharacteristicZero,RetractableTo(Integer),RetractableTo(Fraction(Integer))),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for LinearOrdinaryDifferentialOperatorFactorizer is LODOF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LODOF
--R
--R----------------------------- Operations --------------------------------
--R factor : (LinearOrdinaryDifferentialOperator1(Fraction(UP)),(UP -> List(F))) -> List(LinearOrdinaryDifferentialOperator1(Fraction(UP))) if F has ACF
--R factor : LinearOrdinaryDifferentialOperator1(Fraction(UP)) -> List(LinearOrdinaryDifferentialOperator1(Fraction(UP))) if F has ACF
--R factor1 : LinearOrdinaryDifferentialOperator1(Fraction(UP)) -> List(LinearOrdinaryDifferentialOperator1(Fraction(UP)))
--R
--E 1

)spool
)lisp (bye)

-- LinearOrdinaryDifferentialOperatorFactorizer.help --

================================= Operations ================================
LinearOrdinaryDifferentialOperatorFactorizer examples

LinearOrdinaryDifferentialOperatorFactorizer provides a factorizer for linear ordinary differential operators whose coefficients are rational
functions.

See Also:
o )show LinearOrdinaryDifferentialOperatorFactorizer

——

LinearOrdinaryDifferentialOperatorFactorizer (LODOF)

LODOF

ACF

Exports:
factor factor1

—— package LODOF LinearOrdinaryDifferentialOperatorFactorizer ——

)abbrev package LODOF LinearOrdinaryDifferentialOperatorFactorizer
++ Author: Fritz Schwarz, Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{LinearOrdinaryDifferentialOperatorFactorizer} provides a
++ factorizer for linear ordinary differential operators whose coefficients
++ are rational functions.

LinearOrdinaryDifferentialOperatorFactorizer(F, UP): Exports == Impl where
  F : Join(Field, CharacteristicZero,
    RetractableTo Integer, RetractableTo Fraction Integer)
  UP: UnivariatePolynomialCategory F
  RF ==> Fraction UP
  L ==> LinearOrdinaryDifferentialOperator1 RF

Exports => with
  factor: (L, UP -> List F) -> List L
++ factor(a, zeros) returns the factorisation of a.
++ \spad{zeros} is a zero finder in \spad{UP}.
if \spad{F} has AlgebraicallyClosedField then
  factor: \spad{L} -> \spad{List L}
    ++ factor(a) returns the factorisation of a.
  factor1: \spad{L} -> \spad{List L}
    ++ factor1(a) returns the factorisation of a,
    ++ assuming that a has no first-order right factor.

Impl => add
  import RationalLODE(F, UP)
  import RationalRicDE(F, UP)
-- import AssociatedEquations RF

dd := D()$L

expsol : (L, UP -> List F, UP -> Factored UP) -> Union(RF, "failed")
expsols : (L, UP -> List F, UP -> Factored UP, Boolean) -> List RF
opeval : (L, L) -> L
recurfactor: (L, L, UP -> List F, UP -> Factored UP, Boolean) -> List L
rfactor : (L, L, UP -> List F, UP -> Factored UP, Boolean) -> List L
rightFactor: (L, NonNegativeInteger, UP -> List F, UP -> Factored UP)
    -> Union(L, "failed")
innerFactor: (L, UP -> List F, UP -> Factored UP, Boolean) -> List L

factor(l, zeros) == innerFactor(l, zeros, squareFree, true)

expsol(l, zeros, ezfactor) ==
  empty?(sol := expsols(l, zeros, ezfactor, false)) => "failed"
  first sol

expsols(l, zeros, ezfactor, all?) ==
  sol := [differentiate(f)/f for f in ratDsolve(l, 0).basis | f ^= 0]
  not(all? or empty? sol) => sol
  concat(sol, ricDsolve(l, zeros, ezfactor))

-- opeval(l1, l2) returns l1(l2)
opeval(l1, l2) ==
  ans:L := 0
  l2n:L := 1
  for i in 0..degree l1 repeat
    ans := ans + coefficient(l1, i) * l2n
    l2n := l2 * l2n
  ans

recurfactor(l, r, zeros, ezfactor, adj?) ==
  q := rightExactQuotient(l, r)::L
  if adj? then q := adjoint q
  innerFactor(q, zeros, ezfactor, true)
rfactor(op, r, zeros, ezfactor, adj?) ==
    degree r > 1 or not one? leadingCoefficient r =>
    degree r > 1 or not ((leadingCoefficient r) = 1) =>
    recurfactor(op, r, zeros, ezfactor, adj?)
    op1 := opeval(op, dd - coefficient(r, 0)::L)
    map_!((z:L):L+->opeval(z,r), recurfactor(op1, dd, zeros, ezfactor, adj?))

-- r1? is true means look for 1st-order right-factor also
innerFactor(l, zeros, ezfactor, r1?) ==
    (n := degree l) <= 1 => [l]
    ll := adjoint l
    for i in 1..(n quo 2) repeat
        (r1? or (i > 1)) and ((u := rightFactor(l, i, zeros, ezfactor)) case L) =>
            return concat_!(rfactor(l, u::L, zeros, ezfactor, false), u::L)
        (2 * i < n) and ((u := rightFactor(ll, i, zeros, ezfactor)) case L) =>
            return concat(adjoint(u::L), rfactor(ll, u::L, zeros, ezfactor, true))
    [l]

rightFactor(l, n, zeros, ezfactor) ==
    -- one? n =>
    (n = 1) =>
        (u := expsol(l, zeros, ezfactor)) case "failed" => "failed"
        D() - u::RF::L
    -- rec := associatedEquations(l, n::PositiveInteger)
    -- empty?(sol := expsols(rec.eq, zeros, ezfactor, true)) => "failed"
    "failed"
    if F has AlgebraicallyClosedField then
        zro1: UP -> List F
        zro : (UP, UP -> Factored UP) -> List F

        zro(p, ezfactor) ==
            concat [zro1(r.factor) for r in factors ezfactor p]
        zro1 p ==
            [zeroOf(map((z1:F):F+->z1,p_ $UnivariatePolynomialCategoryFunctions2(F, UP, F, SparseUnivariatePolynomial F))] 
    if F is AlgebraicNumber then
        import AlgFactor UP

        factor 1 ==
            innerFactor(1, (p:UP):List(F)++>zro(p,factor),factor, true)
        factor1 1 ==
            innerFactor(1, (p:UP):List(F)++>zro(p,factor),factor, false)
    else
        factor 1 ==
            innerFactor(1, (p:UP):List(F)++>zro(p,squareFree),squareFree, true)
factor1 l ==
in innerFactor1, (p:UP):List(F) +-> zro(zr, squareFree), squareFree, false)

— LODOF.dotabb —
"LODOF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODOF"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"LODOF" -> "ACF"

package LODOOPS LinearOrdinaryDifferentialOperatorsOps

— LinearOrdinaryDifferentialOperatorsOps.input —

)set break resume
/sys rm -f LinearOrdinaryDifferentialOperatorsOps.output
/spool LinearOrdinaryDifferentialOperatorsOps.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearOrdinaryDifferentialOperatorsOps
--R
--R LinearOrdinaryDifferentialOperatorsOps(A: Field,L: LinearOrdinaryDifferentialOperatorCategory(A)) is a
--R Abbreviation for LinearOrdinaryDifferentialOperatorsOps is LODOOPS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LODOOPS
--R
--R---------------------------------------------------- Operations ----------------------------------
--R directSum : (L,L,(A -> A)) -> L
--R symmetricPower : (L,NonNegativeInteger,(A -> A)) -> L
--R symmetricProduct : (L,L,(A -> A)) -> L
--R
--E 1

)spool
)lisp (bye)

— —
LinearOrdinaryDifferentialOperatorsOps (LODOOPS)

Exports:
directSum  symmetricPower  symmetricProduct

— package LODOOPS LinearOrdinaryDifferentialOperatorsOps —

)abbrev package LODOOPS LinearOrdinaryDifferentialOperatorsOps
++ Author: Manuel Bronstein
++ Date Created: 18 January 1994
++ Date Last Updated: 15 April 1994
++ Description:
++ \texttt{LinearOrdinaryDifferentialOperatorsOps} provides symmetric
++ products and sums for linear ordinary differential operators.
-- Putting those operations here rather than defaults in LODOCAT allows
-- LODOCAT to be defined independently of the derivative used.
-- MB 1/94
LinearOrdinaryDifferentialOperatorsOps(A, L): Exports == Implementation where
A: Field
L: LinearOrdinaryDifferentialOperatorCategory A

N ==> NonNegativeInteger
V ==> OrderlyDifferentialVariable Symbol
P ==> DifferentialSparseMultivariatePolynomial(A, Symbol, V)

Exports ==> with
symmetricProduct: (L, L, A -> A) -> L
++ symmetricProduct(a,b,D) computes an operator \spad{c} of
++ minimal order such that the nullspace of \spad{c} is
++ generated by all the products of a solution of \spad{a} by
++ a solution of \spad{b}.
++ D is the derivation to use.
symmetricPower: (L, N, A -> A) -> L
++ symmetricPower(a,n,D) computes an operator \spad{c} of
++ minimal order such that the nullspace of \spad{c} is
++ generated by all the products of \spad{n} solutions
++ of \spad{a}.
++ D is the derivation to use.
directSum: (L, L, A -> A) -> L
++ directSum(a,b,D) computes an operator \spad{c} of
++ minimal order such that the nullspace of \spad{c} is
++ generated by all the sums of a solution of \spad{a} by
++ a solution of \spad{b}.
++ D is the derivation to use.

Implementation ==> add
import IntegerCombinatoricFunctions

var1 := new()$Symbol
var2 := new()$Symbol

nonTrivial?: Vector A -> Boolean
applyLODO : (L, V) -> P
killer : (P, N, List V, List P, A -> A) -> L
vec2LODO : Vector A -> L

nonTrivial? v == any?((x1:A):Boolean +-> x1 ^= 0, v)$Vector(A)
vec2LODO v == +/[monomial(v.i, (i-1)::N) for i in 1..#v]

symmetricPower(l, m, diff) ==
  u := var1::V; n := degree l
  un := differentiate(u, n)
a := applyLODO(inv(- leadingCoefficient l) * reductum l, u)
killer(u::P ** m, binomial(n + m - 1, n - 1)::N, [un], [a], diff)

-- returns an operator L such that L(u) = 0, for a given differential
-- polynomial u, given that the differential variables appearing in u
-- satisfy some linear ode's
-- m is a bound on the order of the operator searched.
-- lvar, lval describe the substitution(s) to perform when differentiating
-- the expression u (they encode the fact the the differential variables
-- satisfy some differential equations, which can be seen as the rewrite
-- rules lvar --> lval)
-- diff is the derivation to use
killer(u, m, lvar, lval, diff) ==
lu:List P := [u]
for q in 0..m repeat
  mat := reducedSystem(matrix([lu])@Matrix(P))@Matrix(A)
  (sol := find(nonTrivial?, 1 := nullSpace mat)) case Vector(A) =>
    return vec2LODO(sol::Vector(A))
  u := eval(differentiate(u, diff), lvar, lval)
  lu := concat!(lu, [u])
error "killer: no linear dependence found"
symmetricProduct(l1, l2, diff) ==
u := var1::V; v := var2::V
n1 := degree l1; n2 := degree l2
un := differentiate(u, n1); vn := differentiate(v, n2)
a := applyLODO(inv(- leadingCoefficient l1) * reductum l1, u)
b := applyLODO(inv(- leadingCoefficient l2) * reductum l2, v)
killer(u::P * v::P, n1 * n2, [un, vn], [a, b], diff)
directSum(l1, l2, diff) ==
u := var1::V; v := var2::V
n1 := degree l1; n2 := degree l2
un := differentiate(u, n1); vn := differentiate(v, n2)
a := applyLODO(inv(- leadingCoefficient l1) * reductum l1, u)
b := applyLODO(inv(- leadingCoefficient l2) * reductum l2, v)
killer(u::P + v::P, n1 + n2, [un, vn], [a, b], diff)
applyLODO(l, v) ==
p:P := 0
while l ^= 0 repeat
  p := p + monomial(leadingCoefficient(l)::P,
                    differentiate(v, degree l), 1)
  l := reductum l
p

| LODOOPS.dotabb |

"LODOOPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LODOOPS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"LODOOPS" -> "ALIST"
package LPEFRAC LinearPolynomialEquationByFractions

LinearPolynomialEquationByFractions examples

Given a PolynomialFactorizationExplicit ring, this package provides a defaulting rule for the solveLinearPolynomialEquation operation, by moving into the field of fractions, and solving it there via the multiEuclidean operation.

See Also:
  o )show LinearPolynomialEquationByFractions
Exports:
solveLinearPolynomialEquationByFractions

— package LPEFRAC LinearPolynomialEquationByFractions —

)abbrev package LPEFRAC LinearPolynomialEquationByFractions
++ Author: James Davenport
++ Description:
++ Given a PolynomialFactorizationExplicit ring, this package
++ provides a defaulting rule for the \spad{solveLinearPolynomialEquation}
++ operation, by moving into the field of fractions, and solving it there
++ via the \spad{multiEuclidean} operation.

LinearPolynomialEquationByFractions(R:PolynomialFactorizationExplicit): with
solveLinearPolynomialEquationByFractions: ( _
   List SparseUnivariatePolynomial R, _
   SparseUnivariatePolynomial R) -> _
   Union(List SparseUnivariatePolynomial R, "failed")
++ solveLinearPolynomialEquationByFractions([f1, ..., fn], g)
++ (where the fi are relatively prime to each other)
++ returns a list of ai such that
++ \spad{g/prod fi = sum ai/fi}
++ or returns "failed" if no such exists.

== add
SupR ==> SparseUnivariatePolynomial R
F ==> Fraction R
SupF ==> SparseUnivariatePolynomial F
import UnivariatePolynomialCategoryFunctions2(R,SupR,F,SupF)
lp : List SupR
pp : SupR
pF: SupF
pullback : SupF -> Union(SupR,"failed")
pullback(pF) ==
  pF = 0 => 0
  c:=retractIfCan leadingCoefficient pF
  c case "failed" => "failed"
  r:=pullback reductum pF
  r case "failed" => "failed"
  monomial(c,degree pF) + r

solveLinearPolynomialEquationByFractions(lp,pp) ==
  lpF:List SupF:=[map((x:R):F +-> x@R::F,u) for u in lp]
  pF:SupF:=map((x:R):F +-> x::F,pp)
  ans:= solveLinearPolynomialEquation(lpF,pF)$F
  ans case "failed" => "failed"
  [(vv:= pullback v;
    vv case "failed" => return "failed";
    vv)
    for v in ans]

package LISYSER LinearSystemFromPowerSeriesPackage

— LinearSystemFromPowerSeriesPackage.input —

)set break resume
)sys rm -f LinearSystemFromPowerSeriesPackage.output
)spool LinearSystemFromPowerSeriesPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemFromPowerSeriesPackage
--R
--R LinearSystemFromPowerSeriesPackage(K: Field,PCS: LocalPowerSeriesCategory(K)) is a package constructor
--R Abbreviation for LinearSystemFromPowerSeriesPackage is LISYSER
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LISYSER
--R
--R------------------------------- Operations --------------------------------
--R finiteSeries2LinSys : (List(PCS),Integer) -> Matrix(K)
--R finiteSeries2LinSysWOVectorise : (List(PCS),Integer) -> Matrix(K)
--R finiteSeries2Vector : (PCS,Integer) -> List(K)
--R
--E 1

)spool
)lisp (bye)

---

--- LinearSystemFromPowerSeriesPackage.help ---

====================================================================
LinearSystemFromPowerSeriesPackage examples
====================================================================

Part of the PAFF package

See Also:
o )show LinearSystemFromPowerSeriesPackage

---

LinearSystemFromPowerSeriesPackage (LISYSER)

Exports:
finiteSeries2LinSys  finiteSeries2LinSysWOVectorise  finiteSeries2Vector
package LISYSER LinearSystemFromPowerSeriesPackage —

)abbrev package LISYSER LinearSystemFromPowerSeriesPackage
++ Authors: Gaetan Hache
++ Date Created: 1996
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ Part of the PAFF package
LinearSystemFromPowerSeriesPackage(K,PCS):P==T where
  K : Field
  PCS: LocalPowerSeriesCategory(K)

INT    ==> Integer
TERM   ==> Record(k:INT,c:K)
SER    ==> Stream(TERM)
LOpPack ==> LinesOpPack(K)

P==>
  finiteSeries2LinSysWOVectorise: (List PCS, INT) -> Matrix K

finiteSeries2LinSys: (List PCS, INT) -> Matrix K
  ++ finiteSeries2LinSys(ls,n) returns a matrix which right kernel
  ++ is the solution of the linear combinations of the series in ls
  ++ which has order greater or equal to n.
  ++ NOTE: All the series in ls must be finite and must have order
  ++ at least 0: so one must first call on each of them the
  ++ function filterUpTo(s,n) and apply an appropriate shift
  ++ (mult by a power of t).

finiteSeries2Vector: (PCS, INT) -> List K
T==> add
  finiteSeries2ListOfTerms: PCS -> List TERM
  finiteSeries2ListOfTermsStream: SER -> List TERM
  finiteSeries2ListOfTermsStream(s)==
    empty?(s) => empty()
    cons(frst s , finiteSeries2ListOfTermsStream(rst(s)))

finiteSeries2LinSys(ls,n)==
  ll:List K:= [0$K]
  lZero:=new(#ls pretend NonNegativeInteger,ll)$List(List(K))
  n <= 0 => transpose matrix lZero
  tMat:=transpose matrix [finiteSeries2Vector(s,n) for s in ls]
  rowEchWoZeroLines(tMat)$LOpPack

finiteSeries2LinSysWOVectorise(ls,n)==
  ll:List K:= [0$K]
lZero := new(#ls pretend NonNegativeInteger, 11)$List(List(K))
n <= 0 => transpose matrix lZero
tMat := transpose matrix [finiteSeries2Vector(s, n) for s in ls] rowEchNoZeroLinesWOVectorise(tMat)$L0pPack

finiteSeries2List0fTerms(s) ==
ss:SER := s :: SER
finiteSeries2List0fTermsStream(ss)
finiteSeries2Vector(ins, n) ==
lZero := new((n pretend NonNegativeInteger), 0)$List(K)
s := removeFirstZeros ins
lOfTerm := finiteSeries2List0fTerms(s)
for t in lOfTerm repeat lZero.((t.k)+1) := t.c
lZero

———

— LISYSER.dotabb —
"LISYSER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LISYSER"]
"LOCPOWC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LOCPOWC"]
"LISYSER" -> "LOCPOWC"

———

package LSMP LinearSystemMatrixPackage

— LinearSystemMatrixPackage.input —

)set break resume
)sys rm -f LinearSystemMatrixPackage.output
)spool LinearSystemMatrixPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemMatrixPackage
--R
--R LinearSystemMatrixPackage(F: Field, Row, Col, M: MatrixCategory(F, Row, Col))where
--R Row: FiniteLinearAggregate(F) with
--R shallowlyMutable
--R Col: FiniteLinearAggregate(F) with
--R     shallowlyMutable is a package constructor
--R Abbreviation for LinearSystemMatrixPackage is LSMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LSMP
--R
--R----------------------------------- Operations -----------------------------------
--R hasSolution? : (M,Col) -> Boolean  rank : (M,Col) -> NonNegativeInteger
--R particularSolution : (M,Col) -> Union(Col,"failed")
--R solve : (M,Col) -> Record(particular: Union(Col,"failed"),basis: List(Col))
--R solve : (M,List(Col)) -> List(Record(particular: Union(Col,"failed"),basis: List(Col)))
--R
--E 1
)
spool
lisp (bye)

— LinearSystemMatrixPackage.help —

====================================================================
LinearSystemMatrixPackage examples
====================================================================

This package solves linear system in the matrix form AX = B.

See Also:
o )show LinearSystemMatrixPackage

LinearSystemMatrixPackage (LSMP)
CHAPTER 13. CHAPTER L

Exports:
hasSolution? particularSolution rank solve

— package LSMP LinearSystemMatrixPackage —

)abbrev package LSMP LinearSystemMatrixPackage
++ Author: P.Gianni, S.Watt
++ Date Created: Summer 1985
++ Date Last Updated: Summer 1990
++ Description:
++ This package solves linear system in the matrix form \spad{AX = B}.

LinearSystemMatrixPackage(F, Row, Col, M): Cat == Capsule where
  F: Field
  Row: FiniteLinearAggregate F with shallowlyMutable
  Col: FiniteLinearAggregate F with shallowlyMutable
  M : MatrixCategory(F, Row, Col)

N ==> NonNegativeInteger
PartialV ==> Union(Col, "failed")
Both ==> Record(particular: PartialV, basis: List Col)

Cat ==> with
  solve : (M, Col) -> Both
  ++ solve(A,B) finds a particular solution of the system \spad{AX = B}
  ++ and a basis of the associated homogeneous system \spad{AX = 0}.
  solve : (M, List Col) -> List Both
  ++ solve(A,LB) finds a particular soln of the systems \spad{AX = B}
  ++ and a basis of the associated homogeneous systems \spad{AX = 0}
  ++ where B varies in the list of column vectors LB.

  particularSolution: (M, Col) -> PartialV
  ++ particularSolution(A,B) finds a particular solution of the linear
  ++ system \spad{AX = B}.
  hasSolution?: (M, Col) -> Boolean
  ++ hasSolution?(A,B) tests if the linear system \spad{AX = B}
  ++ has a solution.
  rank : (M, Col) -> N
  ++ rank(A,B) computes the rank of the complete matrix \spad{(A|B)}
  ++ of the linear system \spad{AX = B}.

Capsule ==> add
  systemMatrix : (M, Col) -> M
  aSolution : M -> PartialV

-- rank theorem
  hasSolution?(A, b) == rank A = rank systemMatrix(A, b)
  systemMatrix(m, v) == horizConcat(m, -(v::M))
  rank(A, b) == rank systemMatrix(A, b)
particularSolution(A, b) == aSolution rowEchelon systemMatrix(A,b)

-- m should be in row-echelon form.
-- last column of m is -(right-hand-side of system)
aSolution m ==
  nvar := (ncols m - 1)::N
  rk := maxRowIndex m
  while (rk >= minRowIndex m) and every?(zero?, row(m, rk))
    repeat rk := dec rk
  rk < minRowIndex m => new(nvar, 0)
  ck := minColIndex m
  while (ck < maxColIndex m) and zero? qelt(m, rk, ck) repeat
    ck := inc ck
  ck = maxColIndex m => "failed"
sol := new(nvar, 0)$Col
  -- find leading elements of diagonal
  v := new(nvar, minRowIndex m - 1)$PrimitiveArray(Integer)
  for i in minRowIndex m .. rk repeat
    for j in 0.. while zero? qelt(m, i, j+minColIndex m) repeat 0
      v.j := i
    sol :+ (qelt(m, v.j, maxColIndex m))
sol

solve(A:M, b:Col) ==
  -- Special case for homogeneous systems.
  every?(zero?, b) => [new(ncols A, 0), nullSpace A]
  -- General case.
  m := rowEchelon systemMatrix(A, b)
  [aSolution m,
   nullSpace subMatrix(m, minRowIndex m, maxRowIndex m,
                        minColIndex m, maxColIndex m - 1)]

solve(A:M, l:List Col) ==
  null l => [[new(ncols A, 0), nullSpace A]]
  nl := (sol0 := solve(A, first l)).basis
  cons(sol0,
       [[aSolution rowEchelon systemMatrix(A, b), nl]
        for b in rest l])
package LSMP1 LinearSystemMatrixPackage1

— LinearSystemMatrixPackage1.input —

)set break resume
)sys rm -f LinearSystemMatrixPackage1.output
)spool LinearSystemMatrixPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemMatrixPackage1
--R
--R LinearSystemMatrixPackage1(F: Field) is a package constructor
--R Abbreviation for LinearSystemMatrixPackage1 is LSMP1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LSMP1
--R
--R------------------- Operations -------------------
--R hasSolution? : (Matrix(F),Vector(F)) -> Boolean
--R particularSolution : (Matrix(F),Vector(F)) -> Union(Vector(F),"failed")
--R rank : (Matrix(F),Vector(F)) -> NonNegativeInteger
--R solve : (Matrix(F),Vector(F)) -> Record(particular: Union(Vector(F),"failed"),basis: List(Vector(F)))
--R solve : (List(List(F)),Vector(F)) -> Record(particular: Union(Vector(F),"failed"),basis: List(List(Vector(F))))
--R solve : (Matrix(F),List(Vector(F))) -> List(Record(particular: Union(Vector(F),"failed"),basis: List(Vector(F))))
--R solve : (List(List(F)),List(Vector(F))) -> List(Record(particular: Union(Vector(F),"failed"),basis: List(List(Vector(F)))))
--R
--E 1

)spool
)lisp (bye)

— LinearSystemMatrixPackage1.help —

====================================================================
LinearSystemMatrixPackage1 examples
====================================================================

This package solves linear system in the matrix form \( AX = B \). It is
essentially a particular instantiation of the package
LinearSystemMatrixPackage for Matrix and Vector. This package's
existence makes it easier to use solve in the Axiom interpreter.

See Also:
o )show LinearSystemMatrixPackage1

LinearSystemMatrixPackage1 (LSMP1)

Exports:
  hasSolution?  particularSolution  rank  solve

— package LSMP1 LinearSystemMatrixPackage1 —

)abbrev package LSMP1 LinearSystemMatrixPackage1
++ Author: R. Sutor
++ Date Created: June, 1994
++ Description:
++ This package solves linear system in the matrix form \(AX = B\).
++ It is essentially a particular instantiation of the package
++ \spadtype{LinearSystemMatrixPackage} for Matrix and Vector. This
++ package's existence makes it easier to use \spadfun{solve} in the
++ Axiom interpreter.

LinearSystemMatrixPackage1(F): Cat == Capsule where
  F: Field
  Row   ==> Vector F
  Col   ==> Vector F
  M     ==> Matrix(F)
  LL    ==> List List F
N ==> NonNegativeInteger
PartialV ==> Union(Col, "failed")
Both ==> Record(particular: PartialV, basis: List Col)
LSMP ==> LinearSystemMatrixPackage(F, Row, Col, M)

Cat ==> with
solve : (M, Col) -> Both
++ solve(A,B) finds a particular solution of the system \(AX = B\)
++ and a basis of the associated homogeneous system \(AX = 0\).
solve : (LL, Col) -> Both
++ solve(A,B) finds a particular solution of the system \(AX = B\)
++ and a basis of the associated homogeneous system \(AX = 0\).
solve : (M, List Col) -> List Both
++ solve(A,LB) finds a particular soln of the systems \(AX = B\)
++ and a basis of the associated homogeneous systems \(AX = 0\)
++ where B varies in the list of column vectors LB.
solve : (LL, List Col) -> List Both
++ solve(A,LB) finds a particular soln of the systems \(AX = B\)
++ and a basis of the associated homogeneous systems \(AX = 0\)
++ where B varies in the list of column vectors LB.

particularSolution: (M, Col) -> PartialV
++ particularSolution(A,B) finds a particular solution of the linear
++ system \(AX = B\).
hasSolution?: (M, Col) -> Boolean
++ hasSolution?(A,B) tests if the linear system \(AX = B\)
++ has a solution.
rank : (M, Col) -> N
++ rank(A,B) computes the rank of the complete matrix \(\text{spad}((A|B))\)
++ of the linear system \(AX = B\).

Capsule ==> add
solve(m : M, c: Col): Both == solve(m,c)$LSMP
solve(ll : LL, c: Col): Both == solve(matrix(ll)$M,c)$LSMP
solve(m : M, l : List Col): List Both == solve(m, l)$LSMP
solve(ll : LL, l : List Col): List Both == solve(matrix(ll)$M, l)$LSMP
particularSolution (m : M, c : Col): PartialV == particularSolution(m, c)$LSMP
hasSolution?(m :M, c : Col): Boolean == hasSolution?(m, c)$LSMP
rank(m : M, c : Col): N == rank(m, c)$LSMP
package LSPP LinearSystemPolynomialPackage

---

--- LinearSystemPolynomialPackage.input ---

)set break resume
)sys rm -f LinearSystemPolynomialPackage.output
)spool LinearSystemPolynomialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LinearSystemPolynomialPackage

--R LinearSystemPolynomialPackage(R: IntegralDomain,E: OrderedAbelianMonoidSup,OV: OrderedSet,P: PolynomialCategory(R,E,OV)) is a package constructor

--R Abbreviation for LinearSystemPolynomialPackage is LSPP

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LSPP

--R----------------------------- Operations --------------------------------

--R linSolve : (List(P),List(OV)) -> Record(particular: Union(Vector(Fraction(P)),”failed”),basis: List(Vector(Fraction(P))))

--E 1

)spool
)lisp (bye)

---

--- LinearSystemPolynomialPackage.help ---

====================================================================
LinearSystemPolynomialPackage examples
====================================================================

This package finds the solutions of linear systems presented as a list of polynomials.

See Also:
o )show LinearSystemPolynomialPackage

---
LinearSystemPolynomialPackage (LSPP)

Exports:
linSolve

— package LSPP LinearSystemPolynomialPackage —

)abbrev package LSPP LinearSystemPolynomialPackage
++ Author:  P.Gianni
++ Date Created: Summer 1985
++ Date Last Updated: Summer 1993
++ Description:
++ This package finds the solutions of linear systems presented as a
++ list of polynomials.

LinearSystemPolynomialPackage(R, E, OV, P): Cat == Capsule where
  R : IntegralDomain
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  P : PolynomialCategory(R,E,OV)

F ==> Fraction P
NNI ==> NonNegativeInteger
V ==> Vector
M ==> Matrix
Soln ==> Record(particular: Union(V F, "failed"), basis: List V F)

Cat == with
  linSolve: (List P, List OV) -> Soln
    ++ linSolve(lp,lvar) finds the solutions of the linear system
    ++ of polynomials lp = 0 with respect to the list of symbols lvar.

Capsule == add

    ---- Local Functions ----

    poly2vect:  (P, List OV)  -> Record(coefvec: V F, reductum: F)
intoMatrix: (List P, List OV) -> Record(mat: M F, vec: V F)

poly2vect(p : P, vs : List OV) : Record(coefvec: V F, reductum: F) ==
coops := new(#vs, 0)$(V F)
for v in vs for i in 1.. while p ^= 0 repeat
  u := univariate(p, v)
degree u = 0 => "next v"
  coops.i := (c := leadingCoefficient u) :: F
  p := p - monomial(c,v, 1)
[coops, p :: F]

intoMatrix(ps : List P, vs : List OV) : Record(mat: M F, vec: V F) ==
m := zero(#ps, #vs)$M(F)
v := new(#ps, 0)$V(F)
for p in ps for i in 1.. repeat
  totalDegree(p,vs) > 1 => error "The system is not linear"
  r := poly2vect(p,vs)
m:=setRow_!(m,i,r.coefvec)
v.i := - r.reductum
[m, v]

linSolve(ps, vs) ==
r := intoMatrix(ps, vs)
solve(r.mat, r.vec)$LinearSystemMatrixPackage(F,V F, V F, M F)

package LGROBP LinGroebnerPackage
chapter 13. chapter l

)set message auto off
)clear all

--S 1 of 1
)show LinGroebnerPackage
--R
--R LinGroebnerPackage(lv: List(Symbol),F: GcdDomain) is a package constructor
--R Abbreviation for LinGroebnerPackage is LGROBP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LGROBP
--R
--R------------------------------- Operations --------------------------------
--R anticoord : (List(F),DistributedMultivariatePolynomial(lv,F),List(DistributedMultivariatePolynomial(lv,F))) -> DistributedMultivariatePolynomial(lv,F)
--R choosemon : (DistributedMultivariatePolynomial(lv,F),List(DistributedMultivariatePolynomial(lv,F))) -> DistributedMultivariatePolynomial(lv,F)
--R computeBasis : List(HomogeneousDistributedMultivariatePolynomial(lv,F)) -> List(HomogeneousDistributedMultivariatePolynomial(lv,F))
--R coord : (HomogeneousDistributedMultivariatePolynomial(lv,F),List(HomogeneousDistributedMultivariatePolynomial(lv,F),List(DistributedMultivariatePolynomial(lv,F),List(HomogeneousDistributedMultivariatePolynomial(lv,F))) -> List(HomogeneousDistributedMultivariatePolynomial(lv,F))
--R groebgen : List(DistributedMultivariatePolynomial(lv,F)) -> Record(glbase: List(DistributedMultivariatePolynomial(lv,F),glval: List(Integer))
--R intcompBasis : (OrderedVariableList(lv),List(HomogeneousDistributedMultivariatePolynomial(lv,F),List(HomogeneousDistributedMultivariatePolynomial(lv,F))) -> List(HomogeneousDistributedMultivariatePolynomial(lv,F))
--R linGenPos : List(HomogeneousDistributedMultivariatePolynomial(lv,F)) -> Record(gblist: List(DistributedMultivariatePolynomial(lv,F),gvlist: List(Integer))
--R minPol : (List(HomogeneousDistributedMultivariatePolynomial(lv,F),List(HomogeneousDistributedMultivariatePolynomial(lv,F),OrderedVariableList(lv))) -> HomogeneousDistributedMultivariatePolynomial(lv,F)
--R minPol : (List(HomogeneousDistributedMultivariatePolynomial(lv,F),OrderedVariableList(lv))) -> HomogeneousDistributedMultivariatePolynomial(lv,F)
--R totolex : List(HomogeneousDistributedMultivariatePolynomial(lv,F)) -> List(DistributedMultivariatePolynomial(lv,F))
--R transform : DistributedMultivariatePolynomial(lv,F) -> HomogeneousDistributedMultivariatePolynomial(lv,F)
--R
--E 1

)spool
)lisp (bye)

-----------

— LinGroebnerPackage.help —

====================================================================
LinGroebnerPackage examples
====================================================================

Given a Groebner basis B with respect to the total degree ordering for a zero-dimensional ideal I, compute a Groebner basis with respect to the lexicographical ordering by using linear algebra.

See Also:
  o )show LinGroebnerPackage

-----------
LinGroebnerPackage (LGROBP)

Exports:
anticoord choosemon computeBasis coord groebgen
intcompBasis linGenPos minPol totolex transform

— package LGROBP LinGroebnerPackage —

)abbrev package LGROBP LinGroebnerPackage
++ Description:
++ Given a Groebner basis B with respect to the total degree ordering for
++ a zero-dimensional ideal I, compute
++ a Groebner basis with respect to the lexicographical ordering by using
++ linear algebra.

LinGroebnerPackage(lv,F) : C == T

where
Z ==> Integer
lv : List Symbol
F : GcdDomain

DP ==> DirectProduct(#lv,NonNegativeInteger)
DPoly ==> DistributedMultivariatePolynomial(lv,F)

HDP ==> HomogeneousDirectProduct(#lv,NonNegativeInteger)
HDPoly ==> HomogeneousDistributedMultivariatePolynomial(lv,F)

OV ==> OrderedVariableList(lv)
NNI ==> NonNegativeInteger
LVals ==> Record(gblist : List DPoly,gvlist : List Z)
VF ==> Vector F
VV ==> Vector NNI
MF ==> Matrix F
cLVars ==> Record(glbase:List DPoly,glval:List Z)

C == with
linGenPos : List HDPoly -> LVals
++ linGenPos \
undocumented

groebgen : List DPoly -> cLVars
++ groebgen \
undocumented
totolex : List HDPoly -> List DPoly
++ totolex \
undocumented

minPol : (List HDPoly, List HDPoly, OV) -> HDPoly
++ minPol \
undocumented

minPol : (List HDPoly, OV) -> HDPoly
++ minPol \
undocumented

computeBasis : List HDPoly -> List HDPoly
++ computeBasis \
undocumented

coord : (HDPoly, List HDPoly) -> VF
++ coord \
undocumented

anticoord : (List F, DPoly, List DPoly) -> DPoly
++ anticoord \
undocumented

intcompBasis : (OV, List HDPoly, List HDPoly) -> List HDPoly
++ intcompBasis \
undocumented

choosemon : (DPoly, List DPoly) -> DPoly
++ choosemon \
undocumented

transform : DPoly -> HDPoly
++ transform \
undocumented

T == add

import GroebnerPackage(F, DP, OV, DPoly)
import GroebnerPackage(F, HDP, OV, HDPoly)
import GroebnerInternalPackage(F, HDP, OV, HDPoly)
import GroebnerInternalPackage(F, DP, OV, DPoly)

lvar :=[variable(yx)::OV for yx in lv]

reduceRow(M:MF, v : VF, lastRow: Integer, pivots: Vector(Integer)) : VF ==
a1:F := 1
b:F := 0
dim := #v
for j in 1..lastRow repeat -- scan over rows
mj := row(M, j)
k := pivots(j)
b := mj.k
vk := v.k
for kk in 1..(k-1) repeat
v(kk) := ((-b*v(kk)) exquo a1) :: F
for kk in k..dim repeat
v(kk) := ((vk*mj(kk)-b*v(kk)) exquo a1)::F
a1 := b
v

rRedPol(f:HDPoly, B:List HDpoly):Record(poly:HDpoly, mult:F) ==
gm := redPo(f,B)
gm.poly = 0 => gm
gg := reductum(gm.poly)
ggm := rRedPol(gg,B)
[ggm.mult*(gm.poly - gg) + ggm.poly, ggm.mult*gm.mult]

----- transform the total basis B in lex basis -----
totolex(B : List HDpoly) : List Dpoly ==
result:List Dpoly :=[]
ltresult:List Dpoly :=[]
vBasis:= computeBasis B
nBasis:List Dpoly :=[1$DPoly]
ndim:=(#vBasis)::PositiveInteger
ndim1:NNI:=ndim+1
lm:VF
linmat:MF:=zero(ndim,2*ndim+1)
linmat(1,1):=1$F
linmat(1,ndim1):=1
pivots:Vector Integer := new(ndim,0)
pivots(1) := 1
firstmon:DPoly:=1$DPoly
ofirstmon:DPoly:=1$DPoly
orecfmon:Record(poly:HDpoly, mult:F) := [1,1]
i:NNI:=2
while (firstmon:=choosemon(firstmon,ltresult))"failed" repeat
  if (v:=firstmon exquo ofirstmon) case "failed" then
    recfmon:=rRedPol(transform firstmon,B)
  else
    recfmon:=rRedPol(transform(v:DPoly) *orecfmon.poly,B)
    recfmon.poly := recfmon.poly * orecfmon.poly
    recfmon.mult := recfmon.mult * orecfmon.mult
    cc := gcd(content recfmon.poly, recfmon.mult)
    recfmon.poly := (recfmon.poly exquo cc)::HDpoly
    recfmon.mult := (recfmon.mult exquo cc)::F
    veccoef:VF:=coord(recfmon.poly,vBasis)
ofirstmon:=firstmon
    orecfmon := recfmon
    lm:=zero(2*ndim+1)
    for j in 1..ndim repeat lm(j):=veccoef(j)
    lm(ndim1):=recfmon.mult
    lm := reduceRow(linmat, lm, i-1, pivots)
    if i=ndim1 then j:=ndim1
    else
      j:=1
      while lm(j) = 0 and j< ndim1 repeat j:=j+1
    if j=ndim1 then
      cordlist:List F:=[lm(k) for k in ndim1..ndim1+(#nBasis)]
antc:=+[/c*b for c in reverse cordlist]
for b in concat(firstmon,nBasis)]
  antc:=primitivePart antc
  result:=concat(antc,result)
  lttresult:=concat(antc-reductum antc,ltresult)
else
  pivots(i) := j
  setRow_!(linmat,i,lm)
  i:=i+1
  nBasis:=cons(firstmon,nBasis)
result

---- Compute the univariate polynomial for x
---- oldBasis is a total degree Groebner basis
minPol(oldBasis:List HDPoly,x:OV) :HDPoly ==
  algBasis:= computeBasis oldBasis
  minPol(oldBasis,algBasis,x)

---- Compute the univariate polynomial for x
---- oldBasis is total Groebner, algBasis is the basis as algebra
minPol(oldBasis:List HDPoly,algBasis:List HDPoly,x:OV) :HDPoly ==
  nvp:HDPoly:=x::HDPoly
  f:=1$HDPoly
  omult:F :=1
  ndim:=(#algBasis)::PositiveInteger
  ndim1:NNI:=ndim+1
  lm:VF
  linmat:MF:=zero(ndim,2*ndim+1)
  linmat(1,1):=1$F
  linmat(1,ndim1):=1
  pivots:Vector Integer := new(ndim,0)
  pivots(1) := 1
  for i in 2..ndim1 repeat
    recf:=rRedPol(f*nvp,oldBasis)
    omult := recf.mult * omult
    f := recf.poly
    cc := gcd(content f, omult)
    f := (f exquo cc)::HDPoly
    omult := (omult exquo cc)::F
    veccoef:VF:=coord(f,algBasis)
    lm:=zero(2*ndim+1)
    for j in 1..ndim repeat lm(j) := veccoef(j)
    lm(ndim+1):=omult
    lm := reduceRow(linmat, lm, i-1, pivots)
    j:=1
    while lm(j)=0 and j<ndim1 repeat j:=j+1
    if j=ndim1 then return
    g:HDPoly:=0
    for k in ndim1..2*ndim+1 repeat
      g:=g+lm(k) * nvp**((k-ndim1):NNI)
  primitivePart g
pivots(i) := j
setRow!(linmat,i,lm)

----- transform a DPoly in a HDPoly -----  
transform(dpol:DPoly) : HDPoly ==
dpol=0 => 0$HDPoly
monomial(leadingCoefficient dpol,
directProduct(degree(dpol)::VV)$HDP)$HDPoly +
transform(reductum dpol)

----- compute the basis for the vector space determined by B -----  
computeBasis(B:List HDPoly) : List HDPoly ==
mB:List HDPoly:=[monomial(1$F,degree f)$HDPoly for f in B]
result:List HDPoly := [1$HDPoly]
for var in lvar repeat
  part:=intcompBasis(var,result,mB)
  result:=concat(result,part)
result

----- internal function for computeBasis -----  
intcompBasis(x:OV,lr:List HDPoly,mB : List HDPoly):List HDPoly ==
  lr=[] => lr
  part:List HDPoly :=[]
  for f in lr repeat
    g:=x::HDPoly * f
    if redPo(g,mB).poly^=0 then part:=concat(g,part)
  concat(part,intcompBasis(x,part,mB))

----- coordinate of f with respect to the basis B -----  
----- f is a reduced polynomial -----  
coord(f:HDPoly,B:List HDPoly) : VF ==
  ndim := #B
  vv:VF:=new(ndim,0$F)$VF
  while f^=0 repeat
    rf := reductum f
    lf := f-rf
    lcf := leadingCoefficient f
    i:Z:=position(monomial(1$F,degree lf),B)
    vv.i:=lcf
    f := rf
  vv

----- reconstruct the polynomial from its coordinate -----  
anticoord(vv:List F,mf:DPoly,B:List DPoly) : DPoly ==
  for f in B for c in vv repeat (mf:=mf-c*f)
mf

----- choose the next monom -----  
choosemon(mf:DPoly,nB:List DPoly) : DPoly ==
nB = [] => ((lvar.last)::DPoly)*mf
for x in reverse lvar repeat
  xx:=x :DPoly
  mf:=xx*mf
  if redPo(mf,nB).poly ^= 0 then return mf
  dx := degree(mf,x)
  mf := (mf exquo (xx ** dx))::DPoly
mf

----- put B in general position, B is Groebner -----
linGenPos(B : List HDPoly) : LVals ==

result:List DPoly :=[]
lresult:List DPoly :=[]
vBasis:= computeBasis B
nBasis:List DPoly :=[1$DPoly]
dim:=#vBasis : PositiveInteger
ndim1:NNI:=ndim+1
lm:VF
linmat:MF:=zero(ndim,2*ndim+1)
linmat(1,1):=1$F
linmat(1,ndim1):=1
pivots:Vector Integer := new(ndim,0)
pivots(1) := 1
i:NNI:=2
rval:List Z :=[]
for ii in 1..(#lvar-1) repeat
  c:=0
  while c=0 repeat c:=random()$Z rem 11
  rval:=concat(c,rval)
  nval:DPoly := (last.lvar)::DPoly -
  (/*[r*(vv)::DPoly for r in rval for vv in lvar])
  firstmon:DPoly:=i$DPoly
  ofirstmon:DPoly:=i$DPoly
  orecfmon:Record(poly:HDPoly, mult:F) := [1,1]
  lx:= lvar.last
  while (firstmon:=choosemon(firstmon,lresult)) ^=1 repeat
    if (v:=firstmon exquo ofirstmon) case "failed" then
      orecfmon:=rRedPol(transform(eval(firstmon,lx,nval)),B)
      orecfmon.mult := orecfmon.mult * orecfmon.mult
      cc := gcd(content orecfmon.poly, orecfmon.mult)
      orecfmon.poly := (orecfmon.poly exquo cc)::HDPoly
      orecfmon.mult := (orecfmon.mult exquo cc)::F
      veccoef:VF:=coord(orecfmon.poly,vBasis)
      ofirstmon:=firstmon
      orecfmon := orecfmon
      lm:=zero(2*ndim+1)
    for j in 1..ndim repeat lm(j):=veccoef(j)
    lm(ndim+i):=orecfmon.mult
    lm := reduceRow(linmat, lm, i-1, pivots)
j:=1
while lm(j) = 0 and j<ndim1 repeat j:=j+1
if j=ndim1 then
   cordlist:List F:=[lm(j) for j in ndim1..ndim1+(#nBasis)]
   antc:+=/[c*b for c in reverse cordlist
   for b in concat(firstmon,nBasis)]
   result:=concat(primitivePart antc,result)
   lresult:=concat(antc-reductum antc,lresult)
else
   pivots(i) := j
   setRow_!(linmat,i,lm)
   i:=i+1
   nBasis:=concat(firstmon,nBasis)
[result,rval]$LVals

----- given a basis of a zero-dimensional ideal,
----- performs a random change of coordinates
----- computes a Groebner basis for the lex ordering

    groebgen(L:List DPoly) : cLVars ==
    xn:=lvar.last
    val := xn::DPoly
    nvar1:NNI:=(#lvar-1):NNI
    ll: List Z :=[random()$Z rem 11 for i in 1..nvar1]
    val:=val+ +/[ll.i*(lvar.i)::DPoly for i in 1..nvar1]
    LL:=[elt(univariate(f,xn),val) for f in L]
    LL:= groebner(LL)
    [LL,ll]$cLVars

— LGROBP.dotabb —

"LGROBP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LGROBP"]
"DIRPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"LGROBP" -> "DIRPCAT"
"LGROBP" -> "PFECAT"

package LOP LinesOpPack

— LinesOpPack.input —

)set break resume
### LinesOpPack.help

A package that exports several linear algebra operations over lines of matrices. Part of the PAFF package.

See Also:
- show LinesOpPack
LinesOpPack (LOP)

Exports:
- quotVecSpaceBasis
- reduceLineOverLine
- reduceRow
- reduceRowOnList
- rowEchWoZeroLines
- rowEchWoZeroLinesWOVectorise

package LOP LinesOpPack

)abbrev package LOP LinesOpPack
++ Authors: G. Hache
++ Date Created: 21 sept 1994
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that exports several linear algebra operations over lines
++ of matrices. Part of the PAFF package.

LinesOpPack(K):P==T where

K:Field

P==> with
- rowEchWoZeroLinesWOVectorise: Matrix(K) -> Matrix(K)
- rowEchWoZeroLines: Matrix(K) -> Matrix(K)
- reduceRow: List(List(K)) -> List(List(K))

++ reduceRow: if the input is considered as a matrix, the output would
++ be the row reduction matrix. It’s almost the rowEchelon form
++ except that no permutation of lines is performed.
- quotVecSpaceBasis: (List(List(K)),List(List(K))) -> List(List(K))

++ quotVecSpaceBasis(b1,b2) returns a basis of V1/V2 where
++ V1 and V2 are vector space with basis b1 and b2 resp. and
++ V2 is suppose to be include in V1; Note that if
++ it is not the case then it returs the basis of V1/W
++ where W = intersection of V1 and V2
- reduceLineOverLine: (List(K),List(K),K) -> List(K)

++ reduceLineOverLine(v1,v2,a) returns v1-a*v1 where
++ v1 and v2 are considered as vector space.
- reduceRowOnList: (List(K),List(List(K))) -> List(List(K))

++ reduceRowOnList(v,lvec) applies a row reduction on each of the
++ element of lv using v according to a pivot in v which is set to
++ be the first non null element in v.

T==> add
localRowEchelon: Matrix(K) -> Matrix(K)
localRowEchelon(m)==
  ^(K has PseudoAlgebraicClosureOfPerfectFieldCategory ) => rowEchelon m
  llm:List(List(K)):= listOfLists m
  l:= first llm
  maxT:= maxTower l
  lv := [vectorise(a,maxT)$K for a in l]
  subMatl := transpose matrix [entries(v) for v in lv]
  matl:= subMatl
  for l in rest llm repeat
    maxT:= maxTower l
    lv := [vectorise(a,maxT)$K for a in l]
    subMatl := transpose matrix [entries(v) for v in lv]
    matl:= vertConcat(matl,subMatl)
  rowEchelon matl

rowEchWoZeroLines(m)==
  mm:=localRowEchelon m
  ll:=listOfLists mm
  n:= # first ll
  lZero:=new(n pretend NonNegativeInteger,0)$List(K)
  llll:= [ l for l in ll | ^(lZero = l) ]
  empty?(llll) => matrix [lZero]
  matrix llll

rowEchWoZeroLinesWOVectorise(m)==
  mm:=rowEchelon m
  ll:=listOfLists mm
  n:= # first ll
  lZero:=new(n pretend NonNegativeInteger,0)$List(K)
  llll:= [ l for l in ll | ^(lZero = l) ]
  empty?(llll) => matrix [lZero]
  matrix llll

quotVecSpaceBasis(l2,l1)==
  redBasis:=reduceRow(concat(l1,l2))
  tempRes:=rest(redBasis,#l1)
  allZero:=new(#l1.1,0$K)
  [l for l in tempRes | ^(l=allZero)]

reduceRowOnList(line,listOfLine)==
  frsNonNul:Integer:=position(^zero?(#1),line)
  ^(frsNonNul > 0) => listOfLine
  a:= line.frsNonNul
  inva:= inv a
  newLine:=[inva*c for c in line]
[reduceLineOverLine(newLine, 1, 1.frsNonNul) for 1 in listOfLine]

reduceLineOverLine(11, 12, b) ==
   [c2 - b*c1 for c2 in 12 for c1 in 11]

reduceRow(m: List(List(K))) ==
   n := #m
   mcopy := copy m
   newBottom := List(List(K))
   for i in 1..(n-1) repeat
      newBottom := reduceRowOnList(mcopy.i, [mcopy.j for j in (i+1)..<n])
      mcopy := concat([mcopy.k for k in 1..<i] :: List(List(K)), newBottom)
   mcopy

---

— LOP.dotabb —

"LOP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LOP"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"LOP" -> "IVECTOR"

---

package LF LiouvillianFunction

— LiouvillianFunction.input —

)set break resume
)sys rm -f LiouvillianFunction.output
)spool LiouvillianFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show LiouvillianFunction
--R
--R LiouvillianFunction(R: Join(OrderedSet,IntegralDomain),F: Join(FunctionSpace(R),RadicalCategory,TranscendentalFunctionCategory)) is a package constructor
--R Abbreviation for LiouvillianFunction is LF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LF
--R
--R----------------------------------- Operations ---------------------------------
LiouvillianFunction (LF)

Exports:
  belong?  Ci  dilog  Ei  erf
  integral  li  integral  operator  Si
— package LF LiouvillianFunction —

)abbrev package LF LiouvillianFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 10 August 1994
++ Description:
++ This package provides liouvillian functions over an integral domain.

LiouvillianFunction(R, F):Exports == Implementation where
R:Join(OrderedSet, IntegralDomain)
F:Join(FunctionSpace R,RadicalCategory,TranscendentalFunctionCategory)

OP => BasicOperator
PR => Polynomial R
K => Kernel F
SE => Symbol
O => OutputForm
INP => InputForm
INV => error "Invalid argument"

SPECIALDIFF => "%specialDiff"
SPECIALDISP => "%specialDisp"
SPECIALINPUT => "%specialInput"
SPECIALEQUAL => "%specialEqual"

Exports => with
  belong? : OP -> Boolean
  ++ belong?(op) checks if op is Liouvillian
  operator: OP -> OP
  ++ operator(op) returns the Liouvillian operator based on op
  Ei : F -> F
  ++ Ei(f) denotes the exponential integral
  Si : F -> F
  ++ Si(f) denotes the sine integral
  Ci : F -> F
  ++ Ci(f) denotes the cosine integral
  li : F -> F
  ++ li(f) denotes the logarithmic integral
  erf : F -> F
  ++ erf(f) denotes the error function
  dilog : F -> F
  ++ dilog(f) denotes the dilogarithm
  fresnelS : F -> F
  ++ fresnelS(f) denotes the Fresnel integral S
  fresnelC : F -> F
  ++ fresnelC(f) denotes the Fresnel integral C
  integral: (F, SE) -> F
  ++ integral(f,x) indefinite integral of f with respect to x.
integral: (F, SegmentBinding F) -> F
++ integral(f, x = a..b) denotes the definite integral of f with
++ respect to x from spad{a} to b.

Implementation ==> add
iei : F -> F
isi : F -> F
ici : F -> F
ierf : F -> F
ili : F -> F
ili2 : F -> F
iint : List F -> F
eqint : (K,K) -> Boolean
dvint : (List F, SE) -> F
dvdint : (List F, SE) -> F
ddint : List F -> O
integrand : List F -> F

dummy := new()$SE :: F

opint := operator("integral":Symbol)$CommonOperators
opdint := operator("%defint":Symbol)$CommonOperators
opei := operator("Ei":Symbol)$CommonOperators
opl1 := operator("li":Symbol)$CommonOperators
opsi := operator("Si":Symbol)$CommonOperators
opci := operator("Ci":Symbol)$CommonOperators
opli2 := operator("dilog":Symbol)$CommonOperators
operf := operator("erf":Symbol)$CommonOperators
opfis := operator("fresnelS":Symbol)$CommonOperators
opfic := operator("fresnelC":Symbol)$CommonOperators

Si x == opsi x
Ci x == opci x
Ei x == opei x
erf x == operf x
li x == opl1 x
dilog x == opli2 x
fresnelS x == opfis x
fresnelC x == opfic x

belong? op == has?(op, "prim")
isI x == kernel(opsi, x)
ici x == kernel(opci, x)
ierf x == (zero? x => 0; kernel(operf, x))
-- ili2 x == (one? x => INV; kernel(opli2, x))
ili2 x == ((x = 1) => INV; kernel(opli2, x))
ifis(x:F):F == (zero? x => 0; kernel(opfis,x))
ific(x:F):F == (zero? x => 0; kernel(opfic,x))
integrand 1 == eval(first 1, retract(second 1)$K, third 1)
integral(f:F, x:SE) == opint [eval(f, k:=kernel(x)$K, dummy), dummy, k::F]
PACKAGE LF LIOUVILLIANFUNCTION

\[
iint 1 =
\begin{align*}
\text{zero? first } 1 & \Rightarrow 0 \\
\text{kernel}(\text{opint, } 1)
\end{align*}
\]

\[
ddint 1 =
\begin{align*}
\text{int}(\text{integrand}(1) \cdot 0 + \text{hconcat}("d" \cdot \text{SE} \cdot 0, \text{third}(1) \cdot 0), \\
\text{third}(\text{rest } 1) \cdot 0, \text{third}(\text{rest rest } 1) \cdot 0)
\end{align*}
\]

\[
eqint(k1, k2) =
\begin{align*}
a1 & := \text{argument } k1 \\
a2 & := \text{argument } k2 \\
\text{res} & := \text{operator } k1 = \text{operator } k2 \\
\text{if not } \text{res} \text{ then return } \text{res} \\
\text{res} & := a1 = a2 \\
\text{if } \text{res} \text{ then return } \text{res} \\
\text{res} & := (a1.3 = a2.3) \text{ and } (\text{subst}(a1.1, \text{retract}(a1.2) \cdot \text{K}), [a2.2]) = a2.1
\end{align*}
\]

\[
dvint(l, x) =
\begin{align*}
k & := \text{retract}(\text{second } 1) \cdot \text{K} \\
\text{differentiate}(\text{third } 1, x) & \cdot \text{integrand } l \\
+ \text{opint} \left[\text{differentiate}(\text{first } 1, x), \text{second } 1, \text{third } 1\right]
\end{align*}
\]

\[
dvdint(l, x) =
\begin{align*}
x & = \text{retract}(y := \text{third } 1) \cdot \text{SE} \Rightarrow 0 \\
k & := \text{retract}(d := \text{second } 1) \cdot \text{K} \\
\text{differentiate}(h := \text{third rest rest } 1, x) & \cdot \text{eval}(f := \text{first } 1, k, h) \\
- \text{differentiate}(g := \text{third rest } 1, x) & \cdot \text{eval}(f, k, g) \\
+ \text{opdint} \left[\text{differentiate}(f, x), d, y, g, h\right]
\end{align*}
\]

\[
\text{integral}(f:F, s: \text{SegmentBinding } F) =
\begin{align*}
x & := \text{kernel}(\text{variable } s) \cdot \text{K} \\
\text{opdint} \left[\text{eval}(f, x, \text{dummy}), \text{dummy}, x \cdot F, \text{lo } \text{segment } s, \text{hi } \text{segment } s\right]
\end{align*}
\]

\[
\text{ili } x =
\begin{align*}
x & = 1 \Rightarrow \text{INV} \\
\text{is?}(x, "\text{exp}" \cdot \text{Symbol}) & \Rightarrow \text{Ei } \text{first argument}(\text{retract}(x) \cdot \text{K}) \\
\text{kernel}(\text{opli, } x)
\end{align*}
\]

\[
\text{iei } x =
\begin{align*}
x & = 0 \Rightarrow \text{INV} \\
\text{is?}(x, "\text{log}" \cdot \text{Symbol}) & \Rightarrow \text{li } \text{first argument}(\text{retract}(x) \cdot \text{K}) \\
\text{kernel}(\text{opei, } x)
\end{align*}
\]

\[
\text{operator } op =
\begin{align*}
\text{is?}(\text{op, "integral" \cdot Symbol}) & \Rightarrow \text{opint} \\
\text{is?}(\text{op, "%defint" \cdot Symbol}) & \Rightarrow \text{opdint} \\
\text{is?}(\text{op, "Ei" \cdot Symbol}) & \Rightarrow \text{opei} \\
\text{is?}(\text{op, "Si" \cdot Symbol}) & \Rightarrow \text{opsi}
\end{align*}
\]
is?(op, "Ci"::Symbol) => opci
is?(op, "li"::Symbol) => opli
is?(op, "erf"::Symbol) => operf
is?(op, "dilog"::Symbol) => opli2
is?(op, "fresnelC"::Symbol) => opfis
is?(op, "fresnelS"::Symbol) => opfic
error "Not a Liouvillian operator"

evaluate(opei, iei)$BasicOperatorFunctions1(F)
evaluate(opli, ili)
evaluate(opsi, isi)
evaluate(opci, ici)
evaluate(operf, ierf)
evaluate(opli2, ili2)
evaluate(opfis, ifis)
evaluate(opfic, ific)
evaluate(opint, iint)
derivative(opsi, (z1:F):F +-> sin(z1) / z1)
derivative(opci, (z1:F):F +-> cos(z1) / z1)
derivative(opei, (z1:F):F +-> exp(z1) / z1)
derivative(opli, (z1:F):F +-> inv log(z1))
derivative(operf, (z1:F):F +-> 2 * exp(-(z1**2)) / sqrt(pi()))
derivative(opli2, (z1:F):F +-> log(z1) / (1 - z1))
derivative(opfis, (z1:F):F +-> sin(z1**2))
derivative(opfic, (z1:F):F +-> cos(z1**2))
setProperty(opint,SPECIALEQUAL,eqint@((K,K) -> Boolean) pretend None)
setProperty(opint,SPECIALDIFF,dvint@((List F,SE) -> F) pretend None)
setProperty(opdint,SPECIALDIFF,dvint@((List F,SE)->F) pretend None)
setProperty(opdint, SPECIALDISP, ddint@(List F -> O) pretend None)

if R has ConvertibleTo INP then
  inint : List F -> INP
  indint: List F -> INP
  pint : List INP -> INP

  pint l  == convert concat(convert("integral"::SE)@INP, l)
inint l  ==
r2:= convert(
    [convert("::"::SE)@INP,
     convert(third l)@INP,
     convert("Symbol"::SE)@INP]@List INP)@INP
  pint [convert(integrand l)@INP, r2]

  indint l  ==
pint [convert(integrand l)@INP,
     convert concat(convert("::"::SE)@INP,
                [convert(third l)@INP,
                 convert concat(convert("SEGMENT"::SE)@INP,
                                 [convert(third rest l)@INP,]@INP,]@INP,]@INP,]@INP,
convert(third rest rest l)@INP])]

setProperty(opint, SPECIALINPUT, inint@((List F -> INP) pretend None))
setProperty(opdint, SPECIALINPUT, indint@((List F -> INP) pretend None))

<table>
<thead>
<tr>
<th>LF.dotabb</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;LF&quot; [color=&quot;#FF4488&quot;,href=&quot;bookvol10.4.pdf#nameddest=LF&quot;]</td>
</tr>
<tr>
<td>&quot;FS&quot; [color=&quot;#4488FF&quot;,href=&quot;bookvol10.2.pdf#nameddest=FS&quot;]</td>
</tr>
<tr>
<td>&quot;LF&quot; -&gt; &quot;FS&quot;</td>
</tr>
</tbody>
</table>

package LIST2 ListFunctions2

--- ListFunctions2.input ---

)set break resume
)sys rm -f ListFunctions2.output
)spool ListFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ListFunctions2

--R ListFunctions2(A: Type,B: Type) is a package constructor
--R Abbreviation for ListFunctions2 is LIST2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LIST2

--R----------------------------------------------------------- Operations -------------------------------
--R map : ((A -> B),List(A)) -> List(B)
--R reduce : (((A,B) -> B),List(A),B) -> B
--R scan : (((A,B) -> B),List(A),B) -> List(B)

--E 1
)spool
)lisp (bye)
ListFunctions2 (LIST2)

Exports:
  map  reduce  scan

— package LIST2 ListFunctions2 —

)abbrev package LIST2 ListFunctions2
++ Description:
++ \texttt{ListFunctions2} implements utility functions that
++ operate on two kinds of lists, each with a possibly different
++ type of element.

ListFunctions2(A:Type, B:Type): public == private where
  LA ==> List A
  LB ==> List B
  O2 ==> FiniteLinearAggregateFunctions2(A, LA, B, LB)
public ==> with
  scan: ((A, B) -> B, LA, B) -> LB
++ scan(fn,u,ident) successively uses the binary function
++ \spad{fn} to reduce more and more of list \spad{u}.
++ \spad{ident} is returned if the \spad{u} is empty.
++ The result is a list of the reductions at each step. See
++ \spadfun{reduce} for more information. Examples:
++ \spad{scan(fn,[1,2],0) = [fn(2,fn(1,0)),fn(1,0)]} and
++ \spad{scan(*,[2,3],1) = [2 * 1, 3 * (2 * 1)]}.
reduce: ((A, B) -> B, LA, B) -> B
++ reduce(fn,u,ident) successively uses the binary function
++ \spad{fn} on the elements of list \spad{u} and the result
++ of previous applications. \spad{ident} is returned if the
++ \spad{u} is empty. Note the order of application in
++ the following examples:
++ \spad{reduce(fn,[1,2,3],0) = fn(3,fn(2,fn(1,0))))} and
++ \spad{reduce(*,[2,3],1) = 3 * (2 * 1)}.
map: (A -> B, LA) -> LB
++ map(fn,u) applies \spad{fn} to each element of
++ list \spad{u} and returns a new list with the results.
++ For example \spad{map(square,[1,2,3]) = [1,4,9]}.

private ==> add
  map(f, l) == map(f, l)$O2
  scan(f, l, b) == scan(f, l, b)$O2
  reduce(f, l, b) == reduce(f, l, b)$O2

package LIST3 ListFunctions3

— ListFunctions3.input —

)set break resume
)sys rm -f ListFunctions3.output
ListFunctions3 implements utility functions that operate on three kinds of lists, each with a possibly different type of element.

See Also:
- )show ListFunctions3
ListFunctions3 (LIST3)

Exports:
map

— package LIST3 ListFunctions3 —

)abbrev package LIST3 ListFunctions3
++ Description:
++ \spadtype{ListFunctions3} implements utility functions that
++ operate on three kinds of lists, each with a possibly different
++ type of element.

ListFunctions3(A:Type, B:Type, C:Type): public == private where
  LA ==> List A
  LB ==> List B
  LC ==> List C

public ==> with
  map: ( (A,B)->C, LA, LB) -> LC
    ++ map(fn,list1, u2) applies the binary function \spad{fn}
    ++ to corresponding elements of lists \spad{u1} and \spad{u2}
    ++ and returns a list of the results (in the same order). Thus
    ++ \spad{map(/,[1,2,3],[4,5,6]) = [1/4,2/4,1/2]}. The computation
    ++ terminates when the end of either list is reached. That is,
    ++ the length of the result list is equal to the minimum of the
    ++ lengths of \spad{u1} and \spad{u2}.

private ==> add
  map(fn : (A,B) -> C, la : LA, lb : LB): LC ==
    empty?(la) or empty?(lb) => empty()$LC
    concat(fn(first la, first lb), map(fn, rest la, rest lb))
package LIST2MAP ListToMap

-- ListToMap.input --

)set break resume
)sys rm -f ListToMap.output
)spool ListToMap.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show ListToMap
--R
--R ListToMap(A: SetCategory,B: Type) is a package constructor
--R Abbreviation for ListToMap is LIST2MAP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LIST2MAP
--R
--R----------------------------------- Operations -----------------------------------
--R match : (List(A),List(B),A) -> B  match : (List(A),List(B),A,B) -> B
--R match : (List(A),List(B)) -> (A -> B)
--R match : (List(A),List(B),B) -> (A -> B)
--R match : (List(A),List(B),(A -> B)) -> (A -> B)
--R match : (List(A),List(B),A,(A -> B)) -> B
--E 1

)spool
)lisp (bye)

-- ListToMap.help --
ListToMap allows mappings to be described by a pair of lists of equal lengths. The image of an element \( x \), which appears in position \( n \) in the first list, is then the \( n \)-th element of the second list. A default value or default function can be specified to be used when \( x \) does not appear in the first list. In the absence of defaults, an error will occur in that case.

See Also:
- \( \text{show ListToMap} \)

---

ListToMap (LIST2MAP)

Exports:
- match

---

)abbrev package LIST2MAP ListToMap
++ Author: Manuel Bronstein
++ Date Created: 22 Mar 1988
++ Change History:11 Oct 1989
++ Description:
++ \spad{ListToMap} allows mappings to be described by a pair of
++ lists of equal lengths. The image of an element \spad{x},
++ which appears in position \spad{n} in the first list, is then
++ \spad{n}\-th element of the second list. A default value or
++ default function can be specified to be used when \spad{x}
++ does not appear in the first list. In the absence of defaults,
++ an error will occur in that case.

ListToMap(A: SetCategory, B: Type): Exports == Implementation where
  LA ==> List A
  LB ==> List B
  AB ==> (A -> B)

Exports ==> with
  match: (LA, LB) -> AB
    ++ match(la, lb) creates a map with no default source or target values
    ++ defined by lists la and lb of equal length.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Error: if la and lb are not of equal length.
    ++ Note that when this map is applied, an error occurs when
    ++ applied to a value missing from la.
  match: (LA, LB, A) -> B
    ++ match(la, lb, a) creates a map
    ++ defined by lists la and lb of equal length, where \spad{a} is used
    ++ as the default source value if the given one is not in \spad{la}.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Error: if la and lb are not of equal length.
  match: (LA, LB, B) -> AB
    ++ match(la, lb, b) creates a map
    ++ defined by lists la and lb of equal length, where \spad{b} is used
    ++ as the default target value if the given function argument is
    ++ not in \spad{la}.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Error: if la and lb are not of equal length.
  match: (LA, LB, A, B) -> B
    ++ match(la, lb, a, b) creates a map
    ++ defined by lists la and lb of equal length.
    ++ and applies this map to a.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Argument b is the default target value if a is not in la.
    ++ Error: if la and lb are not of equal length.
  match: (LA, LB, AB) -> AB
    ++ match(la, lb, f) creates a map
    ++ defined by lists la and lb of equal length.
    ++ The target of a source value \spad{x} in la is the
    ++ value y with the same index lb.
    ++ Argument \spad{f} is used as the
    ++ function to call when the given function argument is not in
    ++ \spad{la}.
    ++ The value returned is f applied to that argument.
  match: (LA, LB, A, AB) -> B
    ++ match(la, lb, a, f) creates a map
++ defined by lists la and lb of equal length.
++ and applies this map to a.
++ The target of a source value \texttt{\spad{x}} in la is the
++ value y with the same index lb.
++ Argument \texttt{\spad{f}} is a default function to call if a is not in la.
++ The value returned is then obtained by applying f to argument a.

Implementation ==> add

\begin{verbatim}
match(la, lb)  == (z1:A):B +-> match(la, lb, z1)
machine(la:LA, lb:LB, a:A)  == lb.position(a, la)
machine(la:LA, lb:LB, b:B)  == (z1:A):B +-> match(la, lb, z1, b)
machine(la:LA, lb:LB, f:AB)  == (z1:A):B +-> match(la, lb, z1, f)
machine(la:LA, lb:LB, a:A, b:B) ==
  (p := position(a, la)) < minIndex(la) => b
  lb.p

match(la:LA, lb:LB, a:A, f:AB) ==
  (p := position(a, la)) < minIndex(la) => f a
  lb.p
\end{verbatim}

package LPARSPT LocalParametrizationOfSimplePoint Package

— LocalParametrizationOfSimplePointPackage.input —

)set break resume
)sys rm -f LocalParametrizationOfSimplePointPackage.output
)spool LocalParametrizationOfSimplePointPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show LocalParametrizationOfSimplePointPackage
--R
--R LocalParametrizationOfSimplePointPackage(K: Field,symb: List(Symbol),PolyRing: PolynomialCategoryK -> ProjectiveSpaceCategory(K),PCS: LocalPowerSeriesCategory(K),Plc: PlacesCategory(K,PCS)) is a package constructor
--R Abbreviation for LocalParametrizationOfSimplePointPackage is LPARSPT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LPARSPT
--R
--R-------------------------------------- Operations --------------------------------------
--R pointDominateBy : Plc -> ProjPt printInfo : Boolean -> Boolean
--R printInfo : () -> Boolean
--R localParamOfSimplePt : (ProjPt,PolyRing,Integer) -> List(PCS)
--R localize : (PolyRing,ProjPt,PolyRing,Integer) -> Record(fnc: PolyRing,crv: PolyRing,chart: List(Integer))
--R pointToPlace : (ProjPt,PolyRing) -> Plc
--R
--E 1

)spool
)lisp (bye)

— LocalParametrizationOfSimplePointPackage.help —

====================================================================
LocalParametrizationOfSimplePointPackage examples
====================================================================

This package is part of the PAFF package

See Also:
  o )show LocalParametrizationOfSimplePointPackage
LocalParametrizationOfSimplePointPackage (LPARSPT)

Exports:
pointDominateBy printInfo localParamOfSimplePt localize pointToPlace

-- package LPARSPT LocalParametrizationOfSimplePointPackage

)abbrev package LPARSPT LocalParametrizationOfSimplePointPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This package is part of the PAFF package
LocalParametrizationOfSimplePointPackage(K,symb,PolynRing,E,ProjPt,PCS,Plc):_
Exports == Implementation where

K:Field
symb: List(Symbol)

E : DirectProductCategory(#symb,NonNegativeInteger)

OV ==> OrderedVariableList(symb)

PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)
Plc : PlacesCategory(K,PCS)

PI ==> PositiveInteger
NNI ==> NonNegativeInteger
PPFC1 ==> PolynomialPackageForCurve(K,PolynRing,E,#symb,ProjPt)
PackPoly ==> PackageForPoly(K,PolynRing,E,#symb)

UP ==> SparseUnivariatePolynomial(K)
UPUP ==> SparseUnivariatePolynomial(UP)

Exports ==> with

printInfo: Boolean -> Boolean
++ printInfo(b) set a flag such that when true (b <- true) prints
++ some information during some critical computation.

printInfo: () -> Boolean
++ returns the value of the \spad{printInfo} flag.

pointToPlace: (ProjPt,PolyRing) -> Plc
++ pointToPlace(pt,pol) takes for input a simple point pt on the curve
++ defined by pol and set the local parametrization of the point.

localParamOfSimplePt: (ProjPt,PolyRing,Integer) -> List PCS
++ localParamOfSimplePt(pt,pol,n) computes the local parametrization
++ of the simple point pt on the curve defined by pol. This local
++ parametrization is done according to the standard open affine
++ plane set by n

pointDominateBy : Plc -> ProjPt
++ pointDominateBy(pl) returns the projective point dominated
++ by the place pl.

localize: (PolyRing,ProjPt,PolyRing,Integer) -> _
++ localize(f,pt,crv,n) returns a record containing the polynomials f
++ and crv translate to the origin with respect to pt. The last
++ element of the records, consisting of three integers contains
++ information about the local parameter that will be used
++ (either x or y): the first integer correspond to the variable
++ that will be used as a local parameter.

Implementation ==> add
import PCS
import PolyRing
import PPFC1
import PackPoly

valuationAndMore: (UPUP,UPUP) -> _
++ valuationAndMore(f,crv) returns a record containing the value ord
++ of the function at the field element x and the coefficient of
++ the highest degree term in the polynomial f.

localize2: (PolyRing,ProjPt,PolyRing,Integer) -> _
++ localize2(f,pt,crv) returns a record containing the polynomials f
++ and crv translate to the origin with respect to pt. The last
++ element of the records, consisting of three integers contains
++ information about the local parameter that will be used
++ (either x or y): the first integer correspond to the variable
++ that will be used as a local parameter.

coerceToUPUP: (PolyRing,List Integer) -> UPUP

paramAtOrigin: (UPUP,UPUP,Integer) -> PCS

strictTransform: (UPUP,NNI) -> UPUP

translate: (UPUP,K) -> UPUP
constant: UPUP -> K
intCoord: UPUP -> K
localMultiplicity: UPUP -> NNI
mapDegree: (NNI,NNI,NNI) -> NNI
listVar:List(OV):= [index(i::PI)$OV for i in 1..#symb]
listMonoPols:List(PolyRing):=listVariable()

pointDominantBy(pl)==
lpl:List PCS:=localParam(pl)
empty? lpl => _
error "LPARSPT:pointDominantBy::parametrization of point not done yet"
lK:List K:= [ findCoef(s,0) for s in lpl]
projectivePoint(lK)

localParamOfSimplePt(pt,curve,nV)==
mult:NNI:=multiplicity(curve,pt,nV)
"one?(mult) => _
error "The point is not simple or is not on the curve !"
lcl:=[localize2(var,pt,curve,nV) for var in listMonoPols]
[paramAtOrigin(l.fnc2,l.crv2,0) for l in lcl]

pointToPlace(pt,curve)==
-- define the chart for strictTransform (of simple point)
nV:Integer:=lastNonNull pt
pth:=homogenize(pt,nV)
chart:List Integer:=[0,0,nV]
mult:NNI:=multiplicity(curve,pth,nV)
"one?(mult) =>
error "The point is not simple or is not on the curve"
-- create a place from the simple point. This is done by giving
-- a name to the place: in this case it is the coordinate of
-- the projective point.
lpth:List K:= pth :: List(K)
plc:Plc:=create(lpth)$Plc
"empty?(localParam(plc)) => plc
lcl:=[localize2(var,pt,curve,nV) for var in listMonoPols]
1Par:=[paramAtOrigin(l.fnc2,l.crv2,0) for l in lcl]
setParam!(plc,1Par)
dd:=degree pth
setDegree!(plc,dd)
plc

localVarForPrintInfo:Boolean:=false()$Boolean
printInfo()==localVarForPrintInfo
printInfo(flag) := localVarForPrintInfo := flag

mapDegree(n, mx, m) ==
  dd := (n + mx - m)
  dd < 0 => _
    error "LPARSPT:mapDegree called by PARAMP:strictTransform failed"
  dd pretend NNI

strictTransform(pol, m) ==
  zero?(pol) => 0
  tc := leadingCoefficient pol
  tk := degree pol
  newTc := mapExponents(mapDegree(#1, tk, m), tc)
  monomial(newTc, tk) $UPUP + strictTransform(reductum pol, m)

Y == monomial(1, 1) $UPUP

trY: (K, NonNegativeInteger) -> UPUP
trY(a, n) == (monomial(monomial(a, 0) $UP, 0) $UPUP + Y)**n

translate(pol, a) ==
  zero?(pol) => 0
  tc := leadingCoefficient pol
  tk := degree pol
  trY(a, tk) * tc + translate(reductum pol, a)

constant(pol) == coefficient(coefficient(pol, 0) $UPUP, 0) $UP

intCoord(pol) ==
  coefY := coefficient(coefficient(pol, 1) $UPUP, 0) $UP
  cnst := constant(pol)
  -cnst * inverse(coefY)

localMultiplicity(pol) ==
  zero?(pol) => error "Cannot compute the multiplicity for 0"
  redPol := reductum pol
  tc := leadingCoefficient pol
  tk := degree pol
  m := tk + minimumDegree(tc) $UP
  zero?(redPol) => m
  min(m, localMultiplicity(redPol))

coerceToUPUP(pol, chart) ==
  zero?(pol) => 0
  lExp := parts degree pol
  lCoef := leadingCoefficient pol
  expX := lExp(chart.1)
  expY := lExp(chart.2)
  monomial(monomial(lCoef, expX) $UP, expY) $UPUP + _
coerceToUPUP(reductum(pol),chart)

-- testing this function. See paramPack for original version.
valuationAndMore(f:UPUP,curve:UPUP)==
-- this function evaluate the function f at the origin
-- which must be a simple point on the curve define by "curve"
val:= constant(f)
zer?(val) => [0,val,f,curve]
sTrCurve:=strictTransform(curve,1)
slp:=intCoord sTrCurve
multPtf:Integer:= localMultiplicity(f) pretend Integer
sTrFnc:=strictTransform(f,multPtf pretend NNI)
newCurve:=translate(sTrFnc,slp)
val:= constant(f2)
[multPtf, val, f2, newCurve]

paramAtOrigin(f:UPUP,curve:UPUP,ex:Integer)== delay
-- this function must be
-- called for parametrization a the origin
u:=f
zer?(u) => 0
tt:=u exquo curve
"(tt case "failed") => 0
firstTerm:=valuationAndMore(u,curve)
od:=firstTerm.ord
coeff:=firstTerm.value
newU:=firstTerm.fnc - monomial(monomial(coeff,0)$UP,0)$UPUP
newCurve:=firstTerm.crv
series(od+ex,coeff,paramAtOrigin(newU,newCurve,ex+od))

localize(f:PolyRing,pt:ProjPt,curve:PolyRing,nV:Integer)==
curveT:=translateToOrigin(curve,pt,nV)
ft:=translateToOrigin(f,pt,nV)
fm:=minimalForm(curveT)
zer?(d:=totalDegree(fm)$PackPoly) => _
  error "the point is not on the curve"
"one?(d) => error "the point is singular"
subChart:=[i for i in 1..#symb | ~(i= (nV pretend PI))]
cl1:=degOneCoef(fm,(subChart.1) pretend PI)
cl2:=degOneCoef(fm,(subChart.2) pretend PI)
crt:List(Integer)
sc:List(Integer):=[(i pretend Integer) for i in subChart]
zer?(cl1) =>
crt:=concat(sc,nV)
[ft,curveT,crt]
zer?(cl2) =>
crt:=concat(reverse(sc),nV)
[ft,curveT,crt]
degl:=degree(curveT,listVar(subChart.1))
deg2 := degree(curveT, listVar(subChart.2))

deg1 > deg2 =>
    crt := concat(sc, nV)
    [ft, curveT, crt]
    crt := concat(reverse(sc), nV)
    [ft, curveT, crt]

localize2(f: PolyRing, pt: ProjPt, curve: PolyRing, nV: Integer) ==
    recBlowUp := localize(f, pt, curve, nV)
    f2 := coerceToUPUP(recBlowUp.fnc, recBlowUp.chart)
    curve2 := coerceToUPUP(recBlowUp.crv, recBlowUp.chart)
    [f2, curve2]
package MKBCFUNC MakeBinaryCompiledFunction

— MakeBinaryCompiledFunction.input —

)set break resume
)sys rm -f MakeBinaryCompiledFunction.output
)spool MakeBinaryCompiledFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MakeBinaryCompiledFunction

--R
--R MakeBinaryCompiledFunction(S: ConvertibleTo(InputForm),D1: Type,D2: Type,I: Type) is a package constructor
--R Abbreviation for MakeBinaryCompiledFunction is MKBCFUNC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MKBCFUNC

--R----------------------- Operations --------------------------
--R binaryFunction : Symbol -> ((D1,D2) -> I)
--R compiledFunction : (S,Symbol,Symbol) -> ((D1,D2) -> I)

--)spool
)lisp (bye)

———

1581
MakeBinaryCompiledFunction (MKBCFUNC)

Exports:
  binaryFunction  compiledFunction

— package MKBCFUNC MakeBinaryCompiledFunction —

)abbrev package MKBCFUNC MakeBinaryCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1988
++ Date Last Updated: 5 Mar 1990
++ Description:
++ Tools and transforms for making compiled functions from
++ top-level expressions

MakeBinaryCompiledFunction(S, D1, D2, I):Exports == Implementation where
  S: ConvertibleTo InputForm
  D1, D2, I: Type
SY ==> Symbol
DI ==> devaluate((D1, D2) -> I)$Lisp

Exports ==> with
  binaryFunction : SY -> ((D1, D2) -> I)
    ++ binaryFunction(s) is a local function
  compiledFunction: (S, SY, SY) -> ((D1, D2) -> I)
    ++ compiledFunction(expr,x,y) returns a function \spad{f: (D1, D2) -> I}
    ++ defined by \spad{f(x, y) == expr}.
    ++ Function f is compiled and directly
    ++ applicable to objects of type \spad{(D1, D2)}

Implementation ==> add
  import MakeFunction(S)
  func: (SY, D1, D2) -> I
  func(name, x, y) == FUNCALL(name, x, y, NIL$Lisp)$Lisp
  binaryFunction name == (d1:D1,d2:D2):I +-> func(name, d1, d2)
  compiledFunction(e, x, y) ==
  t := [devaluate(D1)$Lisp, devaluate(D2)$Lisp]$List(InputForm)
  binaryFunction compile(function(e, declare DI, x, y), t)

— MKBCFUNC.dotabb —

"MKBCFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MKBCFUNC"]
"KONVERT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KONVERT"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"MKBCFUNC" -> "KONVERT"
"MKBCFUNC" -> "TYPE"

package MKFLCFN MakeFloatCompiledFunction

— MakeFloatCompiledFunction.input —

)set break resume
)sys rm -f MakeFloatCompiledFunction.output
)spool MakeFloatCompiledFunction.output
```plaintext
--S 1 of 1
)show MakeFloatCompiledFunction
--R
--R MakeFloatCompiledFunction(S: ConvertibleTo(InputForm)) is a package constructor
--R Abbreviation for MakeFloatCompiledFunction is MKFLCFN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MKFLCFN
--R
--R------------------------------- Operations --------------------------------
--R makeFloatFunction : (S,Symbol) -> (DoubleFloat -> DoubleFloat)
--R makeFloatFunction : (S,Symbol,Symbol) -> ((DoubleFloat,DoubleFloat) -> DoubleFloat)
--R
--E 1

)spool
)lisp (bye)

---

—— MakeFloatCompiledFunction.help ——

====================================================================
MakeFloatCompiledFunction examples
====================================================================

Tools for making compiled functions from top-level expressions
MakeFloatCompiledFunction transforms top-level objects into compiled
Lisp functions whose arguments are Lisp floats. This by-passes the
Axiom compiler and interpreter, thereby gaining several orders of
magnitude.

See Also:
  o )show MakeFloatCompiledFunction

——
```
MakeFloatCompiledFunction (MKFLCFN)

Exports:

\[
\text{package MKFLCFN MakeFloatCompiledFunction}
\]

\[
\text{Exports == Implementation where}
\]

\[
\text{S: ConvertibleTo InputForm}
\]

\[
\text{INF ==> InputForm}
\]

\[
\text{SF ==> DoubleFloat}
\]

\[
\text{DI1 ==> devaluate(SF -> SF)$Lisp}
\]

\[
\text{DI2 ==> devaluate((SF, SF) -> SF)$Lisp}
\]

\[
\text{Exports ==> with}
\]

\[
\text{makeFloatFunction: (S, Symbol) -> (SF -> SF)}
\]

\[
\text{++ makeFloatFunction(expr, x) returns a Lisp function}
\]

\[
\text{++ \spad{(f: \axiomType{DoubleFloat} -> \axiomType{DoubleFloat})} defined by \spad{(f(x) \equiv expr).}}
\]

\[
\text{++ Function f is compiled and directly}
\]

\[
\text{++ applicable to objects of type \axiomType{DoubleFloat}.}
\]

\[
\text{makeFloatFunction: (S, Symbol, Symbol) -> ((SF, SF) -> SF)}
\]

\[
\text{++ makeFloatFunction(expr, x, y) returns a Lisp function}
\]

\[
\text{++ \spad{(f: \axiomType{DoubleFloat}) \rightarrow \axiomType{DoubleFloat}}}
\]
++ defined by \spad{f(x, y) == expr}.
++ Function f is compiled and directly
++ applicable to objects of type \spad{(\axiomType{DoubleFloat},
++ \axiomType{DoubleFloat})}.

Implementation ==> add
import MakeUnaryCompiledFunction(S, SF, SF)
import MakeBinaryCompiledFunction(S, SF, SF, SF)

streq? : (INF, String) -> Boolean
streqlist?: (INF, List String) -> Boolean
gencode : (String, List INF) -> INF
mkLisp : INF -> Union(INF, "failed")
mkLispList: List INF -> Union(List INF, "failed")
mkDefun : (INF, List INF) -> INF
mkPretend : INF -> INF
mkCTOR : INF -> INF

lsf := convert([convert("DoubleFloat":Symbol)@INF]$List(INF))@INF

streq?(s, st) == s = convert(st::Symbol)@INF
gencode(s, l) == convert(concat(convert(s::Symbol)@INF, l))@INF
streqlist?(s, l) == member?(string symbol s, l)

mkPretend form ==
   convert([convert("pretend":Symbol), form, lsf]$List(INF))@INF

mkCTOR form ==
   convert([convert("C-TO-R":Symbol), form]$List(INF))@INF

mkLispCall name ==
   convert([convert("\$elt":Symbol),
            convert("Lisp":Symbol), name]$List(INF))@INF

mkDefun(s, lv) ==
   name := convert(new()$Symbol)@INF
   fun := convert([convert("DEFUN":Symbol), name, convert lv,
                   gencode("DECLARE",[gencode("FLOAT",lv)]),mkCTOR s]$List(INF))@INF
   EVAL(fun)$Lisp
   if _$compileDontDefineFunctions$Lisp then COMPILE(name)$Lisp
   name

makeFloatFunction(f, x, y) ==
   (u := mkLisp(convert(f)@INF)) case "failed" =>
      compiledFunction(f, x, y)
   name := mkDefun(u::INF, [ix := convert x, iy := convert y])
   t := [lsf, lsf]$List(INF)
   spadname := declare DI2
spadform := mkPretend convert([mkLispCall name, ix, iy]$List(INF))@INF
interpret function(spadform, [x, y], spadname)
binaryFunction compile(spadname, t)

makeFloatFunction(f, var) ==
(u := mkLisp(convert(f)@INF)) case "failed" =>
    compiledFunction(f, var)
name := mkDefun(u::INF, [ivar := convert var])
t := [lsf]$List(INF)
spadname := declare DI1
spadform := mkPretend convert([mkLispCall name, ivar]$List(INF))@INF
interpret function(spadform, [var], spadname)
unaryFunction compile(spadname, t)

mkLispList l ==
    ans := nil()$List(INF)
    for s in l repeat
        (u := mkLisp s) case "failed" => return "failed"
        ans := concat(u::INF, ans)
    reverse_! ans

mkLisp s ==
    atom? s => s
    op := first(l := destruct s)
    (u := mkLispList rest l) case "failed" => "failed"
    ll := u::List(INF)
    streqlist?(op, [+",", "/","-"] ) => convert(concat(op, ll))@INF
    streq?(op, "**") => gencode("EXPT", ll)
    streqlist?(op, ["exp","sin","cos","tan","atan",
                   "log", "sinh","cosh","tanh","asinh","acosh","atanh","sqrt"] ) =>
        gencode(upperCase string symbol op, ll)
    streq?(op, "nthRoot") =>
        second ll = convert(2::Integer)@INF =>
            gencode("SQRT", [first ll])
            gencode("EXPT", concat(first ll, [1$INF / second ll]))
    streq?(op, "float") =>
        a := ll.1
e := ll.2
b := ll.3
_*(a, EXPT(b, e)$Lisp)$Lisp pretend INF
"failed"

— MKFLCFN.dotabb —
package MKFUNC MakeFunction

— MakeFunction.input —

)set break resume
)spool MakeFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 10
expr := (x - exp x + 1)^2 * (sin(x^2) * x + 1)^3
--R
--R (1)
--R 3 x 2 4 3 x 5 4 3 2 3
--R (x (%e ) + (- 2x - 2x )%e + x + 2x + x )sin(x )
--R +
--R 2 x 2 3 2 x 4 3 2 2 2
--R (3x (%e ) + (- 6x - 6x )%e + 3x + 6x + 3x )sin(x )
--R +
--R 2 x 2 3 2 x 4 3 2 2 2
--R (3x (%e ) + (- 6x - 6x )%e + 3x + 6x + 3x )sin(x ) + (%e )
--R +
--R x 2
--R (- 2x - 2)%e + x + 2x + 1
--R Type: Expression(Integer)
--E 1

--S 2 of 10
function(expr, f, x)
--R
--R (2) f
--R Type: Symbol
--E 2

--S 3 of 10
tbl := [f(0.1 * i - 1) for i in 0..20]
--R
--R Compiling function f with type Float -> Float
--R
--R (3)
--R [0.0005391844 0362701574, 0.0039657551 1844206653, 0.00040584985 7597822062 55, 0.0015349542 8910500836 48, 0.00003424903 1549879905 716, 0.0000233304 8276098819 6001, 0.0000268186 8782862599 4229, 0.00004691571 3720051642 621, 0.0026924576 5968519586 08, 0.0101486881 736935148 8, 0.0313833725 8543810564 3, 0.0876991144 5154615297 9, 0.2313019789 343968362, 0.5843743955 958098772, 1.4114930171 992819197, 3.2216948276 75164252]  
--R Type: List(Float)
--E 3

--S 4 of 10
--R e := (x - y + 1)^2 * (x^2 * y + 1)^2
--R
--R (4)
--R 4 4 5 4 2 3 6 5 4 3 2 2
--R x y + (- 2x - 2x + 2x )y + (x + 2x + x - 4x - 4x + 1)y
--R +
--R 4 3 2 2
--R (2x + 4x + 2x - 2x - 2)y + x + 2x + 1
--R Type: Polynomial(Integer)
--E 4

--S 5 of 10
--R function(e, g, [x, y])
--R
--R (5) g
--R Type: Symbol
--E 5

--S 6 of 10
--R function(e, h, x, y)
--R
--R (6) h
--R Type: Symbol
--E 6

--S 7 of 10
--R m1 := squareMatrix [ [1, 2], [3, 4] ]
--R
--R +1 2+
--R (7) | | +3 4+
m2 := squareMatrix [[1, 0], [-1, 1]]

h(m1, m2)

)...
MakeFunction examples
====================================================================

Tools for making interpreter functions from top-level expressions
Transforms top-level objects into interpreter functions.

It is sometimes useful to be able to define a function given by
the result of a calculation.

Suppose that you have obtained the following expression after several
computations and that you now want to tabulate the numerical values of
f for x between -1 and +1 with increment 0.1.

\[
\text{expr} := (x - \exp x + 1)^2 \ast (\sin(x^2) \ast x + 1)^3
\]

\[
\begin{align*}
3 & \times 2 \quad 4 & \times 3 & \times 5 & 4 & 3 & 2 & 3 \\
(x \times \%e) & + (-2x - 2x) \times \%e & + x + 2x & + x \sin(x) \\
+ & 2 \times 2 \quad 3 & \times 4 & 3 & 2 & 2 \\
(3x \times \%e) & + (-6x - 6x) \times \%e & + 3x & + 6x & + 3x \sin(x) \\
+ & x \times 2 \quad 2 & \times 3 & 2 & 2 & 2 \\
(3x \times \%e) & + (-6x - 6x) \times \%e & + 3x & + 6x & + 3x \sin(x) & + \%e \\
+ & x \times 2 \\
(-2x - 2) \times \%e & + x & + 2x & + 1
\end{align*}
\]

Type: Expression Integer

You could, of course, use the function eval within a loop and evaluate
expr twenty-one times, but this would be quite slow. A better way is
to create a numerical function f such that f(x) is defined by the
expression expr above, but without retyping expr! The package
MakeFunction provides the operation function which does exactly this.

Issue this to create the function f(x) given by expr.

\[
\text{function(expr, f, x)}
\]

f

Type: Symbol

To tabulate expr, we can now quickly evaluate f 21 times.

\[
\text{tbl} := [f(0.1 \ast i - 1) \text{ for i in 0..20]};
\]

Type: List Float

Use the list [x1,...,xn] as the third argument to function to create a
multivariate function f(x1,...,xn).

\[
\text{e} := (x - y + 1)^2 \ast (x^2 \ast y + 1)^2
\]

\[
\begin{align*}
4 & 4 \quad 5 & 4 & 2 & 3 & 6 & 5 & 4 & 3 & 2 & 2 \\
x & y & + (-2x - 2x + 2x) & y & + (x + 2x + x - 4x - 4x + 1) & y
\end{align*}
\]
\[
(2x^2 + 4x + 2x - 2)y + x^2 + 2x + 1
\]

Type: Polynomial Integer

\[\text{function(e, g, [x, y])}\]
\[g\]

Type: Symbol

In the case of just two variables, they can be given as arguments without making them into a list.

\[\text{function(e, h, x, y)}\]
\[h\]

Type: Symbol

Note that the functions created by function are not limited to floating point numbers, but can be applied to any type for which they are defined.

\[m1 := \text{squareMatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}\]
\[\begin{array}{cc}
+1 & 2+ \\
| & | \\
+3 & 4+
\end{array}
\]

Type: SquareMatrix(2,Integer)

\[m2 := \text{squareMatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}\]
\[\begin{array}{cc}
+1 & 0+ \\
| & | \\
+-1 & 1+
\end{array}
\]

Type: SquareMatrix(2,Integer)

\[h(m1, m2)\]
\[\begin{array}{ccc}
+-7836 & 8960 & + \\
| & | \\
+-17132 & 19588+
\end{array}
\]

Type: SquareMatrix(2,Integer)

See Also:
- )show MakeFunction
MakeFunction (MKFUNC)

Exports:
function

--- package MKFUNC MakeFunction ---

)abbrev package MKFUNC MakeFunction
++ Author: Manuel Bronstein
++ Date Created: 22 Nov 1988
++ Date Last Updated: 8 Jan 1990
++ Description:
++ Tools for making interpreter functions from top-level expressions
++ Transforms top-level objects into interpreter functions.

MakeFunction(S:ConvertibleTo InputForm): Exports == Implementation where
SY ==> Symbol
Exports ==> with
  function: (S, SY ) -> SY
    ++ function(e, foo) creates a function \spad{foo() == e}.
  function: (S, SY, SY) -> SY
    ++ function(e, foo, x) creates a function \spad{foo(x) == e}.
  function: (S, SY, SY, SY) -> SY
    ++ function(e, foo, x, y) creates a function \spad{foo(x, y) = e}.
  function: (S, SY, List SY) -> SY
    ++ \spad{function(e, foo, [x1,...,xn])} creates a function
    ++ \spad{foo(x1,...,xn) == e}.

Implementation ==> add
  function(s, name) == function(s, name, nil())
  function(s:S, name:SY, x:SY) == function(s, name, [x])
  function(s, name, x, y) == function(s, name, [x, y])

  function(s:S, name:SY, args:List SY) ==
    interpret function(convert s, args, name)$InputForm
    name
package MKRECORD MakeRecord

--- MakeRecord.input ---

)set break resume
)sys rm -f MakeRecord.output
)spool MakeRecord.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show MakeRecord
--R
--R MakeRecord(S: Type,T$: Type) is a package constructor
--R Abbreviation for MakeRecord is MKRECORD
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MKRECORD
--R
--R-------------------------------- Operations --------------------------------
--R makeRecord : (S,T$) -> Record(part1: S,part2: T$)
--R
--E 1

)spool
)lisp (bye)

--- MakeRecord.help ---

====================================================================
MakeRecord examples
====================================================================

MakeRecord is used internally by the interpreter to create record
types which are used for doing parallel iterations on streams.

See Also:
o )show MakeRecord

---

MakeRecord (MKRECORD)

Exports:
makeRecord

--- package MKRECORD MakeRecord ---

)abbrev package MKRECORD MakeRecord
++ Description:
++ MakeRecord is used internally by the interpreter to create record
++ types which are used for doing parallel iterations on streams.

MakeRecord(S: Type, T: Type): public == private where
  public == with
    makeRecord: (S,T) -> Record(part1: S, part2: T)
    ++ makeRecord(a,b) creates a record object with type
    ++ Record(part1:S, part2:T),
    ++ where part1 is \spad{a} and part2 is \spad{b}.
  private == add
    makeRecord(s: S, t: T) ==
      [s,t]$Record(part1: S, part2: T)
package MKUCFUNC MakeUnaryCompiledFunction

-- MakeUnaryCompiledFunction.input --

)set break resume
)sys rm -f MakeUnaryCompiledFunction.output
)spool MakeUnaryCompiledFunction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MakeUnaryCompiledFunction
--R
--R MakeUnaryCompiledFunction(S: ConvertibleTo(InputForm),D: Type,I: Type) is a package constructor
--R Abbreviation for MakeUnaryCompiledFunction is MKUCFUNC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MKUCFUNC
--R
--R------------------------------------------------------------------- Operations -------------------------
--R unaryFunction : Symbol -> (D -> I)
--R compiledFunction : (S,Symbol) -> (D -> I)
--R
--E 1

)spool
)lisp (bye)

— MakeUnaryCompiledFunction.help —

====================================================================
MakeUnaryCompiledFunction examples
====================================================================

Tools for making compiled functions from top-level expressions
Transforms top-level objects into compiled functions.

See Also:
o )show MakeUnaryCompiledFunction

---

MakeUnaryCompiledFunction (MKUCFUNC)

Exports:
unaryFunction compiledFunction

--- package MKUCFUNC MakeUnaryCompiledFunction ---

)abbrev package MKUCFUNC MakeUnaryCompiledFunction
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1988
++ Date Last Updated: 5 Mar 1990
++ Description:
++ Tools for making compiled functions from top-level expressions
++ Transforms top-level objects into compiled functions.

MakeUnaryCompiledFunction(S, D, I): Exports == Implementation where
  S:ConvertibleTo InputForm
  D, I: Type

    SY  ==>  Symbol
    DI  ==>  devaluate(D -> I)$Lisp
Exports ==> with
unaryFunction : SY -> (D -> I)
  ++ unaryFunction(a) is a local function
compiledFunction: (S, SY) -> (D -> I)
  ++ compiledFunction(expr, x) returns a function \( f: D \to I \)
  ++ defined by \( f(x) = expr \).
  ++ Function f is compiled and directly
  ++ applicable to objects of type D.

Implementation ==> add
import MakeFunction(S)
func: (SY, D) -> I
func(name, x) == FUNCALL(name, x, NIL$Lisp)$Lisp
unaryFunction name == (d1:D):I +-> func(name, d1)

compiledFunction(e:S, x:SY) ==
t := [convert([devaluate(D)$Lisp]$List(InputForm))
  ]$List(InputForm)
unaryFunction compile(function(e, declare DI, x), t)

package MAPHACK1 MappingPackageInternalHacks1
--S 1 of 1
)show MappingPackageInternalHacks1
--R
--R MappingPackageInternalHacks1(A: SetCategory) is a package constructor
--R Abbreviation for MappingPackageInternalHacks1 is MAPHACK1
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MAPHACK1
--R
--R---------------------------------- Operations ----------------------------------
--R iter : ((A -> A),NonNegativeInteger,A) -> A
--R recur : (((NonNegativeInteger,A) -> A),NonNegativeInteger,A) -> A
--R
--E 1

)spool
)lisp (bye)

——

— MappingPackageInternalHacks1.help —

====================================================================
MappingPackageInternalHacks1 examples
====================================================================

Various Currying operations.

See Also:
o )show MappingPackageInternalHacks1

——
MappingPackageInternalHacks1 (MAPHACK1)

Exports:
iter  recur

— package MAPHACK1 MappingPackageInternalHacks1 —

)abbrev package MAPHACK1 MappingPackageInternalHacks1
++ Author: S.M.Watt and W.H.Burge
++ Date Created: Jan 87
++ Date Last Updated: Feb 92
++ Description: 
++ Various Currying operations.

MappingPackageInternalHacks1(A: SetCategory): MPcat == MPdef where

NNI ==> NonNegativeInteger

MPcat == with
iter: ((A -> A), NNI, A) -> A
++ \spad{iter(f,n,x)} applies \spad{f n} times to \spad{x}.
recur: ((NNI, A)->A, NNI, A) -> A
++ \spad{recur(n,g,x)} is \spad{g(n,g(n-1,..g(1,x)..))}.

MPdef == add
iter(g,n,x) ==
for i in 1..n repeat x := g x  -- g(g(...x))
x
recur(g,n,x) ==
for i in 1..n repeat x := g(i,x)  -- g(n,g(n-1,..g(1,x)...))
x

— MAPHACK1.dotabb —
package MAPHACK2 MappingPackageInternalHacks2

---

MappingPackageInternalHacks2.input ---

)set break resume
)sys rm -f MappingPackageInternalHacks2.output
)spool MappingPackageInternalHacks2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)]show MappingPackageInternalHacks2

--R
--R MappingPackageInternalHacks2(A: SetCategory,C: SetCategory) is a package constructor
--R Abbreviation for MappingPackageInternalHacks2 is MAPHACK2
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MAPHACK2
--R
--R----------------------------- Operations -----------------------------
--R arg1 : (A,C) -> A    arg2 : (A,C) -> C
--R
--E 1

)spool
)lisp (bye)

---

MappingPackageInternalHacks2.help ---

====================================================================
MappingPackageInternalHacks2 examples
====================================================================

Various Currying operations.
See Also:
  o )show MappingPackageInternalHacks2

MappingPackageInternalHacks2 (MAPHACK2)

Exports:
  arg1  arg2

--- package MAPHACK2 MappingPackageInternalHacks2 ---

)abbrev package MAPHACK2 MappingPackageInternalHacks2
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks2(A: SetCategory, C: SetCategory):_
MPcat == MPdef where
  NNI ==> NonNegativeInteger

MPcat == with
  arg1: (A, C) -> A
    +=\spad{arg1(a,c)} selects its first argument.
  arg2: (A, C) -> C
    +=\spad{arg2(a,c)} selects its second argument.

MPdef == add
  arg1(a, c) == a
  arg2(a, c) == c
package MAPHACK3 MappingPackageInternalHacks3

— MappingPackageInternalHacks3.input —

)set break resume
)sys rm -f MappingPackageInternalHacks3.output
)spool MappingPackageInternalHacks3.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MappingPackageInternalHacks3

--R MappingPackageInternalHacks3(A: SetCategory,B: SetCategory,C: SetCategory) is a package constructor
--R Abbreviation for MappingPackageInternalHacks3 is MAPHACK3
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MAPHACK3
--R
--R------------------- Operations -------------------------------
--R comp : ((B -> C),(A -> B),A) -> C
--R
--E 1

)spool
)lisp (bye)

— MappingPackageInternalHacks3.help —

================================================================================
MappingPackageInternalHacks3 examples
================================================================================
Various Currying operations.

See Also:
  o )show MappingPackageInternalHacks3

---

MappingPackageInternalHacks3 (MAPHACK3)

Exports:
  comp

--- package MAPHACK3 MappingPackageInternalHacks3 ---

)abbrev package MAPHACK3 MappingPackageInternalHacks3
++ Description:
++ Various Currying operations.

MappingPackageInternalHacks3(A: SetCategory, B: SetCategory, C: SetCategory):_
  MPCat == MPdef where
    NNI ==> NonNegativeInteger

    MPCat == with
      comp: (B->C, A->B, A) -> C
      ++\spad{comp(f,g,x)} is \spad{f(g(x)).

    MPdef == add
      comp(g,h,x) == g h x

---
package MAPPKG1 MappingPackage1

— MappingPackage1.input —

)set break resume
)spool MappingPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 27
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R    Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R    (Integer) has been added to workspace.
--R
--E 1

--S 2 of 27
power(2,3)
--R
--R    Compiling function power with type (Fraction(Integer),Integer) ->
--R        Fraction(Integer)
--R
--R    (2) 8
--R
--E 2

--S 3 of 27
rewop := twist power
--R
--R
--R    (3) theMap(MAPPKG3;twist;MM;5!0)
--R
--E 3
\(\text{rewop}(3, 2)\)
\(\text{square} : \text{FRAC INT} \rightarrow \text{FRAC INT}\)
\(\text{squirrel} := \text{constantRight}(\text{square})\)
\(\text{squirrel}(1/2, 1/3)\)
sixteen := curry(square, 4/1)

-- R
-- R
-- R (10) theMap(MAPPKG2;curry;MAM;2!0,0)
-- R Type: (() -> Fraction(Integer))
-- E 10

-- S 11 of 27
sixteen()
-- R
-- R (11) 16
-- R Type: Fraction(Integer)
-- E 11

-- S 12 of 27
square2:=square*square
-- R
-- R (12) theMap(MAPPKG3;*;MMM;6!0,0)
-- R Type: (Fraction(Integer) -> Fraction(Integer))
-- E 12

-- S 13 of 27
square2 3
-- R
-- R (13) 81
-- R Type: Fraction(Integer)
-- E 13

-- S 14 of 27
sc(x: FRAC INT): FRAC INT == x + 1
-- R
-- R Function declaration sc : Fraction(Integer) -> Fraction(Integer) has
-- R been added to workspace.
-- R Type: Void
-- E 14

-- S 15 of 27
incfns := [sc**i for i in 0..10]
-- R
-- R Compiling function sc with type Fraction(Integer) -> Fraction(
-- R Integer)
-- R
-- R (15)
-- R [theMap(MAPPKG1;**;MNN1M;6!0,0), theMap(MAPPKG1;**;MNN1M;6!0,0),
-- R theMap(MAPPKG1;**;MNN1M;6!0,0), theMap(MAPPKG1;**;MNN1M;6!0,0),
-- R theMap(MAPPKG1;**;MNN1M;6!0,0), theMap(MAPPKG1;**;MNN1M;6!0,0),
-- R theMap(MAPPKG1;**;MNN1M;6!0,0), theMap(MAPPKG1;**;MNN1M;6!0,0),
-- R theMap(MAPPKG1;**;MNN1M;6!0,0), theMap(MAPPKG1;**;MNN1M;6!0,0),
-- R theMap(MAPPKG1;**;MNN1M;6!0,0), theMap(MAPPKG1;**;MNN1M;6!0,0),]
(f 4 for f in incfns)

(16) [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

(times(n:NNI, i:INT):INT == n*i)

(17) times : (NonNegativeInteger, Integer) -> Integer

r := recur(times)

(18) theMap(MAPPKG1; recur; 2M; 7!0, 0)

fact := curryRight(r, 1)

(19) theMap(MAPPKG3; curryRight; MBM; 1!0, 0)

mto2ton(m, n) ==
  raiser := square^n
mto2ton(3, 3)

shiftfib(r: List INT) : INT ==
t := r.1
r.1 := r.2
r.2 := r.2 + t
t

fibinit: List INT := [0, 1]

fibs := curry(shiftfib, fibinit)
Various Currying operations.

Function are objects of type Mapping. In this section we demonstrate some library operations from the packages MappingPackage1, MappingPackage2, and MappingPackage3 that manipulate and create functions. Some terminology: a nullary function takes no arguments, a unary function takes one argument, and a binary function takes two arguments.

We begin by creating an example function that raises a rational number to an integer exponent.

```
power(q: FRAC INT, n: INT): FRAC INT == q**n

power(2,3)
```
The twist operation transposes the arguments of a binary function. Here rewop(a, b) is power(b, a).

 rewop := twist power  
   theMap(MAPPKG3;twist;MM;5!0)  
   Type: ((Integer,Fraction Integer) -> Fraction Integer)

This is 2^-3.

 rewop(3, 2)  
  8  
  Type: Fraction Integer

Now we define square in terms of power.

 square: FRAC INT -> FRAC INT  
   Type: Void

The curryRight operation creates a unary function from a binary one by providing a constant argument on the right.

 square:= curryRight(power, 2)  
   theMap(MAPPKG3;curryRight;MBM;1!0,0)  
   Type: (Fraction Integer -> Fraction Integer)

Likewise, the curryLeft operation provides a constant argument on the left.

 square 4  
  16  
  Type: Fraction Integer

The constantRight operation creates (in a trivial way) a binary function from a unary one: constantRight(f) is the function g such that g(a,b)= f(a).

 squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)  
   theMap(MAPPKG3;constantRight;MM;3!0)  
   Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)

Likewise, constantLeft(f) is the function g such that g(a,b)= f(b).

 squirrel(1/2, 1/3)  
  1  
  -  
  4  
  Type: Fraction Integer
The curry operation makes a unary function nullary.

\[ \text{sixteen} := \text{curry}(\text{square}, 4/1) \]
\[ \text{theMap(MAPPKG2;curry;MAM;2!0,0)} \]
\[ \text{Type: } ((() \rightarrow \text{Fraction Integer}) \]
\[ \text{sixteen()} \]
\[ 16 \]
\[ \text{Type: Fraction Integer} \]

The * operation constructs composed functions.

\[ \text{square2} := \text{square} \ast \text{square} \]
\[ \text{theMap(MAPPKG3;*;MMM;6!0,0)} \]
\[ \text{Type: } (\text{Fraction Integer} \rightarrow \text{Fraction Integer}) \]
\[ \text{square2 3} \]
\[ 81 \]
\[ \text{Type: Fraction Integer} \]

Use the ** operation to create functions that are n-fold iterations of other functions.

\[ \text{sc}(x: \text{FRAC INT}): \text{FRAC INT} = x + 1 \]
\[ \text{Type: Void} \]

This is a list of Mapping objects.

\[ \text{incfns} := [\text{sc}**i \text{ for } i \text{ in } 0..10] \]
\[ [\text{theMap(MAPPKG1;**;MINI;6!0,0)}, \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \]
\[ \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \]
\[ \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \]
\[ \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \]
\[ \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \text{theMap(MAPPKG1;**;MINI;6!0,0)}, \]
\[ \text{theMap(MAPPKG1;**;MINI;6!0,0)}] \]
\[ \text{Type: List (Fraction Integer} \rightarrow \text{Fraction Integer}) \]

This is a list of applications of those functions.

\[ [f \text{ 4 for } f \text{ in incfns}] \]
\[ [4,5,6,7,8,9,10,11,12,13,14] \]
\[ \text{Type: List Fraction Integer} \]

Use the recur operation for recursion:

\[ \text{g} := \text{recur } f \text{ means } g(n,x) = f(n,f(n-1,\ldots,f(1,x))). \]
\[ \text{times}(n:NNI, i:INT):INT = n*i \]
\[ \text{Type: Void} \]
r := recur(times)
theMap(MAPPKG1;recur;2M;7!0,0)
    Type: ((NonNegativeInteger,Integer) -> Integer)
This is a factorial function.

fact := curryRight(r, 1)
theMap(MAPPKG3;curryRight;MEM;1!0,0)
    Type: (NonNegativeInteger -> Integer)

fact 4
24
    Type: PositiveInteger

Constructed functions can be used within other functions.

mto2ton(m, n) ==
    raiser := square^n
    raiser m
    Type: Void
This is $3^{2^3}$.

mto2ton(3, 3)
6561
    Type: Fraction Integer

Here shiftfib is a unary function that modifies its argument.

shiftfib(r: List INT) : INT ==
    t := r.1
    r.1 := r.2
    r.2 := r.2 + t
    t
    Type: Void
By currying over the argument we get a function with private state.

fibinit: List INT := [0, 1]
[0,1]
    Type: List Integer

fibs := curry(shiftfib, fibinit)
theMap(MAPPKG2;curry;MAM;2!0,0)
    Type: (() -> Integer)

[fibs() for i in 0..30]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040]
MappingPackage1 (MAPPKG1)

Exports:
coerce      fixedPoint id      nullary recur

— package MAPPKG1 MappingPackage1 —

)abbrev package MAPPKG1 MappingPackage1
++ Author: S.M.Watt and W.H.Burge
++ Date Created: Jan 87
++ Date Last Updated: Feb 92
++ Description:
++ Various Currying operations.

MappingPackage1(A:SetCategory): MPcat == MPdef where

N NI  ==>  NonNegativeInteger

MPcat == with
  nullary: A  ->  (()->A)
  coerc e: A  ->  (()->A)

  ++ \spad{nullary A} changes its argument into a
  ++ nullary function.
++\spad{coerce A} changes its argument into a ++ nullary function.

**fixedPoint**: (A->A) -> A
++\spad{fixedPoint f} is the fixed point of function \spad{f}. ++ that is, such that \spad{fixedPoint f = f(fixedPoint f)}.

**fixedPoint**: (List A->List A, Integer) -> List A
++\spad{fixedPoint(f,n)} is the fixed point of function ++ \spad{f} which is assumed to transform a list of length ++ \spad{n}.

**id**: A -> A
++\spad{id x} is \spad{x}.

"**": (A->A, NNI) -> (A->A)
++\spad{f**n} is the function which is the n-fold application ++ of \spad{f}.

**recur**: ((NNI, A)->A) -> ((NNI, A)->A)
++\spad{recur(g)} is the function \spad{h} such that ++ \spad{h(n,x)= g(n,g(n-1,..g(1,x)..))}.

MPdef == add

MappingPackageInternalHacks1(A)

a: A
faa: A -> A
f0a: () -> A

nullary a == a
coerce a == nullary a

fixedPoint faa ==
  g0 := GENSYM()$Lisp
  g1 := faa g0
  EQ(g0, g1)$Lisp => error "All points are fixed points"
  GEQNSUBSTLIST([g0]$Lisp, [g1]$Lisp, g1)$Lisp

fixedPoint(fll, n) ==
  g0 := [(GENSYM()$Lisp):A for i in 1..n]
  g1 := fll g0
  or/[EQ(e0,e1)$Lisp for e0 in g0 for e1 in g1] =>
    error "All points are fixed points"
  GEQNSUBSTLIST([g0]$Lisp, [g1]$Lisp, g1)$Lisp

-- Composition and recursion.

id a == a

**g**n == (a1:A):A +-> iter(g, n, a1)
recur fnaa == (n1:NNI,a2:A):A +-> recur(fnaa, n1, a2)

— MAPPKG1.dotabb —

"MAPPKG1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPPKG1" -> "BASTYPE"
"MAPPKG1" -> "KOERCE"

package MAPPKG2 MappingPackage2

— MappingPackage2.input —

)set break resume
)spool MappingPackage2.output
)set message test on
)set message auto off
)clear all

--S 1 of 27
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R (Integer) has been added to workspace.
--R
--E 1

--S 2 of 27
power(2,3)
--R
--R Compiling function power with type (Fraction(Integer),Integer) ->
--R         Fraction(Integer)
--R
--R (2) 8
--R
Type: Fraction(Integer)
--E 2

--S 3 of 27
rewop := twist power
--R
--R
--R (3) theMap(MAPPGK3;twist;MM;S10)
--R Type: ((Integer,Fraction(Integer)) -> Fraction(Integer))
--E 3

--S 4 of 27
rewop(3, 2)
--R
--R
--R (4) 8
--R Type: Fraction(Integer)
--E 4

--S 5 of 27
square: FRAC INT -> FRAC INT
--R
--R Type: Void
--E 5

--S 6 of 27
square:= curryRight(power, 2)
--R
--R
--R (6) theMap(MAPPGK3;curryRight;MBM;1!0,0)
--R Type: (Fraction(Integer) -> Fraction(Integer))
--E 6

--S 7 of 27
square 4
--R
--R
--R (7) 16
--R Type: Fraction(Integer)
--E 7

--S 8 of 27
squirrel:= constantRight(square)$MAPPGK3(FRAC INT,FRAC INT,FRAC INT)
--R
--R
--R (8) theMap(MAPPGK3;constantRight;MM;3!0)
--R Type: ((Fraction(Integer),Fraction(Integer)) -> Fraction(Integer))
--E 8

--S 9 of 27
squirrel(1/2, 1/3)
--R
--R
--R 1
sixteen := curry(square, 4/1)

sixteen()

square2 := square*square

square2 3

sc(x: FRAC INT): FRAC INT == x + 1

incfns := [sc**i for i in 0..10]
--R
--R (15)
--R [theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R theMap(MAPPKG1;**;MNniM;6!0,0), theMap(MAPPKG1;**;MNniM;6!0,0),
--R theMap(MAPPKG1;**;MNniM;6!0,0),
                    Type: List((Fraction(Integer) -> Fraction(Integer)))
--E 15

--S 16 of 27
[f 4 for f in incfns]
--R
--R (16) [4,5,6,7,8,9,10,11,12,13,14]
--R                     Type: List(Fraction(Integer))
--E 16

--S 17 of 27
times(n:NNI, i:INT):INT == n*i
--R
--R Function declaration times : (NonNegativeInteger,Integer) -> Integer
--R has been added to workspace.
--R                     Type: Void
--E 17

--S 18 of 27
r := recur(times)
--R
--R Compiling function times with type (NonNegativeInteger,Integer) ->
--R   Integer
--R (18) theMap(MAPPKG1;recur;2M;7!0,0)
--R                     Type: ((NonNegativeInteger,Integer) -> Integer)
--E 18

--S 19 of 27
fact := curryRight(r, 1)
--R
--R (19) theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R                     Type: (NonNegativeInteger -> Integer)
--E 19

--S 20 of 27
fact 4
--R
--R (20) 24
1620

CHAPTER 14. CHAPTER M

--R
--E 20

--S 21 of 27
mto2ton(m, n) ==
  raiser := square^n
  raiser m
--R
--R
--E 21

--S 22 of 27
mto2ton(3, 3)
--R
--R Compiling function mto2ton with type (PositiveInteger,
--R PositiveInteger) -> Fraction(Integer)
--R
--R (22) 6561
--R
--E 22

--S 23 of 27
shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
--R
--R Function declaration shiftfib : List(Integer) -> Integer has been
--R added to workspace.
--R
--E 23

--S 24 of 27
fibinit: List INT := [0, 1]
--R
--R
--R (24) [0,1]
--E 24

--S 25 of 27
fibs := curry(shiftfib, fibinit)
--R
--R Compiling function shiftfib with type List(Integer) -> Integer
--R
--R (25) theMap(MAPPKG2;curry;MAM;2!0,0)
--E 25
---S 26 of 27
[fibs() for i in 0..30]
--R
--R
--R (26)
--R [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
--R 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418,
--R 317811, 514229, 832040]
--R Type: List(Integer)
--E 26

---S 27 of 27
)show MappingPackage2
--R
--R MappingPackage2(A: SetCategory,C: SetCategory) is a package constructor
--R Abbreviation for MappingPackage2 is MAPPKG2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MAPPKG2
--R
--R---------------------------------------------------------- Operations -------------------------------------------
--R const : C -> (A -> C)
--R constant : (() -> C) -> (A -> C)
--R curry : ((A -> C),A) -> (() -> C)
--R diag : ((A,A) -> C) -> (A -> C)
--R
--E 27

)spool
)lisp (bye)

-- MappingPackage2.help --

====================================================================
MappingPackage2 examples
====================================================================

Various Currying operations.

Function are objects of type Mapping. In this section we demonstrate
some library operations from the packages MappingPackage1, MappingPackage2,
and MappingPackage3 that manipulate and create functions. Some terminology:
a nullary function takes no arguments, a unary function takes one argument,
and a binary function takes two arguments.

We begin by creating an example function that raises a rational number
to an integer exponent.

power(q: FRAC INT, n: INT): FRAC INT == q**n
   Type: Void
The twist operation transposes the arguments of a binary function. Here $\text{rewop}(a, b)$ is $\text{power}(b, a)$.

\begin{verbatim}
rewop := twist power
theMap(MAPPKG3;twist;MM;5!0)
  Type: ((Integer,Fraction Integer) -> Fraction Integer)
\end{verbatim}

This is $2^{-3}$.

\begin{verbatim}
rewop(3, 2)
  8
  Type: Fraction Integer
\end{verbatim}

Now we define square in terms of power.

\begin{verbatim}
square: FRAC INT -> FRAC INT
  Type: Void
\end{verbatim}

The curryRight operation creates a unary function from a binary one by providing a constant argument on the right.

\begin{verbatim}
square:= curryRight(power, 2)
theMap(MAPPKG3;curryRight;MBM;1!0,0)
  Type: (Fraction Integer -> Fraction Integer)
\end{verbatim}

Likewise, the curryLeft operation provides a constant argument on the left.

\begin{verbatim}
square 4
  16
  Type: Fraction Integer
\end{verbatim}

The constantRight operation creates (in a trivial way) a binary function from a unary one: constantRight(f) is the function g such that $g(a,b) = f(a)$.

\begin{verbatim}
squirrel:= constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
theMap(MAPPKG3;constantRight;MM;3!0)
  Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)
\end{verbatim}

Likewise, constantLeft(f) is the function g such that $g(a,b) = f(b)$.

\begin{verbatim}
squirrel(1/2, 1/3)
  1
  -
  4
\end{verbatim}
The curry operation makes a unary function nullary.

\[
\text{sixteen} := \text{curry} (\text{square}, \frac{4}{1})
\]

\[
\text{theMap} (\text{MAPPKG2}; \text{curry}; \text{MAM}; 2!0, 0)
\]

\[\text{sixteen}()
\]

\[16
\]

Type: Fraction Integer

The * operation constructs composed functions.

\[
\text{square2} := \text{square} * \text{square}
\]

\[
\text{theMap} (\text{MAPPKG3}; *, \text{MMM}; 6!0, 0)
\]

\[\text{square2} \ 3
\]

\[81
\]

Type: Fraction Integer

Use the ** operation to create functions that are n-fold iterations of other functions.

\[
\text{sc} (x: \text{FRAC INT}) : \text{FRAC INT} \equiv x + 1
\]

Type: Void

This is a list of Mapping objects.

\[
\text{incfns} := [\text{sc}**i \text{for i in 0..10}]
\]

\[
\text{[theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0), \text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0),
\text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0), \text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0),
\text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0), \text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0),
\text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0), \text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0),
\text{theMap} (\text{MAPPKG1}; **; \text{MMnN}; 6!0, 0)]
\]

Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

\[
\text{[f} \ 4 \text{for f in incfns]}
\]

\[
[4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
\]

Type: List Fraction Integer

Use the recur operation for recursion:

\[
g := \text{recur} \ f \ \text{means} \ g(n, x) \equiv f(n, f(n-1, \ldots f(1, x)))
\]

\[
\text{times} (n: \text{NNI}, \ \text{i: INT}) : \text{INT} \equiv n*i
\]
Type: Void

\[ r := \text{recur}(\text{times}) \]
\[ \text{theMap}(\text{MAPPG1}; \text{recur}; 2M; ? 10, 0) \]
Type: \(((\text{NonNegativeInteger}, \text{Integer}) \rightarrow \text{Integer})\)

This is a factorial function.

\[ \text{fact} := \text{curryRight}(r, 1) \]
\[ \text{theMap}(\text{MAPPG2}; \text{curryRight}; \text{MBM}; 110, 0) \]
Type: \((\text{NonNegativeInteger} \rightarrow \text{Integer})\)

\[ \text{fact} 4 \]
\[ 24 \]
Type: \text{PositiveInteger}

Constructed functions can be used within other functions.

\[ \text{mto2ton}(m, n) := \]
\[ \text{raiser} := \text{square}^n \]
\[ \text{raiser} \text{ m} \]

This is \(3^2 \times 3^2\).

\[ \text{mto2ton}(3, 3) \]
\[ 6561 \]
Type: \text{Fraction Integer}

Here \(\text{shiftfib}\) is a unary function that modifies its argument.

\[ \text{shiftfib}(r: \text{List INT}) : \text{INT} := \]
\[ t := r.1 \]
\[ r.1 := r.2 \]
\[ r.2 := r.2 + t \]
\[ t \]

Type: Void

By currying over the argument we get a function with private state.

\[ \text{fibinit}: \text{List INT} := [0, 1] \]
\[ [0, 1] \]
Type: \text{List Integer}

\[ \text{fibs} := \text{curry}(\text{shiftfib}, \text{fibinit}) \]
\[ \text{theMap}(\text{MAPPG2}; \text{curry}; \text{MAM}; 2!0, 0) \]
Type: \((() \rightarrow \text{Integer})\)

\[ \text{[fibs()} \text{for} \ i \ \text{in} \ 0..30] \]
\[ [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, \]
See Also:
  o )help MappingPackage1
  o )show MappingPackage2
  o )help MappingPackage3
  o )help MappingPackage4

---

MappingPackage2 (MAPPKG2)

Exports:
  const  constant  curry  diag

— package MAPPKG2 MappingPackage2 —

)abbrev package MAPPKG2 MappingPackage2
++ Description:
++ Various Currying operations.
MappingPackage2(A:SetCategory, C:SetCategory): MPcat == MPdef where
NNI     ==>  NonNegativeInteger

MPcat == with
  const: C   -> (A ->C)
    ++\spad{const c} is a function which produces \spad{c} when
    ++ applied to its argument.
  curry: (A ->C, A)   -> (()->C)
    ++\spad{cu(f,a)} is the function \spad{g}
such that \( g() = f \).

**constant**: \((() \to C) \to (A \to C)\)

\[ g = \text{the function } f \text{ such that } g(a) = f() \]

**diag**: \(((A,A) \to C) \to (A \to C)\)

\[ g = \text{the function } f \text{ such that } g(a) = f(a,a) \]

\[
\text{MPdef} \equiv \text{add}
\]

\[
\text{MappingPackageInternalHacks2}(A, C)
\]

\[
a: A \\
c: C \\
faa: A \to A \\
f0c: () \to C \\
fac: A \to C \\
faac: (A,A) \to C
\]

\[
\text{const } c = (a1:A) : C \mapsto \text{arg2}(a1, c) \\
\text{curry}(\text{fac}, a) = \text{fac } a \\
\text{constant } f0c = (a1:A) : C \mapsto \text{arg2}(a1, f0c()) \\
\text{diag } faac = (a1:A) : C \mapsto \text{faac}(a1, a1)
\]

---

**MAPPKG2.dotabb**

"MAPPKG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"MAPPKG2" -> "BASTYPE"
"MAPPKG2" -> "KOERCE"

---

package MAPPKG3 MappingPackage3

--- MappingPackage3.input ---

)set break resume
)spool MappingPackage3.output
)set message test on
)set message auto off
)clear all

--S 1 of 26
power(q: FRAC INT, n: INT): FRAC INT == q**n
--R
--R Function declaration power : (Fraction(Integer),Integer) -> Fraction
--R (Integer) has been added to workspace.
--R
--E 1

--S 2 of 26
power(2,3)
--R
--R Compiling function power with type (Fraction(Integer),Integer) ->
--R Fraction(Integer)
--R
--R (2) 8
--R
--E 2

--S 3 of 26
rewop := twist power
--R
--R
--R (3) theMap(MAPPKG3;twist;MM;5!0)
--R
--E 3

--S 4 of 26
rewop(3, 2)
--R
--R
--R (4) 8
--R
--E 4

--S 5 of 26
square: FRAC INT -> FRAC INT
--R
--R
--E 5

--S 6 of 26
square:= curryRight(power, 2)
--R
--R
--R (6) theMap(MAPPKG3;curryRight;MBM;1!0,0)
--R
--E 6
squirrel := constantRight(square) \$ MAPPKG3(FRAC INT, FRAC INT, FRAC INT)

squirrel(1/2, 1/3)

sixteen := curry(square, 4/1)

sixteen()

square2 := square * square
square2 3

sc(x: FRAC INT): FRAC INT == x + 1

incfns := [sc**i for i in 0..10]

[f 4 for f in incfns]

times(n:NNI, i:INT):INT == n*i
r := recur(times)

Compiling function times with type (NonNegativeInteger, Integer) -> Integer

(18) theMap(MAPPKG1; recur; 2M; 7!0, 0)
Type: ((NonNegativeInteger, Integer) -> Integer)

fact := curryRight(r, 1)

(19) theMap(MAPPKG3; curryRight; MBM; 1!0, 0)
Type: (NonNegativeInteger -> Integer)

fact 4

(20) 24
Type: PositiveInteger

mto2ton(m, n) ==
  raiser := square^n
  raiser m

Type: Void

mto2ton(3, 3)

Compiling function mto2ton with type (PositiveInteger, PositiveInteger) -> Fraction(Integer)

(22) 6561
Type: Fraction(Integer)

shiftfib(r: List INT) : INT ==
  t := r.1
  r.1 := r.2
  r.2 := r.2 + t
  t
--R Function declaration shiftfib : List(Integer) -> Integer has been
--R added to workspace.
Type: Void
--E 23

--S 24 of 26
fibinit: List INT := [0, 1]
--R
--R (24) [0,1]
Type: List(Integer)
--E 24

--S 25 of 26
fibs := curry(shiftfib, fibinit)
--R
--R Compiling function shiftfib with type List(Integer) -> Integer
--R
--R (25) theMap(MAPPKG2;curry;MAM;2!0,0)
Type: (() -> Integer)
--E 25

--S 26 of 26
[fibs() for i in 0..30]
--R
--R (26)
Type: List(Integer)
--E 26

--S 27 of 27
)show MappingPackage3
--R
--R MappingPackage3(A: SetCategory,B: SetCategory,C: SetCategory) is a package constructor
--R Abbreviation for MappingPackage3 is MAPPKG3
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MAPPKG3
--R
--R----------------------------- Operations -----------------------------
--R twist : ((A,B) -> C) -> ((B,A) -> C)
--R ?*? : ((B -> C),(A -> B)) -> (A -> C)
--R constantLeft : (B -> C) -> ((A,B) -> C)
--R constantRight : (A -> C) -> ((A,B) -> C)
--R curryLeft : (((A,B) -> C),A) -> (B -> C)
--R curryRight : (((A,B) -> C),B) -> (A -> C)
--R
Various Currying operations.

Function are objects of type Mapping. In this section we demonstrate some library operations from the packages MappingPackage1, MappingPackage2, and MappingPackage3 that manipulate and create functions. Some terminology: a nullary function takes no arguments, a unary function takes one argument, and a binary function takes two arguments.

We begin by creating an example function that raises a rational number to an integer exponent.

```lisp
power(q: FRAC INT, n: INT): FRAC INT == q**n
Type: Void
```

```lisp
power(2,3)
```

8
```
Type: Fraction Integer
```

The twist operation transposes the arguments of a binary function. Here rewop(a, b) is power(b, a).

```lisp
rewop := twist power
theMap(MAPPKG3;twist;MM;5!0)
```

```
Type: ((Integer,Fraction Integer) -> Fraction Integer)
```

This is $2^3$. 

```lisp
rewop(3, 2)
```

8
```
Type: Fraction Integer
```

Now we define square in terms of power.

```lisp
square: FRAC INT -> FRAC INT
Type: Void
```

The curryRight operation creates a unary function from a binary one by
providing a constant argument on the right.

```lisp
square := curryRight(power, 2)
theMap(MAPPKG3;curryRight;MBM;1!0,0)
Type: (Fraction Integer -> Fraction Integer)
```

Likewise, the curryLeft operation provides a constant argument on the left.

```lisp
square 4
16
Type: Fraction Integer
```

The constantRight operation creates (in a trivial way) a binary function from a unary one; constantRight(f) is the function g such that g(a,b) = f(a).

```lisp
squirrel := constantRight(square)$MAPPKG3(FRAC INT,FRAC INT,FRAC INT)
theMap(MAPPKG3;constantRight;MM;3!0)
Type: ((Fraction Integer,Fraction Integer) -> Fraction Integer)
```

Likewise, constantLeft(f) is the function g such that g(a,b) = f(b).

```lisp
squirrel(1/2, 1/3)
1
-
4
Type: Fraction Integer
```

The curry operation makes a unary function nullary.

```lisp
sixteen := curry(square, 4/1)
theMap(MAPPKG2;curry;HAM;2!0,0)
Type: (() -> Fraction Integer)
```

```lisp
sixteen()
16
Type: Fraction Integer
```

The * operation constructs composed functions.

```lisp
square2 := square*square
theMap(MAPPKG3;*;MM;6!0,0)
Type: (Fraction Integer -> Fraction Integer)
```

```lisp
square2 3
81
Type: Fraction Integer
```

Use the ** operation to create functions that are n-fold iterations of
other functions.

\[ sc(x: \text{FRAC INT}): \text{FRAC INT} == x + 1 \]
Type: Void

This is a list of Mapping objects.

\[ \text{incfns} := [sc**i \text{ for } i \text{ in } 0..10] \]
\[ \text{[theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0), \text{theMap}(\text{MAPPKG1};**;\text{MNniM};6!0,0)] \]
Type: List (Fraction Integer -> Fraction Integer)

This is a list of applications of those functions.

\[ [4,5,6,7,8,9,10,11,12,13,14] \]
Type: List Fraction Integer

Use the recur operation for recursion:

\[ g := \text{recur } f \text{ means } g(n,x) == f(n,f(n-1,\ldots f(1,x))). \]
\[ \text{times}(n:\text{NNI}, i:\text{INT}):\text{INT} == n*i \]
Type: Void

\[ r := \text{recur}(\text{times}) \]
\[ \text{theMap}(\text{MAPPKG1};\text{recur};2M;7!0,0) \]
Type: ((NonNegativeInteger,Integer) -> Integer)

This is a factorial function.

\[ \text{fact} := \text{curryRight}(r, 1) \]
\[ \text{theMap}(\text{MAPPKG3};\text{curryRight};\text{MBM};1!0,0) \]
Type: (NonNegativeInteger -> Integer)

\[ \text{fact} 4 \]
\[ 24 \]
Type: PositiveInteger

Constructed functions can be used within other functions.

\[ \text{mto2ton}(m, n) == \]
\[ \text{raiser} := \text{square}^n \]
\[ \text{raiser } m \]
Type: Void
This is $3^{2^3}$.

```
mto2ton(3, 3)
6561
```

Type: Fraction Integer

Here `shiftfib` is a unary function that modifies its argument.

```
shiftfib(r: List INT) : INT ==
t := r.1
r.1 := r.2
r.2 := r.2 + t
t
```

Type: Void

By currying over the argument we get a function with private state.

```
fibinit: List INT := [0, 1]
[0,1]
```

Type: List Integer

```
fibs := curry(shiftfib, fibinit)
theMap(MAPPKG2;curry;MAM;2!0,0)
```

Type: ((() -> Integer)

```
[fibs() for i in 0..30]
[0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 317811, 514229, 832040]
```

Type: List Integer

See Also:
- )help MappingPackage1
- )help MappingPackage2
- )show MappingPackage3
- )help MappingPackage4
MappingPackage3 (MAPPKG3)

Exports:
constantLeft constantRight curryLeft curryRight twist ?**?

— package MAPPKG3 MappingPackage3 —

)abbrev package MAPPKG3 MappingPackage3 ++ Description:
++ Various Currying operations.

MappingPackage3(A:SetCategory, B:SetCategory, C:SetCategory):_
MPcat == MPdef where
  NNI  ==> NonNegativeInteger

  MPcat == with
  curryRight: ((A,B)->C, B) -> (A ->C)
++ \spad{curryRight(f,b)} is the function \spad{g} such that
++ \spad{g a = f(a,b)}.
  curryLeft: ((A,B)->C, A) -> (B ->C)
++ \spad{curryLeft(f,a)} is the function \spad{g}
++ such that \spad{g b = f(a,b)}.
  constantRight: (A -> C)   -> ((A,B)->C)
++ \spad{constantRight(f)} is the function \spad{g}
++ such that \spad{g a,b= f a}.
  constantLeft: (B -> C)    -> ((A,B)->C)
++ \spad{constantLeft(f)} is the function \spad{g}
++ such that \spad{g a,b= f b}.
  twist: ((A,B)->C)   -> ((B,A)->C)
++ \spad{twist(f)} is the function \spad{g}
++ such that \spad{g a,b= f(b,a)}.
  "*": (B->C, A->B) -> (A->C)
++ \spad{f*g} is the function \spad{h}
++ such that \spad{h x= f(g x)}. 
MPdef == add

MappingPackageInternalHacks3(A, B, C)

a: A
b: B
c: C
faa: A -> A
f0c: () -> C
fac: A -> C
fbc: B -> C
fab: A -> B
fabc: (A,B) -> C
faac: (A,A) -> C

-- Fix left and right arguments as constants.
curryRight(fabc,b) == (a:A):C +-> fabc(a,b)
curryLeft(fabc,a) == (b:B):C +-> fabc(a,b)

-- Add left and right arguments which are ignored.
constantRight fac == (a:A, b:B):C +-> fac a
constantLeft fbc == (a:A, b:B):C +-> fbc b

-- Combinators to rearrange arguments.
twist fabc == (b:B, a:A):C +-> fabc(a,b)

-- Functional composition
fbc*fab == (a:A):C +-> comp(fbc,fab,a)

———

— MAPPKG3.dotabb ——
package MAPPKG4 MappingPackage4

— MappingPackage4.input —

)set break resume
)spool MappingPackage4.output
)set message test on
)set message auto off
)clear all

--S 1 of 22
p:=(x:EXPR(INT)):EXPR(INT)+->3*x
--R
--R  (1) theMap(Closure)
--R Type: (Expression Integer -> Expression Integer)
--E 1

--S 2 of 22
q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3
--R
--R  (2) theMap(Closure)
--R Type: (Expression Integer -> Expression Integer)
--E 2

--S 3 of 22
(p+q)(4)-(p(4)+q(4))
--R
--R  (3) 0
--R Type: Expression Integer
--E 3

--S 4 of 22
(p+q)(x)-(p(x)+q(x))
--R
--R  (4) 0
--R Type: Expression Integer
--E 4

--S 5 of 22
(p-q)(4)-(p(4)-q(4))
--R
--R  (5) 0
--R Type: Expression Integer
package MAPPKG4 MAPPINGPACKAGE4

--E 5

--S 6 of 22
(p-q)(x)-(p(x)-q(x))
--R
--R (6) 0
--R

--E 6

--S 7 of 22
(p*q)(4)-(p(4)*q(4))
--R
--R (7) 0
--R

--E 7

--S 8 of 22
(p*q)(x)-(p(x)*q(x))
--R
--R (8) 0
--R

--E 8

--S 9 of 22
(p/q)(4)-(p(4)/q(4))
--R
--R (9) 0
--R

--E 9

--S 10 of 22
(p/q)(x)-(p(x)/q(x))
--R
--R (10) 0
--R

--E 10

--S 11 of 22
r:=(x:INT):INT+-> (x*x*x)
--R
--R (11) theMap(Closure)
--R

--E 11

Type: Expression Integer

Type: Expression Integer

Type: Expression Integer

Type: Expression Integer

Type: Expression Integer

Type: Expression Integer

Type: Expression Integer

Type: (Integer -> Integer)
\[ s := (y: \mathbb{INT}) : \rightarrow (y*y + 3) \]  
\[ (r+s)(4) - (r(4) + s(4)) \]  
\[ (r-s)(4) - (r(4) - s(4)) \]  
\[ (r*s)(4) - (r(4) * s(4)) \]  
\[ t := (x: \mathbb{INT}) : \mathbb{EXPR} \rightarrow (x*x*x) \]  
\[ (t/u)(4) - (t(4)/u(4)) \]
--R
--R (18) 0
--R Type: Expression Integer
--E 18

--S 19 of 22
h:=\langle x:EXPR(INT)\rangle:EXPR(INT)+\rightarrow 1
--R
--R (19) theMap(Closure)
--R Type: (Expression Integer \rightarrow Expression Integer)
--E 19

--S 20 of 22
(p/h)(x)
--R
--R (20) 3x
--R Type: Expression Integer
--E 20

--S 21 of 22
(q/h)(x)
--R
--R (21) 2x + 3
--R Type: Expression Integer
--E 21

--S 22 of 22
)show MappingPackage4
--R
--R MappingPackage4(A: SetCategory,B: Ring) is a package constructor
--R Abbreviation for MappingPackage4 is MAPPKG4
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MAPPKG4
--R
--R---------------------------------------- Operations ----------------------------------------
--R ?*? : ((A \rightarrow B),(A \rightarrow B)) \rightarrow (A \rightarrow B)
--R ?+? : ((A \rightarrow B),(A \rightarrow B)) \rightarrow (A \rightarrow B)
--R ?-? : ((A \rightarrow B),(A \rightarrow B)) \rightarrow (A \rightarrow B)
--R ?/? : ((A \rightarrow Expression(Integer)),(A \rightarrow Expression(Integer))) \rightarrow (A \rightarrow Expression(Integer))
--R
--E 22

)spool
)lisp (bye)
CHAPTER 14. CHAPTER M

— MappingPackage4.help —

====================================================================
MappingPackage examples
====================================================================

Given functions \( f \) and \( g \), returns the applicable closure

We can construct some simple maps that take a variable \( x \)
into an equation:

\[
p := (x:EXPR(INT))::EXPR(INT) +\rightarrow 3\times
\]
\[
q := (x:EXPR(INT))::EXPR(INT) +\rightarrow 2\times + 3
\]

Now we can do the four arithmetic operations, +, -, *, / on these
newly constructed mappings. Since the maps are from the domain
Expression Integer to the same domain we can also use symbolic
values for the argument. All of the following will return 0,
showing that function composition is equivalent to the result
of doing the operations individually.

\[(p+q)(4)-(p(4)+q(4))\]
\[(p+q)(x)-(p(x)+q(x))\]
\[(p-q)(4)-(p(4)-q(4))\]
\[(p-q)(x)-(p(x)-q(x))\]
\[(p\times q)(4)-(p(4)\times q(4))\]
\[(p\times q)(x)-(p(x)\times q(x))\]
\[(p/q)(4)-(p(4)/q(4))\]
\[(p/q)(x)-(p(x)/q(x))\]

We can construct simple maps from Integer to Integer but this
limits our ability to do division.

\[
r := (x:INT) :: INT +\rightarrow (x\times x\times x)
\]
\[
s := (y:INT) :: INT +\rightarrow (y\times y+3)
\]

Again, all of these will return 0:

\[(r+s)(4)-(r(4)+s(4))\]
\[(r-s)(4)-(r(4)-s(4))\]
\[(r\times s)(4)-(r(4)\times s(4))\]

If we want to do division with Integer inputs we create the
appropriate map:

\[
t := (x:INT) :: EXPR(INT) +\rightarrow (x\times x\times x)
\]
u:=(y:INT):EXPR(INT)+-> (y*y+3)
(t/u)(4)-(t(4)/u(4))

We can even recover the original functions if we make a map that always returns the constant 1:

h:=(x:EXPR(INT)):EXPR(INT)+->1

theMap(Closure)

Type: (Expression Integer -> Expression Integer)

(p/h)(x)

3x

Type: Expression Integer

(q/h)(x)

2x + 3

Type: Expression Integer

See Also:
o )show MappingPackage1
o )help MappingPackage2
o )help MappingPackage3
o )help MappingPackage4

MappingPackage4 (MAPPKG4)

Exports:

?*?  ?+?  ?-?  ?/?
chapter M

| package MAPPKG4 MappingPackage4 |

)abbrev package MAPPKG4 MappingPackage4
++ Author: Timothy Daly
++ Description:
++ Functional Composition.
++ Given functions f and g, returns the applicable closure

MappingPackage4(A:SetCategory, B:Ring):
with
  "+": (A->B, A->B) -> (A->B)
  ++ \spad(+) does functional addition
  ++
  ++X f:=(x:INT):INT +-> 3*x
  ++X g:=(x:INT):INT +-> 2*x+3
  ++X (f+g)(4)
  "-": (A->B, A->B) -> (A->B)
  ++ \spad(+) does functional addition
  ++
  ++X f:=(x:INT):INT +-> 3*x
  ++X g:=(x:INT):INT +-> 2*x+3
  ++X (f-g)(4)
  "*": (A->B, A->B) -> (A->B)
  ++ \spad(+) does functional addition
  ++
  ++X f:=(x:INT):INT +-> 3*x
  ++X g:=(x:INT):INT +-> 2*x+3
  ++X (f*g)(4)
  "/": (A->Expression(Integer), A->Expression(Integer)) -> (A->Expression(Integer))
  ++ \spad(+) does functional addition
  ++
  ++X p:=(x:EXPR(INT)):EXPR(INT)+->3*x
  ++X q:=(x:EXPR(INT)):EXPR(INT)+->2*x+3
  ++X (p/q)(4)
  ++X (p/q)(x)

== add
fab ==> (A -> B)
faei ==> (A -> Expression(Integer))

funcAdd(g:fab,h:fab,x:A):B == ((g x) + (h x))$B
(a:fab)+(b:fab) == c +-> funcAdd(a,b,c)
funcSub(g:fab,h:fab,x:A):B == ((g x) - (h x))$B
(a:fab)-(b:fab) == c +-> funcSub(a,b,c)
funcMul(g:fab,h:fab,x:A):B == ((g x) * (h x))$B
(a:fab)*(b:fab) == c += funcMul(a,b,c)

funcDiv(g:faei,h:faei,x:A):Expression(Integer)
  == ((g x) / (h x))$Expression(Integer)

(a:faei)/(b:faei) == c += funcDiv(a,b,c)

-- MAPPKG4.dotabb --

"MAPPKG4" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MAPPKG4"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"MAPPKG4" -> "PID"
"MAPPKG4" -> "OAGROUP"

package MATCAT2 MatrixCategoryFunctions2

-- MatrixCategoryFunctions2.input --

)set break resume
)sys rm -f MatrixCategoryFunctions2.output
)spool MatrixCategoryFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show MatrixCategoryFunctions2
--R
--R MatrixCategoryFunctions2(R1: Ring,Row1: FiniteLinearAggregate(R1),Col1: FiniteLinearAggregate(R1),M1: MatrixCategory(R1,Row1,Col1),Row2: FiniteLinearAggregate(R2),Col2: FiniteLinearAggregate(R2),M2: MatrixCategory(R2,Row2,Col2)) is a package constructor
--R Abbreviation for MatrixCategoryFunctions2 is MATCAT2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MATCAT2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((R1 -> R2),M1) -> M2
--R map : ((R1 -> Union(R2,"failed")),M1) -> Union(M2,"failed")
--R reduce : (((R1,R2) -> R2),M1,R2) -> R2
--R
--E 1
MatrixCategoryFunctions2 provides functions between two matrix domains. The functions provided are map and reduce.

See Also:
o )show MatrixCategoryFunctions2

MatrixCategoryFunctions2 (MATCAT2)

Exports:
map  reduce

-- package MATCAT2 MatrixCategoryFunctions2 --

)abbrev package MATCAT2 MatrixCategoryFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 21 November 1989
++ Date Last Updated: 21 March 1994
++ Description:
++ \spadtype{MatrixCategoryFunctions2} provides functions between two matrix ++ domains. The functions provided are \spadfun{map} and \spadfun{reduce}.

MatrixCategoryFunctions2(R1,Row1,Col1,M1,R2,Row2,Col2,M2):_
    Exports == Implementation where
    R1 : Ring
    Row1 : FiniteLinearAggregate R1
    Col1 : FiniteLinearAggregate R1
    M1 : MatrixCategory(R1,Row1,Col1)
    R2 : Ring
    Row2 : FiniteLinearAggregate R2
    Col2 : FiniteLinearAggregate R2
    M2 : MatrixCategory(R2,Row2,Col2)

    Exports ==> with
    map: (R1 -> R2,M1) -> M2
    ++ \spad{map(f,m)} applies the function f to the elements of the matrix m.
    map: (R1 -> Union(R2,"failed"),M1) -> Union(M2,"failed")
    ++ \spad{map(f,m)} applies the function f to the elements of the matrix m.
    reduce: ((R1,R2) -> R2,M1,R2) -> R2
    ++ \spad{reduce(f,m,r)} returns a matrix n where
    ++ \spad{n[i,j] = f(m[i,j],r)} for all indices i and j.

    Implementation ==> add
    minr => minRowIndex
    maxr => maxRowIndex
    minc => minColIndex
    maxc => maxColIndex

    map(f:(R1->R2),m:M1):M2 ==
    ans : M2 := new(nrows m,ncols m,0)
    for i in minr(m)..maxr(m) for k in minr(ans)..maxr(ans) repeat
        for j in minc(m)..maxc(m) for l in minc(ans)..maxc(ans) repeat
            qsetelt_!(ans,k,l,f qelt(m,i,j))
    ans

    map(f:(R1 -> Union(R2,"failed")),m:M1):Union(M2,"failed") ==
    ans : M2 := new(nrows m,ncols m,0)
    for i in minr(m)..maxr(m) for k in minr(ans)..maxr(ans) repeat
        for j in minc(m)..maxc(m) for l in minc(ans)..maxc(ans) repeat
            (r := f qelt(m,i,j)) = "failed" => return "failed"
            qsetelt_(!ans,k,l,r::R2)
    ans

    reduce(f,m,ident) ==
    s := ident
    for i in minr(m)..maxr(m) repeat
        for j in minc(m)..maxc(m) repeat
            s := f(qelt(m,i,j),s)
    s
package MCDEN MatrixCommonDenominator

MatrixCommonDenominator.input

)set break resume
)sys rm -f MatrixCommonDenominator.output
)spool MatrixCommonDenominator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MatrixCommonDenominator
--R
--R MatrixCommonDenominator(R: IntegralDomain,Q: QuotientFieldCategory(R)) is a package constructor
--R Abbreviation for MatrixCommonDenominator is MCDEN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MCDEN
--R
--R-------------------------------- Operations --------------------------------
--R commonDenominator : Matrix(Q) -> R
--R clearDenominator : Matrix(Q) -> Matrix(R)
--R splitDenominator : Matrix(Q) -> Record(num: Matrix(R),den: R)
--R
--E 1

)spool
)lisp (bye)

MatrixCommonDenominator.help
MatrixCommonDenominator provides functions to compute the common denominator of a matrix of elements of the quotient field of an integral domain.

See Also:

- )show MatrixCommonDenominator

---

MatrixCommonDenominator (MCDEN)

Exports:
- commonDenominator
- clearDenominator
- splitDenominator

— package MCDEN MatrixCommonDenominator —

MatrixCommonDenominator(R, Q):Exports == Implementation where
R: IntegralDomain
Q: QuotientFieldCategory R
VR ==> Vector R
VQ ==> Vector Q

Exports ==> with
  commonDenominator: Matrix Q -> R
    ++ commonDenominator(q) returns a common denominator d for
    ++ the elements of q.
  clearDenominator : Matrix Q -> Matrix R
    ++ clearDenominator(q) returns p such that \spad{q = p/d} where d is
    ++ a common denominator for the elements of q.
  splitDenominator : Matrix Q -> Record(num: Matrix R, den: R)
    ++ splitDenominator(q) returns \spad{[p, d]} such that \spad{q = p/d} and d
    ++ is a common denominator for the elements of q.

Implementation ==> add
import ListFunctions2(Q, R)
import MatrixCategoryFunctions2(Q,VQ,VQ,Matrix Q,R,VR,VR,Matrix R)

clearDenominator m ==
  d := commonDenominator m
  map(x +-> numer(d*x), m)

splitDenominator m ==
  d := commonDenominator m
  [map(x +-> numer(d*x), m), d]

if R has GcdDomain then
  commonDenominator m == lcm map(denom, parts m)
else
  commonDenominator m == reduce("*",map(denom, parts m),1)$List(R)

—— MCDEN.dotabb ——

"MCDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MCDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MCDEN" -> "PFECAT"

——

package MATLIN MatrixLinearAlgebraFunctions

—— MatrixLinearAlgebraFunctions.input ——
--S 1 of 1

)set break resume
)sys rm -f MatrixLinearAlgebraFunctions.output
)spool MatrixLinearAlgebraFunctions.output
)set message test on
)set message auto off
)clear all

--R MatrixLinearAlgebraFunctions(R: CommutativeRing,Row: FiniteLinearAggregate(R),Col: FiniteLinearAggregate(R)) is a package constructor
--R Abbreviation for MatrixLinearAlgebraFunctions is MATLIN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MATLIN
--R
--R------------------------------- Operations --------------------------------
--R determinant : M -> R
e1Row1! : (M,Integer,Integer) -> M
e1Row2! : (M,R,Integer,Integer) -> M
minordet : M -> R
rowEchelon : M -> M if R has EUCDOM
adjoint : M -> Record(adjMat: M,detMat: R) if R has INTDOM
e1Column2! : (M,R,Integer,Integer) -> M
fractionFreeGauss! : M -> M if R has INTDOM
inverse : M -> Union(M,"failed") if R has FIELD
invertIfCan : M -> Union(M,"failed") if R has INTDOM
normalizedDivide : (R,R) -> Record(quotient: R,remainder: R) if R has EUCDOM
nullSpace : M -> List(Col) if R has INTDOM
nullity : M -> NonNegativeInteger if R has INTDOM
rank : M -> NonNegativeInteger if R has INTDOM
--R

)spool
)lisp (bye)

-- MatrixLinearAlgebraFunctions.help --

====================================================================
MatrixLinearAlgebraFunctions examples
====================================================================

MatrixLinearAlgebraFunctions provides functions to compute inverses
and canonical forms.

See Also:
o )show MatrixLinearAlgebraFunctions
MatrixLinearAlgebraFunctions (MATLIN)

Exports:
adjoint       determinant    elColumn2!   elRow1!   elRow2!
fractionFreeGauss!    inverse    invertIfCan    minordet    normalizedDivide
nullSpace          nullity     rank           rowEchelon

| package MATLIN MatrixLinearAlgebraFunctions |

)abbrev package MATLIN MatrixLinearAlgebraFunctions
++ Author: Clifton J. Williamson, P. Gianni
++ Date Created: 13 November 1989
++ Date Last Updated: December 1992
++ Description:
++ \spadtype{MatrixLinearAlgebraFunctions} provides functions to compute
++ inverses and canonical forms.

MatrixLinearAlgebraFunctions(R,Row,Col,M):Exports == Implementation where
  R : CommutativeRing
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R,Row,Col)
  I ==> Integer

Exports ==> with

  determinant: M -> R
  ++ \spad{determinant(m)} returns the determinant of the matrix \spad{m}.
  ++ an error message is returned if the matrix is not square.
  minordet: M -> R
  ++ \spad{minordet(m)} computes the determinant of the matrix \spad{m} using
  ++ minors. Error: if the matrix is not square.
  elRow1!: (M,I,I) -> M
  ++ elRow1!(\spad{m},i,j) swaps rows \spad{i} and \spad{j} of matrix \spad{m} : elementary operation
++ of first kind
elRow2! : (M,R,I,I) -> M
++ elRow2!(m,a,i,j) adds to row i a*row(m,j) : elementary operation of
++ second kind. (i ^=j)
elColumn2! : (M,R,I,I) -> M
++ elColumn2!(m,a,i,j) adds to column i a*column(m,j) : elementary
++ operation of second kind. (i ^=j)

if R has IntegralDomain then
rank: M -> NonNegativeInteger
++ \spad{rank(m)} returns the rank of the matrix m.
nullity: M -> NonNegativeInteger
++ \spad{nullity(m)} returns the nullity of the matrix m. This is
++ the dimension of the null space of the matrix m.
nullSpace: M -> List Col
++ \spad{nullSpace(m)} returns a basis for the null space of the
++ matrix m.

fractionFreeGauss! : M -> M
++ \spad{fractionFreeGauss(m)} performs the fraction
++ free gaussian elimination on the matrix m.
invertIfCan : M -> Union(M,"failed")
++ \spad{invertIfCan(m)} returns the inverse of m over R
adjoint : M -> Record(adjMat:M, detMat:R)
++ \spad{adjoint(m)} returns the adjoint matrix of m (i.e. the matrix
++ n such that m*n = determinant(m)*id) and the determinant of m.

if R has EuclideanDomain then
rowEchelon: M -> M
++ \spad{rowEchelon(m)} returns the row echelon form of the matrix m.

normalizedDivide: (R, R) -> Record(quotient:R, remainder:R)
++ normalizedDivide(n,d) returns a normalized quotient and
++ remainder such that consistently unique representatives
++ for the residue class are chosen, e.g. positive remainders

if R has Field then
inverse: M -> Union(M,"failed")
++ \spad{inverse(m)} returns the inverse of the matrix.
++ If the matrix is not invertible, "failed" is returned.
++ Error: if the matrix is not square.

Implementation ==> add

rowAllZeroes?: (M,I) -> Boolean
rowAllZeroes?(x,i) ==
-- determines if the ith row of x consists only of zeroes
-- internal function: no check on index i
for j in minColIndex(x)..maxColIndex(x) repeat
  qelt(x,i,j) ^= 0 => return false
true
colAllZeroes?: (M,I) -> Boolean
colAllZeroes?(x,j) ==
  -- determines if the ith column of x consists only of zeroes
  -- internal function: no check on index j
  for i in minRowIndex(x)..maxRowIndex(x) repeat
    qelt(x,i,j) ^= 0 => return false
  true

minorDet:(M,I,List I,I,PrimitiveArray(Union(R,"uncomputed")))-> R
minorDet(x,m,l,i,v) ==
  z := v.m
  z case R => z
  ans : R := 0; rl : List I := nil()
  j := first l; l := rest l; pos := true
  minR := minRowIndex x; minC := minColIndex x;
  repeat
    if qelt(x,j + minR,i + minC) ^= 0 then
      ans :=
      md := minorDet(x,m - 2**(j :: NonNegativeInteger),_ concat_!(reverse rl,l),i + 1,v) *_
      qelt(x,j + minR,i + minC)
      pos => ans + md
      ans - md
    null l =>
      v.m := ans
      return ans
    pos := not pos; rl := cons(j,rl); j := first l; l := rest l

minordet x ==
  (ndim := nrows x) ^= (ncols x) =>
    error "determinant: matrix must be square"
  -- minor expansion with (s---loads of) memory
  n1 : I := ndim - 1
  v : PrimitiveArray(Union(R,"uncomputed")) :=
    new((2**ndim - 1) :: NonNegativeInteger,"uncomputed")
  minR := minRowIndex x; maxC := maxColIndex x
  for i in 0..n1 repeat
    qsetelt_!(v,(2**i - 1),qelt(x,i + minR,maxC))
  minorDet(x, 2**ndim - 2, [i for i in 0..n1], 0, v)

  -- elementary operation of first kind: exchange two rows --
elRow1!(m:M,i:I,j:I) : M ==
  vec:=row(m,i)
  setRow!(m,i,row(m,j))
  setRow!(m,j,vec)
  m

  -- elementary operation of second kind: add to row i--
  -- a*row j (i^=j) --
elRow2!(m : M,a:R,i:I,j:I) : M ==
vec:= map((r1:R):R +-> a*r1,row(m,j))
vec:=map("+",row(m,i),vec)
setRow!(m,i,vec)
m

-- elementary operation of second kind: add to column i --
-- a*column j (i!=j) --

elColumn2!(m : M,a:R,i:I,j:I) : M ==
vec:= map((r1:R):R +-> a*r1,column(m,j))
vec:=map("+",column(m,i),vec)
setColumn!(m,i,vec)
m

if R has IntegralDomain then
-- Fraction-Free Gaussian Elimination

fractionFreeGauss! x ==
(ndim := nrows x) = 1 => x
ans := b := 1$R
minR := minRowIndex x; maxR := maxRowIndex x
minC := minColIndex x; maxC := maxColIndex x
i := minR
for j in minC..maxC repeat
  if qelt(x,i,j) = 0 then -- candidate for pivot = 0
    rown := minR - 1
    for k in (i+1)..maxR repeat
      if qelt(x,k,j) ^= 0 then
        rown := k -- found a pivot
        leave
      if rown > minR - 1 then
        swapRows_!(x,i,rown)
        ans := -ans
        (c := qelt(x,i,j)) = 0 => "next j" -- try next column
    for k in (i+1)..maxR repeat
      if qelt(x,k,j) = 0 then
        for l in (j+1)..maxC repeat
          qsetelt_!(x,k,l,(c * qelt(x,k,l) exquo b) :: R)
          else
            pv := qelt(x,k,j)
            qsetelt_!(x,k,j,0)
        for l in (j+1)..maxC repeat
          val := c * qelt(x,k,l) - pv * qelt(x,i,l)
          qsetelt_!(x,k,l,(val exquo b) :: R)
        b := c
      (i := i+1)>maxR => leave
    if ans=-1 then
      lasti := i-1
      for j in 1..maxC repeat x(lasti, j) := -x(lasti,j)
x
--
lastStep(x:M) : M ==
  ndim := nrows x
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := minC+ndim -1
  exCol:=maxColIndex x
  det:=x(maxR,maxC)
  maxR1:=maxR-1
  maxC1:=maxC+1
  minC1:=minC+1
  iRow:=maxR
  iCol:=maxC-1
  for i in maxR1..1 by -1 repeat
    for j in maxC1..exCol repeat
      ss:=[x(i,iCol+k)*x(i+k,j) for k in 1..(maxR-i)]
      x(i,j) := _exquo((det * x(i,j) - ss),x(i,iCol))::R
    iCol:=iCol-1
  subMatrix(x,minR,maxR,maxC1,exCol)

invertIfCan(y) ==
  (nr:=nrows y) ^= (ncols y) =>
    error "invertIfCan: matrix must be square"
  adjRec := adjoint y
  (den:=recip(adjRec.detMat)) case "failed" => "failed"
  den::R * adjRec.adjMat

adjoint(y) ==
  (nr:=nrows y) ^= (ncols y) => error "adjoint: matrix must be square"
  maxR := maxRowIndex y
  maxC := maxColIndex y
  x := horizConcat(copy y,scalarMatrix(nr,1$R))
  ffr:= fractionFreeGauss!(x)
  det:=ffr(maxR,maxC)
  [lastStep(ffr),det]

if R has Field then
  VR ==> Vector R
  IMATLIN ==> InnerMatrixLinearAlgebraFunctions(R,Row,Col,M)
  MMATLIN ==> InnerMatrixLinearAlgebraFunctions(R,VR,VR,Matrix R)
  FLA2 ==> FiniteLinearAggregateFunctions2(R,VR,Col)
  MAT2 ==> MatrixCategoryFunctions2(R,Row,Col,M,R,VR,VR,Matrix R)
  rowEchelon y == rowEchelon(y)$IMATLIN
  rank y == rank(y)$IMATLIN
  nullity y == nullity(y)$IMATLIN
  determinant y == determinant(y)$IMATLIN
  inverse y == inverse(y)$IMATLIN
  if Col has shallowlyMutable then
    nullSpace y == nullSpace(y)$IMATLIN
else
  nullSpace y ==
  [map((r1:R):R +-> r1, v)$FLA2
   for v in nullSpace(map((r2:R):R +-> r2, y)$MAT2)$MMATLIN]

else if R has IntegralDomain then
  QF    ==> Fraction R
  Row2  ==> Vector QF
  Co12  ==> Vector QF
  M2    ==> Matrix QF
  IMATQF ==> InnerMatrixQuotientFieldFunctions(R,Row,Col,M,QF,Row2,Co12,M2)

nullSpace m == nullSpace(m)$IMATQF

determinant y ==
  (nrows y) ^= (ncols y) => error "determinant: matrix must be square"
  fm:=fractionFreeGauss!(copy y)
  fm(maxRowIndex fm,maxColIndex fm)

rank x ==
  y :=
  (rk := nrows x) > (rh := ncols x) =>
    rk := rh
    transpose x
  copy x
  y := fractionFreeGauss! y
  i := maxRowIndex y
  while rk > 0 and rowAllZeroes?(y,i) repeat
    i := i - 1
    rk := (rk - 1) :: NonNegativeInteger
  rk :: NonNegativeInteger

nullity x == (ncols x - rank x) :: NonNegativeInteger

if R has EuclideanDomain then
  if R has IntegerNumberSystem then
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
      qr := divide(n, d)
      qr.remainder >= 0 => qr
      d > 0 =>
        qr.remainder := qr.remainder + d
        qr.quotient := qr.quotient - 1
        qr
      qr.remainder := qr.remainder - d
      qr.quotient := qr.quotient + 1
      qr
  else
    normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
      divide(n, d)

rowEchelon y ==
  x := copy y
  minR := minRowIndex x; maxR := maxRowIndex x
  minC := minColIndex x; maxC := maxColIndex x
  n := minR - 1
  i := minR
  for j in minC..maxC repeat
    if i > maxR then leave x
    n := minR - 1
    xnj := R
    for k in i..maxR repeat
      if not zero?(xkj := qelt(x, k, j)) and ((n = minR - 1) _
        or sizeLess?(xkj, xnj)) then
        n := k
        xnj := xkj
      end
    end
    n = minR - 1 => "next j"
    swapRows_!(x, i, n)
  end
  for k in (i+1)..maxR repeat
    qelt(x, k, j) = 0 => "next k"
    aa := extendedEuclidean(qelt(x, i, j), qelt(x, k, j))
    (a, b, d) := (aa.coef1, aa.coef2, aa.generator)
    b1 := (qelt(x, i, j) exquo d) :: R
    a1 := (qelt(x, k, j) exquo d) :: R
    -- a*b1 + a1*b = 1
    for k1 in (j+1)..maxC repeat
      val1 := a * qelt(x, i, k1) + b * qelt(x, k, k1)
      val2 := -a1 * qelt(x, i, k1) + b1 * qelt(x, k, k1)
      qsetelt_!(x, i, k1, val1); qsetelt_!(x, k, k1, val2)
      qsetelt_!(x, i, j, d); qsetelt_!(x, k, j, 0)
    end
    un := unitNormal qelt(x, i, j)
    qsetelt_!(x, i, j, un.canonical)
    if un.associate ^= 1 then for jj in (j+1)..maxC repeat
      qsetelt_!(x, i, jj, un.associate * qelt(x, i, jj))
    end
  end
  xij := qelt(x, i, j)
  for k in minR..(i-1) repeat
    qelt(x, k, j) = 0 => "next k"
    qr := normalizedDivide(qelt(x, k, j), xij)
    qsetelt_!(x, k, j, qr.remainder)
    for k1 in (j+1)..maxC repeat
      qsetelt_!(x, k, k1, qelt(x, k, k1) - qr.quotient * qelt(x, i, k1))
    end
    i := i + 1
  end
  x

else determinant x == minordet x
package MAMA MatrixManipulation

-- MatrixManipulation.input --

)set break resume
)sys rm -f MatrixManipulation.output
)spool MatrixManipulation.output
)set message test on
)set message auto off
)clear all

-- S 1 of 44
M := matrix([[a,b,c],[d,e,f],[g,h,i]])
--R
--R +a b c+
--R |   |
--R (1) |d e f|
--R |   |
--R +g h i+  Type: Matrix(Polynomial(Integer))
--E 1

-- S 2 of 44
element(M, 2,2)
--R
--R (2) [e]  Type: Matrix(Fraction(Polynomial(Integer)))
--E 2

-- S 3 of 44
aRow(M, 1)
--R
--R (3) [a b c]  Type: Matrix(Fraction(Polynomial(Integer)))
-- E 3
-- S 4 of 44
aRow(M, 2)
-- R
-- R
-- R (4) [d e f]
-- R Type: Matrix(Fraction(Polynomial(Integer))
-- E 4

-- S 5 of 44
aColumn(M, 2)
-- R
-- R
-- R +b+
-- R | |
-- R (5) |e|
-- R | |
-- R +h+
-- R Type: Matrix(Fraction(Polynomial(Integer))
-- E 5

-- S 6 of 44
aColumn(M, 3)
-- R
-- R
-- R +c+
-- R | |
-- R (6) |f|
-- R | |
-- R +i+
-- R Type: Matrix(Fraction(Polynomial(Integer))
-- E 6

-- S 7 of 44
rows(M, [1,2])
-- R
-- R
-- R a b c+
-- R (7) | |
-- R +d e f+
-- R Type: Matrix(Fraction(Polynomial(Integer))
-- E 7

-- S 8 of 44
rows(M, [1,3])
-- R
-- R
-- R a b c+
-- R (8) | |
--R +g h i+  
--R                     Type: Matrix(Fraction(Polynomial(Integer)))  
--E 8

--S 9 of 44  
rows(M, [3,2])  
--R  
--R +g h i+  
--R (9) |   |  
--R +d e f+  
--R                     Type: Matrix(Fraction(Polynomial(Integer)))  
--E 9

--S 10 of 44  
rows(M, 2..3)  
--R  
--R +d e f+  
--R (10) |   |  
--R +g h i+  
--R                     Type: Matrix(Fraction(Polynomial(Integer)))  
--E 10

--S 11 of 44  
columns(M, [1,2])  
--R  
--R +a b+  
--R |   |  
--R (11) |d e|  
--R |   |  
--R +g h+  
--R                     Type: Matrix(Fraction(Polynomial(Integer)))  
--E 11

--S 12 of 44  
columns(M, [1,3])  
--R  
--R +a c+  
--R |   |  
--R (12) |d f|  
--R |   |  
--R +g i+  
--R                     Type: Matrix(Fraction(Polynomial(Integer)))  
--E 12

--S 13 of 44  
columns(M, [3,2])
```
--R
--R
--R +c b+
--R | |
--R (13) |f e|
--R | |
--R +i h+
--R

Type: Matrix(Fraction(Polynomial(Integer)))

--E 13

--S 14 of 44
columns(M, 1..2)
--R
--R
--R +a b+
--R | |
--R (14) |d e|
--R | |
--R +g h+
--R

Type: Matrix(Fraction(Polynomial(Integer)))

--E 14

--S 15 of 44
subMatrix(M, [1,2],[1,2])
--R
--R
--R +a b+
--R (15) | |
--R +d e+
--R

Type: Matrix(Fraction(Polynomial(Integer)))

--E 15

--S 16 of 44
subMatrix(M, [1,3],[1,3])
--R
--R
--R +a c+
--R (16) | |
--R +g i+
--R

Type: Matrix(Fraction(Polynomial(Integer)))

--E 16

--S 17 of 44
diagonalMatrix(M)
--R
--R
--R +a 0 0+
--R | |
--R (17) |0 e 0|
--R | |
```
diagonalMatrix(M, 1)

\begin{align*}
\begin{pmatrix}
0 & 0 & f \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\end{align*}

Type: Matrix(Fraction(Polynomial(Integer)))

diagonalMatrix(M, 2)

\begin{align*}
\begin{pmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\end{align*}

Type: Matrix(Fraction(Polynomial(Integer)))

diagonalMatrix(M, -1)

\begin{align*}
\begin{pmatrix}
d & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\end{align*}

Type: Matrix(Fraction(Polynomial(Integer)))

diagonalMatrix(M, -2)

\begin{align*}
\begin{pmatrix}
g & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{pmatrix}
\end{align*}

Type: Matrix(Fraction(Polynomial(Integer)))
bandMatrix(M, [-1,1])

+0  b  0+
|     |
(22)  |d  0  f|
|     |
+0  h  0+

Type: Matrix(Fraction(Polynomial(Integer)))

bandMatrix(M, [-1,0,1])

+a  b  0+
|     |
(23)  |d  e  f|
|     |
+0  h  i+

Type: Matrix(Fraction(Polynomial(Integer)))

bandMatrix(M, -1..1)

+a  b  0+
|     |
(24)  |d  e  f|
|     |
+0  h  i+

Type: Matrix(Fraction(Polynomial(Integer)))

-- Build a larger matrix

A := matrix([[a]])

[25]  [a]

Type: Matrix(Polynomial(Integer))

B := matrix([[b]])
--R
--R
--R (26) [b]  
--R Type: Matrix(Polynomial(Integer))
--E 26

--S 27 of 44
C := matrix([[c]])
--R
--R
--R (27) [c]  
--R Type: Matrix(Polynomial(Integer))
--E 27

--S 28 of 44
A11 := element(M, 3,3)
--R
--R
--R (28) [i]  
--R Type: Matrix(Fraction(Polynomial(Integer)))
--E 28

--S 29 of 44
A12 := horizConcat([A,B,C])
--R
--R
--R (29) [a b c]  
--R Type: Matrix(Fraction(Polynomial(Integer)))
--E 29

--S 30 of 44
A21 := vertConcat([A,B,C])
--R
--R
--R +a+  
--R | |
--R (30) |b|  
--R | |
--R +c+  
--R Type: Matrix(Fraction(Polynomial(Integer)))
--E 30

--S 31 of 44
E := blockConcat([[A11,A12],[A21,M]])
--R
--R
--R +i a b c+  
--R | |
--R (31) |a a b c|  
--R | |
--R |b d e f|
--R |
--R +c g h i+ Type: Matrix(Fraction(Polynomial(Integer)))
--E 31

-- Split again

--S 32 of 44
vertSplit(E, 2)
--R
--R
--R +i a b c+ +b d e f+
--R (32) [I I I I]
--R +a a b c+ +c g h i+
--R Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 32

--S 33 of 44
horizSplit(E, 2)
--R
--R
--R +i a+ +b c+ +b d e f+
--R (33) [I I I I]
--R +a a| b c|
--R (32) [I I I I]
--R +b d| e f|
--R +c g+ +h i+ Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 33

--S 34 of 44
vertSplit(E, [1,2,1])
--R
--R
--R +i a+ +b c+ +b d e f+
--R (34) [[i a b c],[c g h i]]
--R +a a b c+ +b d e f+
--R Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 34

--S 35 of 44
horizSplit(E, [2,2])
--R
--R
--R +i a+ +b c+ +b d e f+
--R (35) [[I I I I]
--R |a a| b c|
--R +c g+ +h i+
--R |b d| |e f|
--R | | | |
--R +c g+ +h i+
--R Type: List(Matrix(Fraction(Polynomial(Integer))))
--E 35

--S 36 of 44
blockSplit(E, 2,2)
--R
--R
--R +i a+ +b c+ +b d+ +e f+
--R (36) [[| |,| |],| |,| |]
--R +a a+ +b c+ +c g+ +h i+
--R Type: List(List(Matrix(Fraction(Polynomial(Integer))))))
--E 36

--S 37 of 44
zero?(blockConcat(%) - E)
--R
--R
--R (37) true
--R Type: Boolean
--E 37

--S 38 of 44
blockSplit(E, [1,2,1], [2,2])
--R
--R
--R +a a+ +b c+
--R (38) [[[| |,| |],[b c]],| |,| |,[[c g],[h i]]]
--R +b d+ +e f+
--R Type: List(List(Matrix(Fraction(Polynomial(Integer))))))
--E 38

--S 39 of 44
zero?(blockConcat(%) - E)
--R
--R
--R (39) true
--R Type: Boolean
--E 39

--S 40 of 44
blockSplit(E, [2,1,1], 2)
--R
--R
--R +i a+ +b c+
--R (40) [[| |,| |],[b d],[e f],[c g],[h i]]
--R +a a+ +b c+
--R Type: List(List(Matrix(Fraction(Polynomial(Integer))))))
zero?(blockConcat(%) - E)  
  (41) true  
  Type: Boolean

blockSplit(E, 4, [2,2])  
  (42) [[[i a],[b c]],[[a a],[b c]],[[b d],[e f]],[[c g],[h i]]]  
  Type: List(List(Matrix(Fraction(Polynomial(Integer)))))

zero?(blockConcat(%) - E)  
  (43) true  
  Type: Boolean

MatrixManipulation(R: Field,Row: FiniteLinearAggregate(R),Col: FiniteLinearAggregate(R),M: MatrixCategory(R,Row,Col)) is a package constructor  
Abbreviation for MatrixManipulation is MAMA  
This constructor is exposed in this frame.  
Issue )edit bookvol10.4.pamphlet to see algebra source code for MAMA

------------------------------- Operations --------------------------------
  aColumn : (M,PositiveInteger) -> M  
  aRow : (M,PositiveInteger) -> M
  bandMatrix : (M,List(Integer)) -> M  
  blockConcat : List(List(M)) -> M
  diagonalMatrix : (M,Integer) -> M  
  diagonalMatrix : M -> M
  horizConcat : List(M) -> M  
  vertConcat : List(M) -> M
  bandMatrix : (M,Segment(Integer)) -> M
  blockSplit : (M,PositiveInteger,PositiveInteger) -> List(List(M))
  blockSplit : (M,PositiveInteger) -> List(List(M))
  blockSplit : (M,PositiveInteger,List(PositiveInteger)) -> List(List(M))
  blockSplit : (M,List(PositiveInteger)) -> List(List(M))
  columns : List(M) -> M
  columns : Segment(PositiveInteger) -> M
  element : (M,PositiveInteger,PositiveInteger) -> M
  horizSplit : (M,PositiveInteger) -> List(M)
  horizSplit : (M,List(PositiveInteger)) -> List(M)
  rows : List(M) -> M
Some functions for manipulating (dense) matrices.

Supported are various kinds of slicing, splitting
and stacking of matrices. The functions resemble
operations often used in numerical linear algebra
algorithms.

\[
M := \text{matrix}([\begin{array}{ccc}
  a & b & c \\
  d & e & f \\
  g & h & i 
\end{array}])
\]

\[
\text{element}(M, 2,2)
\]

\[
\text{aRow}(M, 1)
\]

\[
\text{aRow}(M, 2)
\]

\[
\text{aColumn}(M, 2)
\]
aColumn(M, 3)

rows(M, [1,2])

rows(M, [1,3])

rows(M, [3,2])

rows(M, 2..3)

columns(M, [1,2])

columns(M, [1,3])

\[
\begin{align*}
\text{columns}(M, [3,2]) & \quad +c \quad b+ \\
& \quad |f \quad e| \\
& \quad +i \quad h+ \\
\text{columns}(M, 1..2) & \quad +a \quad b+ \\
& \quad |d \quad e| \\
& \quad +g \quad h+ \\
\text{subMatrix}(M, [1,2],[1,2]) & \quad +a \quad b+ \\
& \quad |d \quad e| \\
& \quad +g \quad h+ \\
\text{subMatrix}(M, [1,3],[1,3]) & \quad +a \quad c+ \\
& \quad |g \quad i| \\
& \quad +g \quad i+ \\
\text{diagonalMatrix}(M) & \quad +a \quad 0 \quad 0+ \\
& \quad |0 \quad e \quad 0| \\
& \quad +0 \quad 0 \quad i+ \\
\text{diagonalMatrix}(M, 1) & \quad +0 \quad b \quad 0+ \\
& \quad |0 \quad 0 \quad f| \\
& \quad +0 \quad 0 \quad 0+ \\
\text{diagonalMatrix}(M, 2) & \quad +0 \quad 0 \quad c+
\end{align*}
\]
diagonalMatrix(M, -1)
  +0 0 0+
  |   |
  +d 0 0+
  +0 h 0+

diagonalMatrix(M, -2)
  +0 0 0+
  |   |
  +0 0 0+
  +g 0 0+

bandMatrix(M, [-1,1])
  +0 b 0+
  |   |
  +d 0 f+
  +0 h 0+

bandMatrix(M, [-1,0,1])
  +a b 0+
  |   |
  +d e f+
  +0 h i+

bandMatrix(M, -1..1)
  +a b 0+
  |   |
  +d e f+
  +0 h i+

Build a larger matrix

A := matrix([[a]])

[a]
B := matrix([b])
    [[b]]
C := matrix([c])
    [[c]]
A11 := element(M, 3,3)
    [[i]]
A12 := horizConcat([A,B,C])
    [a b c]
A21 := vertConcat([A,B,C])
    +a+
    |  |
    | b|
    |  |
    +c+
E := blockConcat([[A11,A12],[A21,M]])
    +i a b c+
    |  |  |
    | a a b c|
    |  |  |
    | b d e f|
    |  |  |
    +c g h i+
Split again
vertSplit(E, 2)
    +i a b c+ +b d e f+
    [ |  |  | ]
    +a a b c+ +c g h i+
horizSplit(E, 2)
    +i a+ +b c+
    |  |  |
    | a a| | b c|
    [ |  |  | ]
    | b d| | e f|
vertSplit(E, [1,2,1])

| +c g+ | +h i+ |
| a a b c+ |
| [i a b c],| |,[c g h i]]
| +b d e f+ |

horizSplit(E, [2,2])

| +i a+ | +b c+ |
| +a a b c+ |
| +b d e f+ |
| [i a b c],| |,[c g h i]]
| +b d e f+ |

blockSplit(E, 2,2)

| +i a+ | +b c+ | +b d+ | +e f+ |
| a a b c+ |
| c g+ | +h i+ |
| +a a b c+ | +c g+ | +h i+ |
| zero?(blockConcat(%) - E) |
| true |

blockSplit(E, [1,2,1], [2,2])

| +a a+ | +b c+ |
| +a a b c+ |
| +b d+ | +e f+ |
| [i a b c],| |,[c g h i]]
| [i a b c],| |,[c g h i]]
| +a a+ | +b c+ |

zero?(blockConcat(%) - E) 
true

blockSplit(E, [2,1,1], 2)

| +i a+ | +b c+ |
| [i a b c],| |,[c g h i]]
| [i a b c],| |,[c g h i]]
| +a a+ | +b c+ |

zero?(blockConcat(%) - E) 
true

blockSplit(E, 4, [2,2])

| +i a+ | +b c+ |
| [i a b c],| |,[c g h i]]
| [i a b c],| |,[c g h i]]
| +a a+ | +b c+ |

zero?(blockConcat(%) - E) 
true
[[[i a],[b c]],[[a d],[b c]],[[b d],[e f]],[[c g],[h i]]]

zero?(blockConcat(%) - E)

true

See Also:
- )show MatrixManipulation
- )help element
- )d op element
- )help aRow
- )d op aRow
- )help rows
- )d op rows
- )help aColumn
- )d op aColumn
- )help columns
- )d op columns
- )help subMatrix
- )d op subMatrix
- )help diagonalMatrix
- )d op diagonalMatrix
- )help bandMatrix
- )d op bandMatrix
- )help horizConcat
- )d op horizConcat
- )help vertConcat
- )d op vertConcat
- )help blockConcat
- )d op blockConcat
- )help vertSplit
- )d op vertSplit
- )help horizSplit
- )d op horizSplit
- )help blockSplit
- )d op blockSplit

element

---

--- element.help ---

====================================================================

---

element from MatrixManipulation (MAMA)

NAME

====
element

DOMAIN
========

MatrixManipulation (MAMA)

SYNOPSIS
========

\[\text{element} : \langle M, \text{PositiveInteger}, \text{PositiveInteger} \rangle \to M\]

where
\begin{align*}
R & : \text{Field} \\
\text{Row} & : \text{FiniteLinearAggregate} R \\
\text{Col} & : \text{FiniteLinearAggregate} R \\
M & : \text{MatrixCategory}(R, \text{Row}, \text{Col})
\end{align*}

DESCRIPTION
============

element returns a single element out of a matrix. The element is put into a one by one matrix.

ARGUMENTS
=========  

The first argument is an element of \( M \)

where
\begin{align*}
R & : \text{Field} \\
\text{Row} & : \text{FiniteLinearAggregate} R \\
\text{Col} & : \text{FiniteLinearAggregate} R \\
M & : \text{MatrixCategory}(R, \text{Row}, \text{Col})
\end{align*}

The second element is a member of PositiveInteger

The third element is a member of PositiveInteger

RETURN VALUE
=============  

The result is an element of \( M \)

where
\begin{align*}
R & : \text{Field} \\
\text{Row} & : \text{FiniteLinearAggregate} R \\
\text{Col} & : \text{FiniteLinearAggregate} R
\end{align*}
M : MatrixCategory(R, Row, Col)

EXAMPLES
========

M := matrix([[a,b,c],[d,e,f],[g,h,i]])

element(M,2,2)

NOTES
=====

REFERENCES
=========

SEE ALSO
========

o )show MatrixManipulation
o )show PositiveInteger
o )d op element

—— aRow.help ——

********************************************************************************
| aRow from MatrixManipulation (MAMA) |
********************************************************************************

NAME
====

aRow

DOMAIN
=====

MatrixManipulation (MAMA)

SYNOPSYS
========

aRow : (M, PositiveInteger) -> M

where
   R : Field
   Row : FiniteLinearAggregate R
DESCRIPTION

aRow returns a single row out of a matrix.
The row is put into a one by N matrix.

ARGUMENTS

The first argument is an element of M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

The second element is a member of PositiveInteger

RETURN VALUE

The result is an element of M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES

M := matrix([[a,b,c],[d,e,f],[g,h,i]])
aRow(M, 1)
aRow(M, 2)

NOTES

REFERENCES

SEE ALSO
--- rows.help ---

rows from MatrixManipulation (MAMA)

NAME

rows

DOMAIN

MatrixManipulation (MAMA)

SYNOPSYS

rows : (M, List PositiveInteger) -> M
rows : (M, Segment PositiveInteger) -> M

where
  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

DESCRIPTION

rows returns several rows out of a matrix.
The rows are stacked into a matrix.

ARGUMENTS

rows : (M, List PositiveInteger) -> M

  The first argument is an element of M
where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

The second element is a List of PositiveInteger

rows : (M, Segment PositiveInteger) -> M

The first argument is an element of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

The second element is a Segment of PositiveInteger

RETURN VALUE
============

The result is an element of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES
========

M := matrix([[a,b,c],[d,e,f],[g,h,i]])
rows(M, [1,2])
rows(M, [3,2])
rows(M, 2..3)

NOTES
=====

REFERENCES
==========

SEE ALSO
========
--- aColumn.help ---

====================================================================
aColumn from MatrixManipulation (MAMA)
====================================================================

NAME
====

aColumn

DOMAIN
======

MatrixManipulation (MAMA)

SYNOPSYS
========

aColumn : (M, PositiveInteger) -> M

where

  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

DESCRIPTION
============

aColumn returns a single column out of a matrix. The column is put into a one by N matrix.

ARGUMENTS
==========

The first argument is an element of M
where

\[
\begin{align*}
R &: \text{Field} \\
\text{Row} &: \text{FiniteLinearAggregate } R \\
\text{Col} &: \text{FiniteLinearAggregate } R \\
M &: \text{MatrixCategory}(R, \text{Row}, \text{Col})
\end{align*}
\]

The second element is a member of PositiveInteger

RETURN VALUE

=============

The result is an element of \( M \)

where

\[
\begin{align*}
R &: \text{Field} \\
\text{Row} &: \text{FiniteLinearAggregate } R \\
\text{Col} &: \text{FiniteLinearAggregate } R \\
M &: \text{MatrixCategory}(R, \text{Row}, \text{Col})
\end{align*}
\]

EXAMPLES

========

\[
M := \text{matrix}([[a,b,c],[d,e,f],[g,h,i]])
\]

\[
a\text{Column}(M, 2)
\]

NOTES

=====

REFERENCES

=========

SEE ALSO

=========---

- \( \text{show MatrixManipulation} \)
- \( \text{show PositiveInteger} \)
- \( \text{d op aColumn} \)

----------

— columns.help —

====================================================================
columns from MatrixManipulation (MAMA)
=====================================================================

NAME
columns

DOMAIN

MatrixManipulation (MAMA)

SYNOPSYS

columns : (M, List PositiveInteger) -> M
columns : (M, Segment PositiveInteger) -> M

where
   R : Field
   Row : FiniteLinearAggregate R
   Col : FiniteLinearAggregate R
   M : MatrixCategory(R, Row, Col)

DESCRIPTION

columns returns several columns out of a matrix.
The columns are stacked into a matrix.

ARGUMENTS

columns : (M, List PositiveInteger) -> M

The first argument is an element of M

where
   R : Field
   Row : FiniteLinearAggregate R
   Col : FiniteLinearAggregate R
   M : MatrixCategory(R, Row, Col)

The second element is a List of PositiveInteger

columns : (M, Segment PositiveInteger) -> M

The first argument is an element of M

where
   R : Field
   Row : FiniteLinearAggregate R
The second element is a Segment of PositiveInteger

RETURN VALUE

The result is an element of M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES

M := matrix([[a,b,c],[d,e,f],[g,h,i]])
columns(M, [1,2])
columns(M, [3,2])
columns(M, 2..3)

NOTES

REFERENCES

SEE ALSO

o )show MatrixManipulation
o )show PositiveInteger
o )show List
o )show Segment
o )d op columns

— subMatrix.help —

subMatrix from MatrixManipulation (MAMA)
NAME
=====

subMatrix

DOMAIN
======

MatrixManipulation (MAMA)

SYNOPSIS
========

\[
\text{subMatrix} : (M, \text{List PositiveInteger}, \text{List PositiveInteger}) \to M \\
\text{subMatrix} : (M, \text{Segment PositiveInteger}, \text{Segment PositiveInteger}) \to M
\]

where
- \( R \) : Field
- Row : FiniteLinearAggregate \( R \)
- Col : FiniteLinearAggregate \( R \)
- \( M \) : MatrixCategory\( (R, \text{Row}, \text{Col}) \)

DESCRIPTION
==========

\text{subMatrix} \text{ returns several elements out of a matrix.}
\text{The elements are stacked into a submatrix.}

ARGUMENTS
==========

\text{subMatrix} : (M, \text{List PositiveInteger}, \text{List PositiveInteger}) \to M

The first argument is an element of \( M \)
where
- \( R \) : Field
- Row : FiniteLinearAggregate \( R \)
- Col : FiniteLinearAggregate \( R \)
- \( M \) : MatrixCategory\( (R, \text{Row}, \text{Col}) \)

The second element is a List of PositiveInteger
The third element is a List of PositiveInteger

\text{subMatrix} : (M, \text{Segment PositiveInteger}, \text{Segment PositiveInteger}) \to M

The first argument is an element of \( M \)
where
\begin{verbatim}
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

The second element is a Segment of PositiveInteger

The third element is a Segment of PositiveInteger

RETURN VALUE

The result is an element of M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES

M := matrix([[a,b,c],[d,e,f],[g,h,i]])
subMatrix(M, [1,2],[1,2])
subMatrix(M, [1,3],[1,3])

M := matrix([[a,b,c],[d,e,f],[g,h,i]])
subMatrix(M, 1..2,2..3)

NOTES

REFERENCES

SEE ALSO

o )show MatrixManipulation
o )show PositiveInteger
o )show List
o )show Segment
o )d op subMatrix
\end{verbatim}
diagonalMatrix : (M, Integer) -> M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

---

diagonalMatrix returns a diagonal out of a matrix.
The diagonal is put into a matrix of same shape as the original one. Positive integer arguments select upper off-diagonals, negative ones lower off-diagonals.

diagonalMatrix : M -> M

diagonalMatrix returns the main diagonal out of a matrix. The diagonal is put into a matrix of same shape as the original one.

ARGUMENTS

The first argument is an element of M

where
\[ R : \text{Field} \\
\text{Row} : \text{FiniteLinearAggregate} \ R \\
\text{Col} : \text{FiniteLinearAggregate} \ R \\
M : \text{MatrixCategory}(R, \text{Row}, \text{Col}) \]

The second element is an Integer

\[
\text{diagonalMatrix} : M \rightarrow M
\]

The first argument is an element of \( M \)

where
\[
\begin{align*}
R & : \text{Field} \\
\text{Row} & : \text{FiniteLinearAggregate} \ R \\
\text{Col} & : \text{FiniteLinearAggregate} \ R \\
M & : \text{MatrixCategory}(R, \text{Row}, \text{Col})
\end{align*}
\]

RETURN VALUE
==========

The result is an element of \( M \)

where
\[
\begin{align*}
R & : \text{Field} \\
\text{Row} & : \text{FiniteLinearAggregate} \ R \\
\text{Col} & : \text{FiniteLinearAggregate} \ R \\
M & : \text{MatrixCategory}(R, \text{Row}, \text{Col})
\end{align*}
\]

EXAMPLES
========

\[
M := \text{matrix}([[a, b, c], [d, e, f], [g, h, i]])
\]

\[
\text{diagonalMatrix}(M, 1)
\]

\[
\text{diagonalMatrix}(M, 2)
\]

\[
\text{diagonalMatrix}(M, -1)
\]

\[
\text{diagonalMatrix}(M)
\]

NOTES
=====

REFERENCES
==========

SEE ALSO
========
bandMatrix : (M, List Integer) -> M
bandMatrix : (M, Segment Integer) -> M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

DESCRIPTION
========

bandMatrix : (M, List Integer) -> M

bandMatrix returns multiple diagonals out of a matrix. The diagonals are put into a matrix of same shape as the original one. Positive integer arguments select upper off-diagonals, negative ones lower off-diagonals.

bandMatrix : (M, Segment Integer) -> M

bandMatrix returns multiple diagonals out of a matrix. The diagonals are put into a matrix of same shape as the
original one. Positive integer arguments select upper
off-diagonals, negative ones lower off-diagonals.

ARGUMENTS
==========

\textbf{bandMatrix} : (M, List Integer) -> M

The first argument is an element of M
where
\begin{align*}
R &: \text{Field} \\
Row &: \text{FiniteLinearAggregate } R \\
Col &: \text{FiniteLinearAggregate } R \\
M &: \text{MatrixCategory}(R, Row, Col)
\end{align*}

The second element is a List of Integer

\textbf{bandMatrix} : (M, Segment Integer) -> M

The first argument is an element of M
where
\begin{align*}
R &: \text{Field} \\
Row &: \text{FiniteLinearAggregate } R \\
Col &: \text{FiniteLinearAggregate } R \\
M &: \text{MatrixCategory}(R, Row, Col)
\end{align*}

The second element is a Segment of Integer

RETURN VALUE
==========

The result is an element of M
where
\begin{align*}
R &: \text{Field} \\
Row &: \text{FiniteLinearAggregate } R \\
Col &: \text{FiniteLinearAggregate } R \\
M &: \text{MatrixCategory}(R, Row, Col)
\end{align*}

EXAMPLES
========

\begin{verbatim}
M := matrix([[a,b,c],[d,e,f],[g,h,i]])
bandMatrix(M, [-1,1])
bandMatrix(M, [-1,0,1])
bandMatrix(M, -1..1)
\end{verbatim}
NAME
====

horizConcat

DOMAIN
=====

MatrixManipulation (MAMA)

SYNOPSYS
========

horizConcat : (List M) -> M

where
   R : Field
   Row : FiniteLinearAggregate R
   Col : FiniteLinearAggregate R
   M : MatrixCategory(R, Row, Col)

DESCRIPTION
============

horizConcat concatenates matrices column wise.
ARGUMENTS
==========

The first argument is a List of elements of M

where
   R : Field
   Row : FiniteLinearAggregate R
   Col : FiniteLinearAggregate R
   M : MatrixCategory(R, Row, Col)

RETURN VALUE
=============

The result is an element of M

where
   R : Field
   Row : FiniteLinearAggregate R
   Col : FiniteLinearAggregate R
   M : MatrixCategory(R, Row, Col)

EXAMPLES
========

A := matrix([[a]])
B := matrix([[b]])
C := matrix([[c]])
A12 := horizConcat([A,B,C])

NOTES
=====

REFERENCES
==========

SEE ALSO
========

  o )show MatrixManipulation
  o )show List
  o )d op horizConcat

|— vertConcat.help —|
vertConcat from MatrixManipulation (MAMA)

NAME

vertConcat

DOMAIN

MatrixManipulation (MAMA)

SYNOPSIS

vertConcat : (List M) -> M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

DESCRIPTION

vertConcat concatenates matrices row wise.

ARGUMENTS

The first argument is a list of elements of M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

RETURN VALUE

The result is an element of M

where

R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES
========

A := matrix([[a]])
B := matrix([[b]])
C := matrix([[c]])
A21 := vertConcat([A,B,C])

NOTES
=====

REFERENCES
==========

SEE ALSO
========

o )show MatrixManipulation
o )show List
o )d op vertConcat

---

blockConcat.help ---

=================================================================================
blockConcat from MatrixManipulation (MAMA)
=================================================================================

NAME
====

blockConcat

DOMAIN
======

MatrixManipulation (MAMA)

SYNOPSYS
========

blockConcat : (List List M) -> M

where
    R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

DESCRIPTION
============

txt: blockConcat concatenates matrices row and column wise, building a block matrix. The order is row major as in matrix.

ARGUMENTS
==========

The first argument is a list of lists of elements of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

RETURN VALUE
=============

The result is an element of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES
========

A := matrix([[a]])
B := matrix([[b]])
C := matrix([[c]])
A11 := element(M, 3,3)
A12 := horizConcat([A,B,C])
A21 := vertConcat([A,B,C])
M := matrix([[a,b,c],[d,e,f],[g,h,i]])
E := blockConcat([[A11,A12],[A21,M]])
t1 := blockSplit(E, 4, [2,2])
t2 := blockConcat t1
zero?(E-t2)
\begin{verbatim}
t3 := blockSplit(E, [1,2,1], [2,2])
t4 := blockConcat t3
zero?(E-t4)
\end{verbatim}

NOTES
=====

REFERENCES
==========

SEE ALSO
========

\texttt{\texttt{o )show MatrixManipulation}}
\texttt{\texttt{o )show List}}
\texttt{\texttt{o }d op blockConcat}

---

\texttt{— vertSplit.help —}

=====================================================================
vertSplit from MatrixManipulation (MAMA)
=====================================================================

NAME
=====

vertSplit

DOMAIN
======

MatrixManipulation (MAMA)

SYNOPSIS
=======

\texttt{vertSplit : (M, PositiveInteger) -> List M}
\texttt{vertSplit : (M, List PositiveInteger) -> List M}

where
\texttt{R : Field}
\texttt{Row : FiniteLinearAggregate R}
\texttt{Col : FiniteLinearAggregate R}
\texttt{M : MatrixCategory(R, Row, Col)}

DESCRIPTION
vertSplit splits a matrix into multiple submatrices row wise.

ARGUMENTS
==========

vertSplit : (M, PositiveInteger) \rightarrow List M

The first argument is an element of M

where

- R : Field
- Row : FiniteLinearAggregate R
- Col : FiniteLinearAggregate R
- M : MatrixCategory(R, Row, Col)

The second argument is a PositiveInteger

vertSplit : (M, List PositiveInteger) \rightarrow List M

The first argument is an element of M

where

- R : Field
- Row : FiniteLinearAggregate R
- Col : FiniteLinearAggregate R
- M : MatrixCategory(R, Row, Col)

The second argument is a List of PositiveInteger

RETURN VALUE
============

The result is an List of elements of M

where

- R : Field
- Row : FiniteLinearAggregate R
- Col : FiniteLinearAggregate R
- M : MatrixCategory(R, Row, Col)

EXAMPLES
========

E := matrix([[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]])
t1:= vertSplit(E, 2)
t2:= vertSplit(E, [1,2,1])
NOTES
=====

REFERENCES
=========

SEE ALSO
========

- )show MatrixManipulation
- )show List
- )show PositiveInteger
- )d op vertSplit

———

— horizSplit.help ——

====================================================================
horizSplit from MatrixManipulation (MAMA)
====================================================================

NAME
===

horizSplit

DOMAIN
=====

MatrixManipulation (MAMA)

SYNOPSYS
========

horizSplit : (M, PositiveInteger) -> List M
horizSplit : (M, List PositiveInteger) -> List M

where
  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

DESCRIPTION
============
horizSplit splits a matrix into multiple submatrices column wise.

ARGUMENTS
==========

horizSplit : (M, PositiveInteger) -> List M

The first argument is an element of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

The second argument is a PositiveInteger

horizSplit : (M, List PositiveInteger) -> List M

The first argument is an element of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

The second argument is a List of PositiveInteger

RETURN VALUE
============

The result is an List of elements of M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

EXAMPLES
========

E := matrix([[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]])
t1:= horizSplit(E, 2)
t2:= horizSplit(E, [2,2])
t3:= horizSplit(E, [1,2,1])
NOTES =====

REFERENCES ========

SEE ALSO ========

- show MatrixManipulation
- show List
- show PositiveInteger
- d op horizSplit

---

blockSplit.help ---

====================================================================
blockSplit from MatrixManipulation (MAMA)
====================================================================

NAME ====

blockSplit

DOMAIN ======

MatrixManipulation (MAMA)

SYNOPSIS ======

blockSplit : (M, PositiveInteger, PositiveInteger) -> List List M
blockSplit : (M, List PositiveInteger, PositiveInteger) -> List List M
blockSplit : (M, PositiveInteger, List PositiveInteger) -> List List M
blockSplit : (M, List PositiveInteger, List PositiveInteger) -> List List M

where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

DESCRIPTION
blockSplit splits a matrix into multiple submatrices row and column wise, dividing a matrix into blocks.

ARGUMENTS

blockSplit : (M, PositiveInteger, PositiveInteger) -> List List M

The first argument is an element of M

where

  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

The second argument is a PositiveInteger

The third argument is a PositiveInteger

blockSplit : (M, List PositiveInteger, PositiveInteger) -> List List M

The first argument is an element of M

where

  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

The second argument is a List of PositiveInteger

The third argument is a PositiveInteger

blockSplit : (M, PositiveInteger, List PositiveInteger) -> List List M

The first argument is an element of M

where

  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

The second argument is a PositiveInteger

The third argument is a List of PositiveInteger
blockSplit : (M, List PositiveInteger, List PositiveInteger) -> List List M

The first argument is an element of M

where
  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

The second argument is a List of PositiveInteger

The third argument is a List of PositiveInteger

RETURN VALUE
==============

The result is an List of Lists of elements of M

where
  R : Field
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R, Row, Col)

EXAMPLES
========

E := matrix([[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]])

t1:= blockSplit(E,2,2)

t2:= blockSplit(E, [2,1,1], 2)

t3:= blockSplit(E, 4, [2,2])

t4:= blockSplit(E, [1,2,1], [2,2])

NOTES
=====

REFERENCES
==========

SEE ALSO
========

 o )show MatrixManipulation
 o )show List
MatrixManipulation (MAMA)

Exports:

<table>
<thead>
<tr>
<th>aColumn</th>
<th>aRow</th>
<th>bandMatrix</th>
<th>blockConcat</th>
<th>blockSplit</th>
</tr>
</thead>
<tbody>
<tr>
<td>columns</td>
<td>diagonalMatrix</td>
<td>element</td>
<td>horizConcat</td>
<td>horizSplit</td>
</tr>
<tr>
<td>rows</td>
<td>subMatrix</td>
<td>vertConcat</td>
<td>vertSplit</td>
<td></td>
</tr>
</tbody>
</table>

—— package MAMA MatrixManipulation ——

)abbrev package MAMA MatrixManipulation
++ Author: Raoul Bourquin
++ Date Created: 17 November 2012
++ Date Last Updated: 1 December 2012
++ Description:
++ Some functions for manipulating (dense) matrices.
++ Supported are various kinds of slicing, splitting and stacking of
++ matrices. The functions resemble operations often used in numerical
++ linear algebra algorithms.
MatrixManipulation(R, Row, Col, M) : Exports == Implementation where
R : Field
Row : FiniteLinearAggregate R
Col : FiniteLinearAggregate R
M : MatrixCategory(R, Row, Col)

I ==> Integer
PI ==> PositiveInteger
LI ==> List I
SI ==> Segment I
LPI ==> List PI
SPI ==> Segment PI
Exports => with

-- Slicing matrices

-- How to call aRow, aColumn? Name clashed with usual row, column
-- Package call is ugly because of many parameters of MAMA

element : (M, PI, PI) -> M
  \spad{element} returns a single element out of a matrix.
  \spad{element} The element is put into a one by one matrix.
  ++
  ++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
  ++X element(M,2,2)

aRow : (M, PI) -> M
  \spad{aRow} returns a single row out of a matrix.
  \spad{aRow} The row is put into a one by N matrix.
  ++
  ++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
  ++X aRow(M, 1)
  ++X aRow(M, 2)

rows : (M, LPI) -> M
  \spad{rows} returns several rows out of a matrix.
  \spad{rows} The rows are stacked into a matrix.
  ++
  ++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
  ++X rows(M, [1,2])
  ++X rows(M, [3,2])

rows : (M, SPI) -> M
  \spad{rows} returns several rows out of a matrix.
  \spad{rows} The rows are stacked into a matrix.
  ++
  ++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
  ++X rows(M, 2..3)

aColumn : (M, PI) -> M
  \spad{aColumn} returns a single column out of a matrix.
  \spad{aColumn} The column is put into a one by N matrix.
  ++
  ++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
  ++X aColumn(M, 2)

columns : (M, LPI) -> M
  \spad{columns} returns several columns out of a matrix.
  \spad{columns} The columns are stacked into a matrix.
  ++
  ++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
++X columns(M, [1,2])
++X columns(M, [3,2])

columns : (M, SPI) -> M
++ \spad{columns} returns several columns out of a matrix.
++ The columns are stacked into a matrix.
++
++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
++X columns(M, 1..2)

subMatrix : (M, LPI, LPI) -> M
++ \spad{subMatrix} returns several elements out of a matrix.
++ The elements are stacked into a submatrix.
++
++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
++X subMatrix(M, [1,2],[1,2])
++X subMatrix(M, [1,3],[1,3])

subMatrix : (M, SPI, SPI) -> M
++ \spad{subMatrix} returns several elements out of a matrix.
++ The elements are stacked into a submatrix.
++
++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
++X subMatrix(M, 1..2,2..3)

diagonalMatrix : (M, I) -> M
++ \spad{diagonalMatrix} returns a diagonal out of a matrix.
++ The diagonal is put into a matrix of same shape as the
++ original one. Positive integer arguments select upper
++ off-diagonals, negative ones lower off-diagonals.
++
++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
++X diagonalMatrix(M, 1)
++X diagonalMatrix(M, 2)
++X diagonalMatrix(M, -1)

diagonalMatrix : M -> M
++ \spad{diagonalMatrix} returns the main diagonal out of
++ a matrix. The diagonal is put into a matrix of same shape
++ as the original one.
++
++X M := matrix([[a,b,c],[d,e,f],[g,h,i]])
++X diagonalMatrix(M)

bandMatrix : (M, LI) -> M
++ \spad{bandMatrix} returns multiple diagonals out of a matrix.
++ The diagonals are put into a matrix of same shape as the
++ original one. Positive integer arguments select upper
++ off-diagonals, negative ones lower off-diagonals.
++
++ \spad{M} := matrix([[a,b,c],[d,e,f],[g,h,i]])
++ \spad{bandMatrix}(M, [-1,1])
++ \spad{bandMatrix}(M, [-1,0,1])

\textbf{bandMatrix : (M, SI) -> M}
++ \spad{bandMatrix} returns multiple diagonals out of a matrix.
++ The diagonals are put into a matrix of same shape as the
++ original one. Positive integer arguments select upper
++ off-diagonals, negative ones lower off-diagonals.
++
++ \spad{M} := matrix([[a,b,c],[d,e,f],[g,h,i]])
++ \spad{bandMatrix}(M, -1..1)

\textbf{-- Stacking matrices}

\textbf{horizConcat : (List M) -> M}
++ \spad{horizConcat} concatenates matrices column wise.
++
++ \spad{A} := matrix([[a]])
++ \spad{B} := matrix([[b]])
++ \spad{C} := matrix([[c]])
++ \spad{A12} := horizConcat([A,B,C])

\textbf{vertConcat : (List M) -> M}
++ \spad{vertConcat} concatenates matrices row wise.
++
++ \spad{A} := matrix([[a]])
++ \spad{B} := matrix([[b]])
++ \spad{C} := matrix([[c]])
++ \spad{A21} := vertConcat([A,B,C])

\textbf{blockConcat : (List List M) -> M}
++ \spad{blockConcat} concatenates matrices row and
++ column wise, building a block matrix. The order
++ is row major as in \spad{matrix}.
++
++ \spad{A} := matrix([[a]])
++ \spad{B} := matrix([[b]])
++ \spad{C} := matrix([[c]])
++ \spad{A11} := element(M, 3,3)
++ \spad{A12} := horizConcat([A,B,C])
++ \spad{A21} := vertConcat([A,B,C])
++ \spad{M} := matrix([[a,b,c],[d,e,f],[g,h,i]])
++ \spad{E} := blockConcat([A11,A12],[A21,M])
++ \spad{t1} := blockSplit(E, 4, [2,2])
++ \spad{t2} := blockConcat t1
++ \spad{zero?}(E-t2)
++ \spad{t3} := blockSplit(E, [1,2,1], [2,2])
++ \spad{t4} := blockConcat t3
++ \spad{zero?}(E-t4)
-- Splitting matrices

vertSplit : (M, PI) -> List M
++ \spad{vertSplit} splits a matrix into multiple
++ submatrices row wise.
++
++X E := matrix([i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i])
++X t1:= vertSplit(E, 2)

vertSplit : (M, LPI) -> List M
++ \spad{vertSplit} splits a matrix into multiple
++ submatrices row wise.
++
++X E := matrix([[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]])
++X t1:= vertSplit(E, [1,2,1])

horizSplit : (M, PI) -> List M
++ \spad{horizSplit} splits a matrix into multiple
++ submatrices column wise.
++
++X E := matrix([i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i])
++X t1:= horizSplit(E, 2)

horizSplit : (M, LPI) -> List M
++ \spad{horizSplit} splits a matrix into multiple
++ submatrices column wise.
++
++X E := matrix([[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]])
++X t1:= horizSplit(E, [2,2])
++X t2:= horizSplit(E, [1,2,1])

blockSplit : (M, PI, PI) -> List List M
++ \spad{blockSplit} splits a matrix into multiple
++ submatrices row and column wise, dividing
++ a matrix into blocks.
++
++X E := matrix([i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i])
++X t1:= blockSplit(E,2,2)

blockSplit : (M, LPI, PI) -> List List M
++ \spad{blockSplit} splits a matrix into multiple
++ submatrices row and column wise, dividing
++ a matrix into blocks.
++
++X E := matrix([i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i])
++X t1:= blockSplit(E, [2,1,1], 2)

blockSplit : (M, PI, LPI) -> List List M
++ \spad{blockSplit} splits a matrix into multiple
++ submatrices row and column wise, dividing
++ a matrix into blocks.
++
++X E := matrix([[[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]]])
++X t1:= blockSplit(E, 4, [2,2])

blockSplit : (M, LPI, LPI) -> List List M
++ \spad{blockSplit} splits a matrix into multiple
++ submatrices row and column wise, dividing
++ a matrix into blocks.
++
++X E := matrix([[[i,a,b,c],[a,a,b,c],[b,d,e,f],[c,g,h,i]]])
++X t1:= blockSplit(E, [1,2,1], [2,2])

Implementation => add

minr => minRowIndex
maxr => maxRowIndex
minc => minColIndex
maxc => maxColIndex

-- Custom function to expand Segment(PositiveInteger) into
-- List(PositiveInteger). This operation is not supported by the
-- overly restrictive library implementation.
expand(spi : SPI) : LPI ==
  lr := empty()$LPI
  l : PI := lo spi
  h : PI := hi spi
  inc : I := incr spi
  zero? inc => error "Cannot expand a segment with an increment of zero"
  if inc > 0 then
    while l <= h repeat
      lr := concat(l, lr)
      l := (l + inc) pretend PI
  else
    while l >= h repeat
      lr := concat(l, lr)
      l := (l + inc) pretend PI
  reverse! lr

element(A, r, c) ==
  matrix([[A(r,c)]])
aRow(A:M, r:PI) : M ==
  subMatrix(A, r, r, minc A, maxc A)
rows(A:M, lst:LPI) : M ==
  ls := [aRow(A, r) for r in lst]
  reduce(vertConcat, ls)
rows(A:M, si:SPI) : M ==
    rows(A, expand(si))

aColumn(A:M, c:PI) : M ==
    subMatrix(A, minr A, maxr A, c, c)

columns(A:M, lst:LPI) : M ==
    ls := [aColumn(A,c) for c in lst]
    reduce(horizConcat, ls)

columns(A:M, si:SPI) : M ==
    columns(A, expand(si))

diagonalMatrix(A, n) ==
    nr := nrows(A)
    nc := ncols(A)
    n > (nc-1) => error "requested diagonal out of range"
    n < 0 and abs(n) > (nr-1) => error "requested diagonal out of range"
    B := zero(nr,nc)
    if n >= 0 then
        dl := min(nc-n, nr)
        sr := minr(A)
        sc := minc(A) + n
    else
        dl := min(nc, nr-abs(n))
        sr := minr(A) + abs(n)
        sc := minc(A)
    for i in 0..(dl-1) repeat
        qsetelt!(B, sr+i, sc+i, A(sr+i, sc+i))
    B

diagonalMatrix(A) ==
    diagonalMatrix(A, 0)

bandMatrix(A:M, ln:LI) : M ==
    -- Really inefficient
    reduce(\+, [diagonalMatrix(A,d) for d in ln])

bandMatrix(A:M, si:SI) : M ==
    bandMatrix(A, expand(si))

subMatrix(A:M, lr:LPI, lc:LPI) : M ==
    -- Really inefficient
    lle := [[ element(A,r,c) for c in lc] for r in lr]
    blockConcat(lle)

subMatrix(A:M, sr:SPI, sc:SPI) : M ==
    subMatrix(A, low sr, high sr, low sc, high sc)

    -- Stack matrices
horizConcat(LA) ==
  reduce(horizConcat, LA)

vertConcat(LA) ==
  reduce(vertConcat, LA)

blockConcat(LLA: List List M) : M ==
  reduce(vertConcat, [reduce(horizConcat, LA) for LA in LLA])

-- Split matrices

vertSplit(A:M, r:PI) : List M ==
  dr := nrows(A) exquo r
  dr case "failed" => error "split does not result in an equal division"
  mir := minr A
  mic := minc A
  mac := maxc A
  [ subMatrix(A, mir+i*dr, mir+(i+1)*dr-1, mic, mac) for i in 0..(r-1) ]

vertSplit(A:M, lr:LPI) : List M ==
  reduce("+", lr) ~= nrows(A) => _
  error "split does not result in proper partition"
  l : List PI := cons(1, scan(_+, lr, 1$PI)$ListFunctions2(PI,PI))
  mir := minr(A) -1 -- additional shift because l starts at 1
  mic := minc A
  mac := maxc A
  result := _
  [ subMatrix(A, mir+l(i-1), mir+l(i)-1, mic, mac) for i in 2..#l ]

horizSplit(A:M, c:PI) : List M ==
  dc := ncols(A) exquo c
  dc case "failed" => error "split does not result in an equal division"
  mir := minr A
  mar := maxr A
  mic := minc A
  [ subMatrix(A, mir, mar, mic+i*dc, mic+(i+1)*dc-1) for i in 0..(c-1) ]

horizSplit(A:M, lc:LPI) : List M ==
  reduce("+", lc) ~= ncols(A) => _
  error "split does not result in proper partition"
  l : List PI := cons(1, scan(_+, lc, 1$PI)$ListFunctions2(PI,PI))
  mir := minr A
  mar := maxr A
  mic := minc(A) -1 -- additional shift because l starts at 1
  result := _
  [ subMatrix(A, mir, mar, mic+l(i-1), mic+l(i)-1) for i in 2..#l ]

blockSplit(A:M, nr:PI, nc:PI) : List List M ==
  [ horizSplit(X, nc) for X in vertSplit(A, nr) ]
blockSplit(A:M, lr:LPI, nc:PI) : List List M ==
[ horizSplit(X, nc) for X in vertSplit(A, lr) ]

blockSplit(A:M, nr:PI, lc:LPI) : List List M ==
[ horizSplit(X, lc) for X in vertSplit(A, nr) ]

blockSplit(A:M, lr:LPI, lc:LPI) : List List M ==
[ horizSplit(X, lc) for X in vertSplit(A, lr) ]

---

package MTHING MergeThing

-- MergeThing.input --

)set break resume
)sys rm -f MergeThing.output
)spool MergeThing.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MergeThing
--R
--R MergeThing(S: OrderedSet) is a package constructor
--R Abbreviation for MergeThing is MTHING
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MTHING
--R
--R------------------------------- Operations --------------------------------
--R mergeDifference : (List(S),List(S)) -> List(S)
--E 1
MergeThing (MTHING)

Exports:
mergeDifference

— package MTHING MergeThing —

)abbrev package MTHING MergeThing
++ Description:
++ This package exports tools for merging lists

MergeThing(S:OrderedSet): Exports == Implementation where
Exports == with
  mergeDifference: (List(S),List(S)) -> List(S)
++ mergeDifference(l1,l2) returns a list of elements in l1 not present ++ in l2. Assumes lists are ordered and all x in l2 are also in l1.

Implementation == add
mergeDifference1: (List S,S,List S) -> List S
mergeDifference(x,y) ==
  null x or null y => x
  mergeDifference1(x,y.first,y.rest)
  x.first=y.first => x.rest
  x
mergeDifference1(x,fy,ry) ==
  rx := x
  while not null rx repeat
    rx := rx.rest
    frx := rx.first
    while fy < frx repeat
      null ry => return x
      fy := first ry
      ry := rest ry
    frx = fy =>
      x.rest := rx.rest
      null ry => return x
      fy := ry.first
      ry := ry.rest
    x := rx

— MTHING.dotabb —

"MTHING" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MTHING"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"MTHING" -> "FLAGG"

— MESH MeshCreationRoutinesForThreeDimensions —

package MESH MeshCreationRoutinesForThreeDimensions

— MeshCreationRoutinesForThreeDimensions.input —

)set break resume
)sys rm -f MeshCreationRoutinesForThreeDimensions.output
)spool MeshCreationRoutinesForThreeDimensions.output
)set message test on
)set message auto off
MeshCreationRoutinesForThreeDimensions is a package constructor
Abbreviation for MeshCreationRoutinesForThreeDimensions is MESH
This constructor is not exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for MESH

---

<table>
<thead>
<tr>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>meshFun2Var : (((DoubleFloat,DoubleFloat) -&gt; DoubleFloat),Union(((DoubleFloat,DoubleFloat,DoubleFloat) -&gt; DoubleFloat),undefined),Segment(DoubleFloat),Segment(DoubleFloat),List(DrawOption)) -&gt; ThreeSpace(DoubleFloat)</td>
</tr>
<tr>
<td>meshPar1Var : (Expression(Integer),Expression(Integer),Expression(Integer),(DoubleFloat -&gt; DoubleFloat),Segment(DoubleFloat),List(DrawOption)) -&gt; ThreeSpace(DoubleFloat)</td>
</tr>
<tr>
<td>meshPar2Var : (((DoubleFloat,DoubleFloat) -&gt; DoubleFloat),((DoubleFloat,DoubleFloat) -&gt; ...,DoubleFloat)) -&gt; DoubleFloat</td>
</tr>
<tr>
<td>meshPar2Var : (((DoubleFloat,DoubleFloat) -&gt; Point(DoubleFloat)),Segment(DoubleFloat),Segment(DoubleFloat),List(DrawOption)) -&gt; ThreeSpace(DoubleFloat)</td>
</tr>
<tr>
<td>meshPar2Var : (ThreeSpace(DoubleFloat),((DoubleFloat,DoubleFloat) -&gt; Point(DoubleFloat)),Segment(DoubleFloat),Segment(DoubleFloat),List(DrawOption)) -&gt; ThreeSpace(DoubleFloat)</td>
</tr>
<tr>
<td>ptFunc : (((DoubleFloat,DoubleFloat) -&gt; DoubleFloat),((DoubleFloat,DoubleFloat) -&gt; ...,DoubleFloat)) -&gt; ((DoubleFloat,DoubleFloat) -&gt; Point(DoubleFloat))</td>
</tr>
</tbody>
</table>

---

MeshCreationRoutinesForThreeDimensions is a package constructor
Abbreviation for MeshCreationRoutinesForThreeDimensions is MESH
This constructor is not exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for MESH

This package has no description

See Also:
• )show MeshCreationRoutinesForThreeDimensions
MeshCreationRoutinesForThreeDimensions (MESH)

Exports:

meshFun2Var meshPar1Var meshPar2Var ptFunc

--- package MESH MeshCreationRoutinesForThreeDimensions ---

)abbrev package MESH MeshCreationRoutinesForThreeDimensions
++ Author: Jim Wen
++ Date Last Updated: October 1991 by Jon Steinbach
++ Description:
++ This package has no description

MeshCreationRoutinesForThreeDimensions():Exports == Implementation where

I ==> Integer
PI ==> PositiveInteger
SF ==> DoubleFloat
L ==> List
SEG ==> Segment
S ==> String
Fn1 ==> SF -> SF
Fn2 ==> (SF,SF) -> SF
Fn3 ==> (SF,SF,SF) -> SF
FnPt ==> (SF,SF) -> Point(SF)
FnU ==> Union(Fn3, "undefined")
EX ==> Expression
DROP ==> DrawOption
POINT ==> Point(SF)
SPACE3 ==> ThreeSpace(SF)
COMPPROP ==> SubSpaceComponentProperty
TUBE ==> TubePlot

Exports == with

meshPar2Var: (Fn2,Fn2,FnU,SEG SF,SEG SF,L DROP) -> SPACE3
++ meshPar2Var(f,g,h,j,s1,s2,l) undocumented
meshPar2Var: (FnPt,SEG SF,SEG SF,L DROP) -> SPACE3
++ meshPar2Var(f,s1,s2,l) undocumented
meshPar2Var: (SPACE3,FnPt,SEG SF,SEG SF,L DROP) -> SPACE3
++ meshPar2Var(sp,f,s1,s2,l) undocumented
meshFun2Var: (Fn2,FnU,SEG SF,SEG SF,L DROP) -> SPACE3
++ meshFun2Var(f,g,s1,s2,l) undocumented
meshPar1Var: (EX I,EX I,EX I,Fn1,SEG SF,L DROP) -> SPACE3
++ meshPar1Var(s,t,u,f,s1,l) undocumented
ptFunc: (Fn2,Fn2,Fn2,Fn3) -> ((SF,SF) -> POINT)
++ ptFunc(a,b,c,d) is an internal function exported in
++ order to compile packages.

Implementation ==> add
import ViewDefaultsPackage()
import SubSpaceComponentProperty()
import DrawOptionFunctions0
import SPACE3
--import TUBE()

-- local functions
numberCheck(nums:Point SF):Void ==
  -- this function checks to see that the small floats are
  -- actually just that - rather than complex numbers or
  -- whatever (the whatever includes nothing presently
  -- since NaN, Not a Number, is not necessarily supported
  -- by common lisp). note that this function is dependent
  -- upon the fact that Common Lisp supports complex numbers.
  for i in minIndex(nums)..maxIndex(nums) repeat
    COMPLEXP(nums.(i::PositiveInteger))$Lisp =>
      error "An unexpected complex number was encountered in the calculations."
makePt:(SF,SF,SF,SF) -> POINT
makePt(x,y,z,c) == point(l : List SF := [x,y,z,c])
ptFunc(f,g,h,c) ==
  (z1:SF,z2:SF):POINT ->
    x := f(z1,z2); y := g(z1,z2); z := h(z1,z2)
    makePt(x,y,z,c(x,y,z))

-- parameterized equations of two variables
meshPar2Var(sp,ptFun,uSeg,vSeg,opts) ==
  -- the issue of open and closed needs to be addressed, here, we are
  -- defaulting to open (which is probably the correct default)
  -- the user should be able to override that (optional argument?)
  llp : L L POINT := nil()
uNum : PI := var1Steps(opts,var1StepsDefault())
vNum : PI := var2Steps(opts,var2StepsDefault())
  ustep := (lo uSeg - hi uSeg)/uNum
  vstep := (lo vSeg - hi vSeg)/vNum
  someV := hi vSeg
  for iv in vNum..0 by -1 repeat
    if zero? iv then someV := lo vSeg
-- hack: get last number in segment within segment
lp : L POINT := nil()

someU := hi uSeg
for iu in uNum..0 by -1 repeat
  if zero? iu then someU := lo uSeg
-- hack: get last number in segment within segment
pt := ptFun(someU,someV)
numberCheck pt
lp := concat(pt,lp)
someU := someU + ustep
llp := concat(lp,llp)
someV := someV + vstep

-- now lp contains a list of lists of points
-- for a surface that is a result of a function of 2 variables,
-- the main component is open and each sublist is open as well
lProp : L COMPPROP := [ new() for l in llp ]
for aProp in lProp repeat
  close(aProp,false)
solid(aProp,false)
aProp : COMPPROP:= new()
  close(aProp,false)
solid(aProp,false)

-- space := create3Space()

mesh(space,llp,lProp,aProp)
space

-- explicit equations of two variables

meshFun2Var((z1:SF,z2:SF):SF +-> z1,
(x1:SF,x2:SF):SF +-> x2,zFun,colorFun,xSeg,ySeg,opts)
package MDDFACT ModularDistinctDegreeFactorizer

-- ModularDistinctDegreeFactorizer.input --

)set break resume  
)sys rm -f ModularDistinctDegreeFactorizer.output  
)spool ModularDistinctDegreeFactorizer.output  
)set message test on  
)set message auto off  
clear all  
--S 1 of 1  
)show ModularDistinctDegreeFactorizer  
--R  
--R ModularDistinctDegreeFactorizer(U: UnivariatePolynomialCategory(Integer)) is a package constructor  
--R Abbreviation for ModularDistinctDegreeFactorizer is MDDFACT  
--R This constructor is exposed in this frame.  
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MDDFACT  
--R  
--R------------------------------------------------ Operations -------------------------------  
--R exptMod : (U,Integer,U,Integer) -> U  factor : (U,Integer) -> List(U)  
--R gcd : (U,U,Integer) -> U  linears : (U,Integer) -> U  
--R ddFact : (U,Integer) -> List(Record(factor: U,degree: Integer))  
--R separateFactors : (List(Record(factor: U,degree: Integer)),Integer) -> List(U)  
--R  
--E 1  

)spool  
)lisp (bye)
--- ModularDistinctDegreeFactorizer.help ---

================================================================================
ModularDistinctDegreeFactorizer examples
================================================================================

This package supports factorization and gcds of univariate polynomials over the integers modulo different primes. The inputs are given as polynomials over the integers with the prime passed explicitly as an extra argument.

See Also:
o )show ModularDistinctDegreeFactorizer

———

ModularDistinctDegreeFactorizer (MDDFACT)

Exports:
factor gcd linears ddFact exptMod separateFactors

——— package MDDFACT ModularDistinctDegreeFactorizer ——

)abbrev package MDDFACT ModularDistinctDegreeFactorizer
++ Author: Barry Trager
++ Date Last Updated: 20.9.95 (JHD)
++ Description:
++ This package supports factorization and gcds
++ of univariate polynomials over the integers modulo different
++ primes. The inputs are given as polynomials over the integers
++ with the prime passed explicitly as an extra argument.
ModularDistinctDegreeFactorizer(U): C == T where
U : UnivariatePolynomialCategory(Integer)
I ==> Integer
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
V ==> Vector
L ==> List
DDRecord ==> Record(factor:EMR, degree:I)
UDDRecord ==> Record(factor:U, degree:I)
DDList ==> L DDRecord
UDDList ==> L UDDRecord

C == with
  gcd:(U,U,I) -> U
  linears: (U,I) -> U
  factor:(U,I) -> L U
  ddFact:(U,I) -> UDDList
  separateFactors:(UDDList,I) -> L U
  exptMod:(U,I,U,I) -> U

T == add
    zero? p => u
    map((i1:I):I +-> positiveRemainder(i1,p),u)
  merge(p:I,q:I):Union(I,"failed") ==
    p = q => p
    p = 0 => q
    q = 0 => p
    "failed"
  modInverse(c:I,p:I):I ==
    (extendedEuclidean(c,p,1)::Record(coef1:I,coef2:I)).coef1
    invlcv := modInverse(leadingCoefficient v,p)
r := monicDivide(u, reduction(invlcv*v, p))
reduction(r.remainder, p) ^= 0 => "failed"
reduction(invlcv*r.quotient, p)

EMR := EuclideanModularRing(Integer, U, Integer,
     reduction, merge, exactquo)

probSplit2: (EMR, EMR, I) -> Union(List EMR, "failed")
trace: (EMR, I, EMR) -> EMR
ddfactor: EMR -> L EMR
ddfact: EMR -> DDList
sepfact1: DDRecord -> L EMR
sepfact: DDList -> L EMR
probSplit: (EMR, EMR, I) -> Union(L EMR, "failed")
makeMonic: EMR -> EMR
exptmod: (EMR, I, EMR) -> EMR

lc(u: EMR): I == leadingCoefficient(u::U)
degree(u: EMR): I == degree(u::U)
makeMonic(u) == modInverse(lc(u), modulus(u)) * u

i: I

exptmod(u1, i, u2) ==
i < 0 => error("negative exponentiation not allowed for exptMod")
ans:= 1$EMR
while i > 0 repeat
    if odd?(i) then ans:= (ans * u1) rem u2
    i:= i quo 2
    u1:= (u1 * u1) rem u2
ans

exptmod(a, i, b, q) ==
ans:= exptmod(reduce(a, q), i, reduce(b, q))
ans::U

ddfactor(u) ==
if (c:= lc(u)) ^= 1$I then u:= makeMonic(u)
ans:= sepfact(ddfact(u))
cons(c::EMR, [makeMonic(f) for f in ans | degree(f) > 0])
gcd(u, v, q) == gcd(reduce(u, q), reduce(v, q))::U

factor(u, q) ==
v:= reduce(u, q)
dv:= reduce(differentiate(u), q)
degree gcd(v, dv) > 0 =>
    error("Modular factor: polynomial must be squarefree")
ans:= ddfactor v
[f::U for f in ans]
ddfact(u) ==
  p := modulus u
  w := reduce(monomial(1, 1)$U, p)
  m := w
  d := 1
  if (c := lc(u)) ^= 1$I then u := makeMonic u
  ans := []
  repeat
    w := exptmod(w, p, u)
    g := gcd(w - m, u)
    if degree g > 0 then
      g := makeMonic(g)
      ans := [g, d],: ans
      u := (u quo g)
      degree(u) = 0 => return [[c::EMR, 0$I], :ans]
      d := d+1
      d > (degree(u):I quo 2) =>
        return [[c::EMR, 0$I], [u, degree(u)], :ans]

ddFact(u, q) ==
  ans := ddfact(reduce(u, q))
  [[(dd.factor)::U, dd.degree]$UDDRecord for dd in ans]$UDDList

linears(u, q) ==
  uu := reduce(u, q)
  m := reduce(monomial(1, 1)$U, q)
  gcd(exptmod(m, q, uu) - m, uu)::U

sepfact(factList) ==
  "append"/[sepFact1(f) for f in factList]

separateFactors(uddList, q) ==
  ans := sepfact [[reduce(udd.factor, q), udd.degree]$DDRecord for
                  udd in uddList]$DDList
  [f::U for f in ans]

decode(s: Integer, p: Integer, x: U): U ==
  s < p => s::U
  qr := divide(s, p)
  qr.remainder :: U + x*decode(qr.quotient, p, x)

sepFact1(f) ==
  u := f.factor
  p := modulus u
  (d := f.degree) = 0 => [u]
  if (c := lc(u)) ^= 1$I then u := makeMonic(u)
  d = (du := degree(u)) => [u]
  ans := L EMR := []
  x: U := monomial(1, 1)
  -- for small primes find linear factors by exhaustion
\[d=1 \text{ and } p < 1000 \Rightarrow\]

\[
\text{for } i \text{ in } 0.. \text{ while } du > 0 \text{ repeat}
\]

\[
\text{if } u(i::U) = 0 \text{ then ans := cons(reduce(x-(i::U),p),ans)}
\]

\[
du := du-1
\]

\[
\text{ans}
\]

\[
y := x
\]

\[
s := 0
\]

\[
ss := 1
\]

\[
\text{stack:L EMR} := [u]
\]

\[
\text{until null stack repeat}
\]

\[
t := \text{reduce}(((s::U)+x),p)
\]

\[
\text{if not ((flist := probSplit(first stack,t,d)) case "failed") then stack := rest stack}
\]

\[
\text{for fact in flist repeat}
\]

\[
f1 := \text{makeMonic(fact)}
\]

\[
\text{if } (\text{df1 := degree(f1)}) = 0 \Rightarrow \text{nil}
\]

\[
df1 > d \Rightarrow \text{stack := [f1,stack]}
\]

\[
\text{ans} := [f1,\text{ans}]
\]

\[
p = 2 \Rightarrow
\]

\[
\text{ss} := \text{ss} + 1
\]

\[
x := y \ast \text{decode(ss, p, y)}
\]

\[
s := s + 1
\]

\[
s := 0
\]

\[
ss := ss + 1
\]

\[
x := y \ast \text{decode(ss, p, y)}
\]

\[
\text{-- not one? leadingCoefficient(x) =>}
\]

\[
\text{not (leadingCoefficient(x) = 1) =>}
\]

\[
\text{ss} := p \ast \text{degree x}
\]

\[
x := y \ast \text{degree(x) + 1}
\]

\[
[c \ast \text{first(ans)}, \text{rest(ans)}]
\]

\[
\text{probSplit(u,t,d)} =
\]

\[
(p := \text{modulus(u)}) = 2 \Rightarrow \text{probSplit2(u,t,d)}
\]

\[
f1 := \text{gcd(u,t)}
\]

\[
r := ((p**d)\text{:-NNI}-1) \text{ quo } 2 : \text{NNI}
\]

\[
n := \text{exptmod}(t,r,u)
\]

\[
f2 := \text{gcd}(u,n + 1)
\]

\[
(g := f1 \ast f2) = 1 \Rightarrow "\text{failed}"
\]

\[
g = u = "\text{failed}"
\]

\[
[f1,f2,(u \text{ quo g}]]
\]

\[
\text{probSplit2(u,t,d)} =
\]

\[
f := \text{gcd(u,trace(t,d,u))}
\]

\[
f = 1 \Rightarrow "\text{failed}"
\]

\[
\text{degree u} = \text{degree f} \Rightarrow "\text{failed}"
\]

\[
[1,f,u \text{ quo f}]
\]

\[
\text{trace(t,d,u)} =
\]
\begin{verbatim}

p:=modulus(t)
d:= d - 1

while d > 0 repeat
    tt:= (tt + (t:=exptmod(t, p, u))) rem u
    d:= d - 1
    tt

---

-MDDFACT.dotabb---

"MDDFACT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MDDFACT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MDDFACT" -> "PFECAT"

---

package MHROWRED ModularHermitianRowReduction

--- ModularHermitianRowReduction.input ---

)set break resume
)sys rm -f ModularHermitianRowReduction.output
)spool ModularHermitianRowReduction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ModularHermitianRowReduction
--R
--R ModularHermitianRowReduction(R: EuclideanDomain) is a package constructor
--R Abbreviation for ModularHermitianRowReduction is MHROWRED
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MHROWRED
--R
--R--------------------------- Operations ---------------------------
--R rowEch : Matrix(R) -> Matrix(R)
--R normalizedDivide : (R,R) -> Record(quotient: R,remainder: R)
--R rowEchLocal : (Matrix(R),R) -> Matrix(R)
--R rowEchelon : (Matrix(R),R) -> Matrix(R)
--R rowEchelonLocal : (Matrix(R),R,R) -> Matrix(R)
--R

\end{verbatim}
ModularHermitianRowReduction (MHROWRED)
++ Modular hermitian row reduction.
-- should be moved into matrix whenever possible

ModularHermitianRowReduction(R): Exports == Implementation where
  R: EuclideanDomain

Z ==> Integer
V ==> Vector R
M ==> Matrix R
REC ==> Record(val:R, cl:Z, rw:Z)

Exports ==> with
  rowEch : M -> M
  ++ rowEch(m) computes a modular row-echelon form of m, finding
  ++ an appropriate modulus.
  rowEchelon : (M, R) -> M
  ++ rowEchelon(m, d) computes a modular row-echelon form mod d of
  ++ [d ]
  ++ [ d ]
  ++ [ . ]
  ++ [ d]
  ++ [ M ]
  ++ where \(\text{spad}\{M = m \mod d}\).
  rowEchLocal : (M, R) -> M
  ++ rowEchLocal(m,p) computes a modular row-echelon form of m, finding
  ++ an appropriate modulus over a local ring where p is the only prime.
  rowEchelonLocal: (M, R, R) -> M
  ++ rowEchelonLocal(m, d, p) computes the row-echelon form of m
  ++ concatenated with d times the identity matrix
  ++ over a local ring where p is the only prime.
  normalizedDivide: (R, R) -> Record(quotient:R, remainder:R)
  ++ normalizedDivide(n,d) returns a normalized quotient and
  ++ remainder such that consistently unique representatives
  ++ for the residue class are chosen, e.g. positive remainders

Implementation ==> add
  order : (R, R) -> Z
  vconc : (M, R) -> M
  non0 : (V, Z) -> Union(REC, "failed")
  nonzero?: V -> Boolean
  mkMat : (M, List Z) -> M
  diagSubMatrix: M -> Union(Record(val:R, mat:M), "failed")
  determinantOfMinor: M -> R
  enumerateBinomial: (List Z, Z, Z) -> List Z

  nonzero? v == any?(s +-> s ^= 0, v)

  -- returns [a, i, rown] if v = [0,...,0,a,0,...,0]
-- where $a \neq 0$ and $i$ is the index of $a$, "failed" otherwise.
non0(v, rown) ==
  ans:=REC
  allZero:Boolean := true
  for i in minIndex v .. maxIndex v repeat
    if qelt(v, i) ^= 0 then
      if allZero then
        allZero := false
        ans := [qelt(v, i), i, rown]
      else return "failed"
    allZero => "failed"
  ans

-- returns a matrix made from the non-zero rows of $x$ whose row number
-- is not in $l$
mkMat(x, l) ==
  empty?(ll := [parts row(x, i)
    for i in minRowIndex x .. maxRowIndex x |
    (not member?(i, l)) and nonzero? row(x, i)]$List(List R)) =>
    zero(1, ncols x)
    matrix ll

-- returns $[m, d]$ where $m = x$ with the zero rows and the rows of
-- the diagonal of $d$ removed, if $x$ has a diagonal submatrix of $d$'s,
-- "failed" otherwise.
diagSubMatrix x ==
  l := [u::REC for i in minRowIndex x .. maxRowIndex x |
    (u := non0(row(x, i), i)) case REC]
  for a in removeDuplicates([r.val for r in l]$List(R)) repeat
    {[r.cl for r in l | r.val = a]$List(Z)}$Set(Z) =
    {[z for z in minColIndex x .. maxColIndex x]$List(Z)}$Set(Z)
      => return [a, mkMat(x, [r.rw for r in l | a = r.val])]
  "failed"

-- returns a non-zero determinant of a minor of $x$ of rank equal to
-- the number of columns of $x$, if there is one, 0 otherwise
-- do not compute a modulus for square matrices, since this is as expensive
-- as the Hermite reduction itself
-- as the Hermite reduction itself
(determinantOfMinor x ==
  (nr := nrows x) <= (nc := ncols x) => 0
  lc := [i for i in minColIndex x .. maxColIndex x]$List(Integer)
  lr := [i for i in minRowIndex x .. maxRowIndex x]$List(Integer)
  for i in 1..(n := binomial(nr, nc)) repeat
    (d := determinant x(enumerateBinomial(lr, nc, i), lc)) ^= 0 =>
      j := i + 1 + (random()$Z rem (n - i))
      return gcd(d, determinant x(enumerateBinomial(lr, nc, j), lc))
  0

-- returns the $i$-th selection of $m$ elements of $l = (a_1, \ldots, a_n)$,
enumerateBinomial(l, m, i) ==
  m1 := minIndex l - 1
  zero?(m := m - 1) => [l(m1 + i)]
  for j in 1..(n := #l) repeat
    i <= (b := binomial(n - j, m)) =>
      return concat(l(m1 + j), enumerateBinomial(rest(l, j), m, i))
    i := i - b
  error "Should not happen"

rowEch x ==
  (u := diagSubMatrix x) case "failed" =>
    zero?(d := determinantOfMinor x) => rowEchelon x
    rowEchelon(x, d)
    rowEchelon(u.mat, u.val)
  vconc(y, m) ==
    vertConcat(diagonalMatrix new(ncols y, m)\$V, map(s +-> s rem m, y))
  order(m, p) ==
    zero? m => -1
    for i in 0.. repeat
      (mm := m exquo p) case "failed" => return i
      m := mm::R
    if R has IntegerNumberSystem then
      normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
        qr := divide(n, d)
        qr.remainder >= 0 => qr
        d > 0 =>
          qr.remainder := qr.remainder + d
          qr.quotient := qr.quotient - 1
          qr
        qr.remainder := qr.remainder - d
        qr.quotient := qr.quotient + 1
        qr
      else
        normalizedDivide(n:R, d:R):Record(quotient:R, remainder:R) ==
        divide(n, d)
    rowEchLocal(x, p) ==
      (u := diagSubMatrix x) case "failed" =>
        zero?(d := determinantOfMinor x) => rowEchelon x
        rowEchelonLocal(x, d, p)
        rowEchelonLocal(u.mat, u.val, p)
    rowEchelonLocal(y, m, p) ==
      m := p**(order(m,p)::NonNegativeInteger)
      x := vconc(y, m)
nrows := maxRowIndex x
ncols := maxColIndex x
minr := i := minRowIndex x
for j in minColIndex x .. ncols repeat
  if i > nrows then leave x
  rown := minr - 1
  pivord : Integer
  npivord : Integer
  for k in i .. nrows repeat
    qelt(x,k,j) = 0 => "next k"
    npivord := order(qelt(x,k,j), p)
    (rown = minr - 1) or (npivord < pivord) =>
      rown := k
      pivord := npivord
  rown = minr - 1 => "enuf"
  x := swapRows!(x, i, rown)
(a, b, d) := extendedEuclidean(qelt(x,i,j), m)
qsetelt_!(x,i,j,d)
pivot := d
for k in j+1 .. ncols repeat
  qsetelt_!(x,i,k, a * qelt(x,i,k) rem m)
for k in i+1 .. nrows repeat
  zero? qelt(x,k,j) => "next k"
  q := (qelt(x,k,j) exquo pivot) :: R
  for k1 in j+1 .. ncols repeat
    v2 := (qelt(x,k,k1) - q * qelt(x,i,k1)) rem m
    qsetelt_!(x, k, k1, v2)
  qsetelt_!(x, k, j, 0)
for k in minr .. i-1 repeat
  zero? qelt(x,k,j) => "enuf"
  qr := normalizedDivide(qelt(x,k,j), pivot)
  qsetelt_!(x,k,j, qr.remainder)
  for k1 in j+1 .. ncols x repeat
    qsetelt_!(x,k,k1, (qelt(x,k,k1) - qr.quotient * qelt(x,i,k1)) rem m)
i := i+1
x
if R has Field then
  rowEchelon(y, m) == rowEchelon vconc(y, m)
else
  rowEchelon(y, m) ==
  x := vconc(y, m)
nrows := maxRowIndex x
ncols := maxColIndex x
minr := i := minRowIndex x
for j in minColIndex x .. ncols repeat
  if i > nrows then leave
rown := minr - 1
for k in i .. nrows repeat
  if (qelt(x,k,j) ^= 0) and ((rown = minr - 1) or
    sizeLess?(qelt(x,k,j), qelt(x,rown,j))) then rown := k
rown = minr - 1 => "next j"
x := swapRows_!(x, i, rown)
for k in i+1 .. nrows repeat
  zero? qelt(x,k,j) => "next k"
  (a, b, d) := extendedEuclidean(qelt(x,i,j), qelt(x,k,j))
  (b1, a1) :=
    ((qelt(x,i,j) exquo d)::R, (qelt(x,k,j) exquo d)::R)
    -- a*b1+a1*b = 1
  for k1 in j+1 .. ncols repeat
    v1 := (a * qelt(x,i,k1) + b * qelt(x,k,k1)) rem m
    v2 := (b1 * qelt(x,k,k1) - a1 * qelt(x,i,k1)) rem m
    qsetelt_!(x, i, k1, v1)
    qsetelt_!(x, k, k1, v2)
    qsetelt_!(x, i, j, d)
    qsetelt_!(x, k, j, 0)
un := unitNormal qelt(x,i,j)
qsetelt_!(x, i, j, un.canonical)
if un.associate ^= 1 then for jj in (j+1)..ncols repeat
  qsetelt_!(x,i,jj,un.associate * qelt(x,i,jj))
xij := qelt(x,i,j)
for k in minr .. i-1 repeat
  zero? qelt(x,k,j) => "next k"
  qr := normalizedDivide(qelt(x,k,j), xij)
  qsetelt_!(x,k,j, qr.remainder)
  for k1 in j+1 .. ncols x repeat
    qsetelt_!(x,k,k1,
      (qelt(x,k,k1) - qr.quotient * qelt(x,i,k1)) rem m)
i := i+1
x

— MHROWRED.dotabb —

"MHROWRED" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MHROWRED"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"MHROWRED" -> "IVECTOR"
package MRF2 MonoidRingFunctions2

— MonoidRingFunctions2.input —

)set break resume
)sys rm -f MonoidRingFunctions2.output
)spool MonoidRingFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MonoidRingFunctions2
--R
--R MonoidRingFunctions2(R: Ring,S: Ring,M: Monoid) is a package constructor
--R Abbreviation for MonoidRingFunctions2 is MRF2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MRF2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((R -> S),MonoidRing(R,M)) -> MonoidRing(S,M)
--R
--E 1

)spool
)lisp (bye)

— MonoidRingFunctions2.help —

====================================================================
MonoidRingFunctions2 examples
====================================================================

MonoidRingFunctions2 implements functions between two monoid rings
defined with the same monoid over different rings.

See Also:
o )show MonoidRingFunctions2

———
MonoidRingFunctions2 (MRF2)

Exports:
map

— package MRF2 MonoidRingFunctions2 —

)`abbrev package MRF2 MonoidRingFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 14 May 1991
++ Date Last Updated: 14 May 1991
++ Description:
++ MonoidRingFunctions2 implements functions between
++ two monoid rings defined with the same monoid over different rings.

MonoidRingFunctions2(R,S,M) : Exports == Implementation where
  R : Ring
  S : Ring
  M : Monoid
Exports ==> with
  map: (R -> S, MonoidRing(R,M)) -> MonoidRing(S,M)
    ++ map(f,u) maps f onto the coefficients f the element
    ++ u of the monoid ring to create an element of a monoid
    ++ ring with the same monoid b.
Implementation ==> add
  map(fn, u) ==
    res : MonoidRing(S,M) := 0
    for te in terms u repeat
      res := res + monomial(fn(te.coef), te.monom)
    res

— MRF2.dotabb —
package MONOTOOL MonomialExtensionTools

— MonomialExtensionTools.input —

)set break resume
)sys rm -f MonomialExtensionTools.output
)spool MonomialExtensionTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MonomialExtensionTools
--) MonomialExtensionTools(F: Field,UP: UnivariatePolynomialCategory(F)) is a package constructor
--) Abbreviation for MonomialExtensionTools is MONOTOOL
--) This constructor is not exposed in this frame.
--) Issue )edit bookvol10.4.pamphlet to see algebra source code for MONOTOOL
--) Operations

--R decompose : (Fraction(UP),(UP -> UP)) -> Record(poly: UP,normal: Fraction(UP),special: Fraction(UP))
--R normalDenom : (Fraction(UP),(UP -> UP)) -> UP
--R split : (UP,(UP -> UP)) -> Record(normal: UP,special: UP)
--R splitSquarefree : (UP,(UP -> UP)) -> Record(normal: Factored(UP),special: Factored(UP))

--) E 1

)spool
)lisp (bye)

— MonomialExtensionTools.help —

====================================================================
MonomialExtensionTools examples
====================================================================
Tools for handling monomial extensions.

See Also:
- )show MonomialExtensionTools

---

**MonomialExtensionTools (MONOTOOL)**

```spad
MONOTOOL

PFECAT
```

Exports:
- decompose
- normalDenom
- split
- splitSquarefree

–– package MONOTOOL MonomialExtensionTools ––

```spad
)abbrev package MONOTOOL MonomialExtensionTools
++ Author: Manuel Bronstein
++ Date Created: 18 August 1992
++ Date Last Updated: 3 June 1993
++ Description:
++ Tools for handling monomial extensions.

MonomialExtensionTools(F, UP): Exports == Implementation where
  F : Field
  UP: UnivariatePolynomialCategory F

RF =>> Fraction UP
FR =>> Factored UP

Exports =>> with
  split : (UP, UP -> UP) -> Record(normal:UP, special:UP)
    ++ split(p, D) returns \spad{[n,s]} such that \spad{p = n s},
    ++ all the squarefree factors of n are normal w.r.t. D,
```
++ and s is special w.r.t. D.
++ D is the derivation to use.

splitSquarefree: (UP, UP -> UP) -> Record(normal:FR, special:FR)
++ splitSquarefree(p, D) returns
++ \spad{[n_1 n_2^2 \ldots n_m^m, s_1 s_2^2 \ldots s_q^q]} such that
++ \spad{p = n_1 n_2^2 \ldots n_m^m s_1 s_2^2 \ldots s_q^q}, each
++ \spad{n_i} is normal w.r.t. D and each \spad{s_i} is special
++ w.r.t D.
++ D is the derivation to use.

normalDenom: (RF, UP -> UP) -> UP
++ normalDenom(f, D) returns the product of all the normal factors
++ of \spad{denom(f)}.
++ D is the derivation to use.

decompose : (RF, UP -> UP) -> Record(poly:UP, normal:RF, special:RF)
++ decompose(f, D) returns \spad{[p,n,s]} such that \spad{f = p+n+s},
++ all the squarefree factors of \spad{denom(n)} are normal w.r.t. D,
++ \spad{denom(s)} is special w.r.t. D,
++ and n and s are proper fractions (no pole at infinity).
++ D is the derivation to use.

Implementation ==> add

normalDenom(f, derivation) == split(denom f, derivation).normal

split(p, derivation) ==
  pbar := (gcd(p, derivation p) exquo gcd(p, differentiate p))::UP
  zero? degree pbar => [p, 1]
  rec := split((p exquo pbar)::UP, derivation)
  [rec.normal, pbar * rec.special]

splitSquarefree(p, derivation) ==
  s:Factored(UP) := 1
  n := s
  q := squareFree p
  for rec in factors q repeat
    r := rec.factor
    g := gcd(r, derivation r)
    if not ground? g then s := s * sqfrFactor(g, rec.exponent)
    h := (r exquo g)::UP
    if not ground? h then n := n * sqfrFactor(h, rec.exponent)
  [n, unit(q) * s]

decompose(f, derivation) ==
  qr := divide(numer f, denom f)
  -- rec.normal = rec.special = denom f
  -- rec := split(denom f, derivation)
  -- eeu.coef1 = rec.normal + eeu.coef2 = rec.special = qr.remainder
  -- and degree(eeu.coef1) < degree(rec.special)
  -- and degree(eeu.coef2) < degree(rec.normal)
  -- qr.remainder/denom(f) = eeu.coef1 / rec.special + eeu.coef2 / rec.normal
  eeu := extendedEuclidean(rec.normal, rec.special,
qr.remainder)::Record(coef1:UP, coef2:UP)
[qr.quotient, eeu.coef2 / rec.normal, eeu.coef1 / rec.special]

— MONOTOOL.dotabb —

"MONOTOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MONOTOOL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MONOTOOL" -> "PFECAT"

package MSYSCMD MoreSystemCommands

— MoreSystemCommands.input —

)set break resume
)sys rm -f MoreSystemCommands.output
)spool MoreSystemCommands.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MoreSystemCommands
--R
--R MoreSystemCommands is a package constructor
--R Abbreviation for MoreSystemCommands is MSYSCMD
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MSYSCMD
--R
--R----------------------------------------------- Operations -----------------------------------------------
--R systemCommand : String -> Void
--R
--E 1

)spool
)lisp (bye)

— MoreSystemCommands.help —
MoreSystemCommands implements an interface with the system command facility. These are the commands that are issued from source files or the system interpreter and they start with a close parenthesis, e.g., the "what" commands.

See Also:
- )show MoreSystemCommands

MoreSystemCommands (MSYSCMD)

Exports:

- systemCommand

--- package MSYSCMD MoreSystemCommands ---

)abbrev package MSYSCMD MoreSystemCommands
++ Description:
++ \spadtype{MoreSystemCommands} implements an interface with the
++ system command facility. These are the commands that are issued
++ from source files or the system interpreter and they start with
++ a close parenthesis, for example, the "what" commands.

MoreSystemCommands: public == private where

public == with

    systemCommand: String -> Void
++ systemCommand(cmd) takes the string `\spadvar{cmd}` and passes
++ it to the runtime environment for execution as a system
++ command. Although various things may be printed, no usable
++ value is returned.

private == add

systemCommand cmd == doSystemCommand(cmd)$Lisp

———

— MSYSCMD.dotabb —

"MSYSCMD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MSYSCMD"]
"Package" [color="#FF4488"]
"MSYSCMD" -> "Package"

———

package MPCPF MPolyCatPolyFactorizer

— MPolyCatPolyFactorizer.input —

)set break resume
)sys rm -f MPolyCatPolyFactorizer.output
)spool MPolyCatPolyFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MPolyCatPolyFactorizer
  --R
  --R MPolyCatPolyFactorizer(E: OrderedAbelianMonoidSup,OV,R: EuclideanDomain,PPR: PolynomialCategory(Polynomial(R),E,OV)) where
  --R OV: OrderedSet with
  --R convert : % -> Symbol
  --R variable : Symbol -> Union(%,"failed") is a package constructor
  --R Abbreviation for MPolyCatPolyFactorizer is MPCPF
  --R This constructor is not exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for MPCPF
  --R
  --R---------------------------------- Operations ----------------------------------
  --R factor : PPR -> Factored(PPR)
  --R
This package exports a factor operation for multivariate polynomials with coefficients which are polynomials over some ring $R$ over which we can factor. It is used internally by packages such as the solve package which need to work with polynomials in a specific set of variables with coefficients which are polynomials in all the other variables.

See Also:
- )show MPolyCatPolyFactorizer

MPolyCatPolyFactorizer (MPCPF)

Exports:
- factor

---

)abbrev package MPCPF MPolyCatPolyFactorizer
++ Author: P. Gianni
++ Date Last Updated: March 1995
++ Description:
++ This package exports a factor operation for multivariate polynomials
++ with coefficients which are polynomials over
++ some ring \( R \) over which we can factor. It is used internally by packages
++ such as the solve package which need to work with polynomials in a specific
++ set of variables with coefficients which are polynomials in all the other
++ variables.

\[
\text{MPolyCatPolyFactorizer}(E, OV, R, PPR) : C == T
\]
where
\[
\begin{align*}
R & : \text{EuclideanDomain} \\
E & : \text{OrderedAbelianMonoidSup} \\
OV & : \text{OrderedSet with convert : } % \to \text{Symbol} \\
& \quad \quad \quad \text{convert(x) converts x to a symbol} \\
& \quad \quad \quad \text{variable: Symbol } \to \text{Union(\%, "failed")} \\
& \quad \quad \quad \quad \quad \text{variable(s) makes an element from symbol s or fails.} \\
PR & \to \text{Polynomial R} \\
PPR & \to \text{PolynomialCategory(PR, E, OV)} \\
NNI & \to \text{NonNegativeInteger} \\
ISY & \to \text{IndexedExponents Symbol} \\
SE & \to \text{Symbol} \\
UP & \to \text{SparseUnivariatePolynomial R} \\
UPPR & \to \text{SparseUnivariatePolynomial PPR}
\end{align*}
\]

\[
C == \text{with}
\]
\[
\begin{align*}
factor & : \text{PPR } \to \text{Factored PPR} \\
& \quad \quad \text{factor(p) factors a polynomial with polynomial} \\
& \quad \quad \text{coefficients.}
\end{align*}
\]

--- Local Functions ----

\[
T == \text{add}
\]

import PushVariables(R, E, OV, PPR)

---- factorization of p ----

\[
factor(p: PPR) : \text{Factored PPR} ==
\]
\[
\begin{align*}
\text{ground? } p & \Rightarrow \text{nilFactor}(p, 1) \\
c & := \text{content } p \\
p & := (p \text{ exquo } c):PPR \\
vars & : \text{List } OV \ := \text{variables } p \\
g & :\text{PR} := \text{retract pushdown}(p, \text{vars}) \\
f\text{list} & := \text{factor}(g)\text{GeneralizedMultivariateFactorize(Symbol, ISY, R, R, PR)} \\
f\text{fact} & : \text{List(Record(}\text{irr:PPR, pow:Integer)}) \\
f\text{fact} & := [\text{pushup(u.factor::PPR, vars, u.exponent] for u in factors f\text{list]} \\
f\text{cont} & := \text{unit f\text{list}}::PPR \\
nil\text{Factor}(c*f\text{cont}, 1)*(\_*/\text{[primeFactor(ff.irr, ff.pow) for ff in f\text{fact]}])}
\end{align*}
\]
package MPRFF MPolyCatRationalFunctionFactorizer

---

MPCPF.dotabb

"MPCPF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPCPF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPCPF" -> "PFECAT"

---

package MPRFF MPolyCatRationalFunctionFactorizer

---

MPolyCatRationalFunctionFactorizer.input

)set break resume
)sys rm -f MPolyCatRationalFunctionFactorizer.output
)spool MPolyCatRationalFunctionFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MPolyCatRationalFunctionFactorizer

--R MPolyCatRationalFunctionFactorizer(E: OrderedAbelianMonoidSup,OV,R: IntegralDomain,PRF: PolynomialCategory(Fraction(Polynomial(R)),E,OV))where
--R OV: OrderedSetwith
--R convert : % -> Symbol is a package constructor
--R Abbreviation for MPolyCatRationalFunctionFactorizer is MPRFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MPRFF
--R
--R----------------------------- Operations -----------------------------
--R factor : PRF -> Factored(PRF) pushdown : (PRF,OV) -> PRF
--R pushup : (PRF,OV) -> PRF
--R pushdterm : (SparseUnivariatePolynomial(PRF),OV) -> PRF
--R pushucoeff : (SparseUnivariatePolynomial(Polynomial(R)),OV) -> PRF
--R pushuconst : (Fraction(Polynomial(R)),OV) -> PRF
--R totalfract : PRF -> Record(sup: Polynomial(R),inf: Polynomial(R))
--R
--E 1

)spool
)lisp (bye)
This package exports a factor operation for multivariate polynomials
with coefficients which are rational functions over some ring \( R \) over
which we can factor. It is used internally by packages such as primary
decomposition which need to work with polynomials with rational
function coefficients, i.e. themselves fractions of polynomials.

See Also:
o )show MPolyCatRationalFunctionFactorizer

MPolyCatRationalFunctionFactorizer (MPRFF)

Exports:
factor pushdown pushup pushdterm pushucoef
pushuconst totalfract

package MPRFF MPolyCatRationalFunctionFactorizer —

)abbrev package MPRFF MPolyCatRationalFunctionFactorizer
++ Author: P. Gianni
++ Description:
++ This package exports a factor operation for multivariate polynomials
++ with coefficients which are rational functions over
++ some ring \( R \) over which we can factor. It is used internally by packages
such as primary decomposition which need to work with polynomials with rational function coefficients, i.e. themselves fractions of polynomials.

MPolyCatRationalFunctionFactorizer(E,OV,R,PRF) : C == T
where
  R : IntegralDomain
  F ==> Fraction Polynomial R
  RN ==> Fraction Integer
  E : OrderedAbelianMonoidSup
  OV : OrderedSet with
         convert : % -> Symbol
              ++ convert(x) converts x to a symbol
  PRF : PolynomialCategory(F,E,OV)
  NNI ==> NonNegativeInteger
  P ==> Polynomial R
  ISE ==> IndexedExponents SE
  SE ==> Symbol
  UP ==> SparseUnivariatePolynomial P
  UF ==> SparseUnivariatePolynomial F
  UPRF ==> SparseUnivariatePolynomial PRF
  QuoForm ==> Record(sup:P,inf:P)

  C == with
      totalfract : PRF -> QuoForm
          ++ totalfract(prf) takes a polynomial whose coefficients are themselves fractions of polynomials and returns a record containing the numerator and denominator resulting from putting prf over a common denominator.
      pushdown : (PRF,OV) -> PRF
          ++ pushdown(prf,var) pushes all top level occurrences of the variable var into the coefficient domain for the polynomial prf.
      pushdterm : (UPRF,OV) -> PRF
          ++ pushdterm(monom,var) pushes all top level occurrences of the variable var into the coefficient domain for the monomial monom.
      pushup : (PRF,OV) -> PRF
          ++ pushup(prf,var) raises all occurrences of the variable var in the coefficients of the polynomial prf back to the polynomial level.
      pushucoef : (UP,OV) -> PRF
          ++ pushucoef(upoly,var) converts the anonymous univariate polynomial upoly to a polynomial in var over rational functions.
      pushuconst : (F,OV) -> PRF
          ++ pushuconst(r,var) takes a rational function and raises all occurrences of the variable var to the polynomial level.
      factor : PRF -> Factored PRF
          ++ factor(prf) factors a polynomial with rational function coefficients.

      --- Local Functions ----


T == add

----- factorization of p -----
factor(p:PRF) : Factored PRF ==
  truelist:List OV :=variables p
  tp:=totalfract(p)
  nump:P:= tp.sup
  denp:F:=inv(tp.inf ::F)
  ffact : List(Record(irr:PRF,pow:Integer))
  flist:Factored P
  if R is Fraction Integer then
    flist:=
      ((factor nump)$MRationalFactorize(ISE,SE,Integer,P))
      pretend (Factored P)
  else
    if R has FiniteFieldCategory then
      flist:= ((factor nump)$MultFiniteFactorize(SE,ISE,R,P))
      pretend (Factored P)
    else
      if R has Field then error "not done yet"
      else
        if R has CharacteristicZero then
          flist:= ((factor nump)$MultivariateFactorize(SE,ISE,R,P))
          pretend (Factored P)
        else error "can't happen"
  ffact:=[[u.factor::F::PRF,u.exponent] for u in factors flist]
  fcont:=(unit flist)::F::PRF
  for x in truelist repeat
    fcont:=pushup(fcont,x)
  ffact:=[[pushup(ff.irr,x),ff.pow] for ff in ffact]
  (denp*fcont)*(_*/[primeFactor(ff.irr,ff.pow) for ff in ffact])

-- the following functions are used to "push" x in the coefficient ring --

----- push x in the coefficient domain for a polynomial -----
pushdown(g:PRF,x:OV) : PRF ==
  ground? g => g
  rf:PRF:=0$PRF
  ug:=univariate(g,x)
  while ug=0 repeat
    rf:=rf+pushdterm(ug,x)
    ug := reductum ug
  rf

----- push x in the coefficient domain for a term -----
pushdterm(t:UPRF,x:OV):PRF ==
n:=degree(t)
cf := \text{monomial}(1, \text{convert } x, n) \cdot F \quad \text{leadingCoefficient } t

\text{pushup}(f: \text{PRF}, x: \text{OV}) : \text{PRF} ==
\text{ground? } f \Rightarrow \text{pushuconst} (\text{retract } f, x)
\text{v} := \text{mainVariable}(f) :: \text{OV}
\text{g} := \text{univariate}(f, v)
\text{multivariate}(\text{map}((y: \text{PRF}): \text{PRF} \Rightarrow \text{pushup}(y, x), g), v)

\text{pushuconst}(r:F,x:OV): \text{PRF} ==
\text{xs: SE := convert } x
\text{degree(denom } r, \text{xs}) > 0 \Rightarrow \text{error } "\text{bad polynomial form}"
\text{inv}((\text{denom } r) :: F) \ast \text{pushucoef} (\text{univariate(numer } r, \text{xs}), x)

\text{pushucoef}(c: \text{UP}, x: \text{OV}) : \text{PRF} ==
c = 0 \Rightarrow 0
\text{monomial}((\text{leadingCoefficient } c) :: F :: \text{PRF}, x, \text{degree } c) +
\text{pushucoef} (\text{reductum } c, x)

\text{totalfract}(p: \text{PRF}) : \text{QuoForm} ==
p = 0 \Rightarrow [0$P, 1$P]$\text{QuoForm}
\text{for } x \text{ in variables } p \text{ repeat } p := \text{pushdown}(p, x)
g := \text{retract } p
[\text{numer } g, \text{denom } g]$\text{QuoForm}

\text{MPolyCatFunctions2.input}
CHAPTER 14. CHAPTER M

)set break resume
)sys rm -f MPolyCatFunctions2.output
)spool MPolyCatFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MPolyCatFunctions2
--R
--) MPolyCatFunctions2(VarSet: OrderedSet,E1: OrderedAbelianMonoidSup,E2: OrderedAbelianMonoidSup,R: Ring,S: Ring,PR: PolynomialCategory(R,E1,VarSet),PS: PolynomialCategory(S,E2,VarSet)) is a package constructor
--) Abbreviation for MPolyCatFunctions2 is MPC2
--) This constructor is exposed in this frame.
--) Issue )edit bookvol10.4.pamphlet to see algebra source code for MPC2
--)  
--)--------------------------------- Operations ---------------------------------  
--) map : ((R -> S),PR) -> PS  reshape : (List(S),PR) -> PS
--)  
--) E 1

)spool
)lisp (bye)

——

— MPolyCatFunctions2.help —

=================================================================================
MPolyCatFunctions2 examples
=================================================================================

Utilities for MPolyCat

See Also:
  o )show MPolyCatFunctions2

——
MPC2 MPolyCatFunctions2 (MPC2)

Exports:
map  reshape

package MPC2 MPolyCatFunctions2

Abbrev package MPC2 MPolyCatFunctions2
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 28 March 1990 (PG)
++ Description:
++ Utilities for MPolyCat

MPCPolyCatFunctions2(VarSet,E1,E2,R,S,PR,PS) : public == private where

VarSet : OrderedSet
E1 : OrderedAbelianMonoidSup
E2 : OrderedAbelianMonoidSup
R : Ring
S : Ring
PR : PolynomialCategory(R,E1,VarSet)
PS : PolynomialCategory(S,E2,VarSet)
SUPR ==> SparseUnivariatePolynomial PR
SUPS ==> SparseUnivariatePolynomial PS

public == with
map: (R -> S,PR) -> PS
++ map(f,p) \ undocumented
reshape: (List S, PR) -> PS
++ reshape(l,p) \ undocumented

private == add

supMap: (R -> S, SUPR) -> SUPS

supMap(fn : R -> S, supr : SUPR) : SUPS ==
\[
\text{supr} = 0 \Rightarrow \text{monomial}(\text{fn}(0) :: \text{PS}, 0) \text{SUPS}
\]
\[
c : \text{PS} := \text{map}(\text{fn}, \text{leadingCoefficient} \text{ supr}) \%
\]
\[
\text{monomial}(c, \text{degree} \text{ supr}) \text{SUPS} + \text{supMap}(\text{fn}, \text{reductum} \text{ supr})
\]
\[
\text{map}(\text{fn} : \text{R} \rightarrow \text{S}, \text{pr} : \text{PR}) : \text{PS} =
\]
\[
\text{varu} : \text{Union}(\text{VarSet}, "\text{failed}") := \text{mainVariable} \text{ pr}
\]
\[
\text{varu case } "\text{failed}" \Rightarrow -- \text{have a constant}
\]
\[
\text{fn}(\text{retract} \text{ pr}) :: \text{PS}
\]
\[
\text{var} : \text{VarSet} := \text{varu} :: \text{VarSet}
\]
\[
\text{supr} : \text{SUPR} := \text{univariate}(\text{pr}, \text{var}) \text{PR}
\]
\[
\text{multivariate}(\text{supMap}(\text{fn}, \text{supr}), \text{var}) \text{PS}
\]

---

--- MPC2.dotabb ---

"MPC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPC2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPC2" -> "PFECAT"

---

package MPC3 MPolyCatFunctions3

--- MPolyCatFunctions3.input ---

)set break resume
)sys rm -f MPolyCatFunctions3.output
)spool MPolyCatFunctions3.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MPolyCatFunctions3
--R
--R MPolyCatFunctions3(Vars1: OrderedSet, Vars2: OrderedSet, E1: OrderedAbelianMonoidSup, E2: OrderedAbelianMonoidSup, R: Ring, PR1: PolynomialCategory(R, E1, Vars1), PR2: PolynomialCategory(R, E2, Vars2)) is a package constructor
--R Abbreviation for MPolyCatFunctions3 is MPC3
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MPC3
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((Vars1 -> Vars2), PR1) -> PR2
--R
--E 1

)spool
)lisp (bye)

― MPolyCatFunctions3.help ―

====================================================================
MPolyCatFunctions3 examples
====================================================================

This package has no description

See Also:
o )show MPolyCatFunctions3

― MPolyCatFunctions3 (MPC3) ―

Exports:
map

― package MPC3 MPolyCatFunctions3 ―

)abbrev package MPC3 MPolyCatFunctions3
++ Description:
++ This package has no description

MPolyCatFunctions3(Vars1,Vars2,E1,E2,R,PR1,PR2): C == T where
E1 : OrderedAbelianMonoidSup
E2 : OrderedAbelianMonoidSup
Vars1: OrderedSet
Vars2: OrderedSet
R : Ring
PR1 : PolynomialCategory(R,E1,Vars1)
PR2 : PolynomialCategory(R,E2,Vars2)

C ==> with
  map: (Vars1 -> Vars2, PR1) -> PR2
++ map(f,x) \ undocumented

T ==> add
map(f:Vars1 -> Vars2, p:PR1):PR2 ==
  (x1 := mainVariable p) case "failed" =>
  c:=R:=(retract p)
  c::PR2
  up := univariate(p, x1::Vars1)
  x2 := f(x1::Vars1)
  ans:PR2 := 0
  while up ^= 0 repeat
    ans := ans + monomial(map(f,leadingCoefficient up),x2,degree up)
    up := reductum up
  ans

—— MPC3.dotabb ——

"MPC3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MPC3"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MPC3" -> "PFECAT"

—— package MRATFAC MRationalFactorize ——

MRationalFactorize.input ——

)set break resume
)sys rm -f MRationalFactorize.output
)spool MRationalFactorize.output
)set message test on
MRationalFactorize examples

MRationalFactorize contains the factor function for multivariate polynomials over the quotient field of a ring R such that the package MultivariateFactorize can factor multivariate polynomials over R.

See Also:
  o )show MRationalFactorize
MRationalFactorize (MRATFAC)

Exports:

factor

-- package MRATFAC MRationalFactorize --

)abbrev package MRATFAC MRationalFactorize
++ Author: P. Gianni
++ Description:
++ MRationalFactorize contains the factor function for multivariate
++ polynomials over the quotient field of a ring R such that the package
++ MultivariateFactorize can factor multivariate polynomials over R.

MRationalFactorize(E,OV,R,P) : C == T

where

E : OrderedAbelianMonoidSup
OV : OrderedSet
R : Join(EuclideanDomain, CharacteristicZero) -- with factor over R[x]
FR ==> Fraction R
P : PolynomialCategory(FR,E,OV)
MPR ==> SparseMultivariatePolynomial(R,OV)
SUP ==> SparseUnivariatePolynomial

C == with

  factor : P -> Factored P
  ++ factor(p) factors the multivariate polynomial p with coefficients
  ++ which are fractions of elements of R.

T == add

IE ==> IndexedExponents OV
PCLFRR ==> PolynomialCategoryLifting(E,OV,FR,P,MPR)
PCLRFR ==> PolynomialCategoryLifting(IE,OV,R,MPR,P)
MFACT ==> MultivariateFactorize(OV,IE,R,MPR)
UPCF2 ==> UnivariatePolynomialCategoryFunctions2

numer1(c:FR): MPR == (numer c) :: MPR
package MFINFACT MultFiniteFactorize

---

MultFiniteFactorize.input ---

)set break resume
)sys rm -f MultFiniteFactorize.output
)spool MultFiniteFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultFiniteFactorize
--R
--R MultFiniteFactorize(OV: OrderedSet,E: OrderedAbelianMonoidSup,F: FiniteFieldCategory,PG: PolynomialCategory)
--R Abbreviation for MultFiniteFactorize is MFINFACT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MFINFACT
--R
--R-------------------------------------------------------- Operations -------------------------------
--R factor : PG -> Factored(PG)
--R factor : SparseUnivariatePolynomial(PG) -> Factored(SparseUnivariatePolynomial(PG))
--R
--E 1

)spool
)lisp (bye)

---

— MultFiniteFactorize.help —

====================================================================
MultFiniteFactorize examples
====================================================================

Package for factorization of multivariate polynomials over finite fields.

See Also:
  o )show MultFiniteFactorize

---

MultFiniteFactorize (MFINFACT)

Exports:
  factor

— package MFINFACT MultFiniteFactorize —

)abbrev package MFINFACT MultFiniteFactorize
++ Author: P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: 19 March 1992
++ Description:
++ Package for factorization of multivariate polynomials over finite fields.

MultFiniteFactorize(OV,E,F,PG) : C == T
where
F : FiniteFieldCategory
OV : OrderedSet
E : OrderedAbelianMonoidSup
PG : PolynomialCategory(F,E,OV)
SUP ==> SparseUnivariatePolynomial
R ==> SUP F
P ==> SparseMultivariatePolynomial(R,OV)
Z ==> Integer
FFPOLY ==> FiniteFieldPolynomialPackage(F)
MParFact ==> Record(irr:P,pow:Z)
MFinalFact ==> Record(contp:R,factors:List MParFact)
SUParFact ==> Record(irr:SUP P,pow:Z)
SUPFinalFact ==> Record(contp:R,factors:List SUParFact)

C == with
factor : PG -> Factored PG
++ factor(p) produces the complete factorization of the multivariate
++ polynomial p over a finite field.
factor : SUP PG -> Factored SUP PG
++ factor(p) produces the complete factorization of the multivariate
++ polynomial p over a finite field. p is represented as a univariate
++ polynomial with multivariate coefficients over a finite field.

T == add
import LeadingCoefDetermination(OV,IndexedExponents OV,R,P)
import MultivariateLifting(IndexedExponents OV,OV,R,P)
import FactoringUtilities(IndexedExponents OV,OV,R,P)
import FactoringUtilities(E,OV,F,PG)
import GenExEuclid(R,SUP R)

NNI ==> NonNegativeInteger
L ==> List
UPCF2 ==> UnivariatePolynomialCategoryFunctions2
LeadFact ==> Record(polfac:L P,correct:R,corrfact:L SUP R)
ContPrim ==> Record(cont:P,prim:P)
ParFact ==> Record(irr:SUP R,pow:Z)
FinalFact ==> Record(contp:R,factors:L ParFact)
NewOrd ==> Record(npol:SUP P,nvar:L OV,newdeg:L NNI)
CHAPTER 14. CHAPTER M

---- Local Functions ----

ran : Z -> R
mFactor : (P,Z) -> MFinalFact
supFactor : (SUP P,Z) -> SUPFinalFact
mfconst : (SUP P,Z,L OV,L NNI) -> L SUP P
mfpol : (SUP P,Z,L OV,L NNI) -> L SUP P
varChoose : (P,L OV,L NNI) -> NewOrd
simplify : (P,Z,L OV,L NNI) -> MFinalFact
intChoose : (SUP P,L OV,R,L P,L L R) -> Valuf
pretest : (P,NNI,L OV,L R) -> FinalFact
checkzero : (SUP P,SUP R) -> Boolean
pushdcoef : PG -> P
pushdown : (PG,OV) -> P
pushupconst : (R,OV) -> PG
pushup : (P,OV) -> PG
norm : L SUP R -> Integer
constantCase : (P,L MParFact) -> MFinalFact
pM : L SUP R -> R
intfact : (SUP P,L OV,L NNI,MFinalFact,L L R) -> L SUP P

basicVar:OV:=NIL$Lisp pretend OV -- variable for the basic step

convertPUP(lfg:MFinalFact): SUPFinalFact ==
[lfg.contp,[[lff.irr ::SUP P,lff.pow]$SUParFact
  for lff in lfg.factors]]$SUPFinalFact

supFactor(um:SUP P,dx:Z) : SUPFinalFact ==
degree(um)=0 => convertPUP(mFactor(ground um,dx))
lvar:L OV:= "setUnion"/[variables cf for cf in coefficients um]
lcont:SUP P
if:L SUP P

flead : SUPFinalFact:=[0,empty()]$SUPFinalFact
factorlist:L SUParFact :=empty()

mdeg :=minimumDegree um ---- is the Mindeg > 0? ----
if mdeg>0 then
  f1:SUP P:=monomial(1,mdeg)
  um:=(um exquo f1)::SUP P
  factorlist:=cons([monomial(1,1),mdeg],factorlist)
  if degree um=0 then return
  lfg:=convertPUP mFactor(ground um, dx)
  [lfg.contp,append(factorlist,lfg.factors)]

om:=map(((p1:P):PG+->pushup(p1,basicVar),um)$UPCF2(P,SUP P,PG,SUP PG)
sqfacs:=squareFree(om)
lcont=
map((p1:PG):P+->pushdown(p1,basicVar),unit sqfacs)_
$UPCF2(PG,SUP PG,P,SUP P)$

----- Factorize the content ----- 
if ground? lcont then
  flead:=convertPUP constantCase(ground lcont,empty())
else
  flead:=supFactor(lcont,dx)
factorlist:=flead.factors

----- Make the polynomial square-free ----- 
sqqfact:=[[map((p:PG):P+->pushdown(p,basicVar),ff.factor),ff.exponent] 
  for ff in factors sqfacs]

--- Factorize the primitive square-free terms ---
for fact in sqqfact repeat
  ffactor:_SUP P:=fact.irr
  ffexp:=fact.pow
  ffcont:=content ffactor
  coefs := coefficients ffactor
  ldeg:= ["max"/[degree(fc,xx) for fc in coefs] for xx in lvar]
  if ground?(leadingCoefficient ffactor) then
    lf:= mfconst(ffactor,dx,lvar,ldeg)
  else lf:=mfpol(ffactor,dx,lvar,ldeg)
  auxfl:=[[lfp,ffexp]$SUParFact for lfp in lf]
  factorlist:=append(factorlist,auxfl)
  lcfacs :=
    */[leadingCoefficient leadingCoefficient(f.irr)**((f.pow)::NNI)
      for f in factorlist]
[(leadingCoefficient leadingCoefficient(um) exquo lcfacs)::R, 
  factorlist]$SUPFinalFact

factor(um:_SUP PG):Factored SUP PG ==
  lv:List OV:=variables um
  ld:=degree(um,lv)
  dx:="min"/ld
  basicVar:=lv.position(dx,ld)
  cm:=map((p1:PG):P++->pushdown(p1,basicVar),um)$UPCF2(PG,SUP PG,P,SUP P)
  flist := supFactor(cm,dx)
  pushupconst(flist.contp,basicVar)::SUP(PG) *
    */[primeFactor(
      map((p1:P):PG++->pushup(p1,basicVar),u.irr)$UPCF2(P,P,SUP PG,SUP PG),
      u.pow) for u in flist.factors])

mFactor(m:P,dx:Z) : MFinalFact ==
  ground?(m) => constantCase(m,empty())
  lvar:L OV:= variables m
  lcont:P
  lf:L SUP P
flead : MFinalFact := [1, empty()]$MFinalFact
factorlist : L MParFact := empty()

--- is the Mindeg > 0? ----
lmdeg := minimumDegree(m, lvar)
or [n > 0 for n in lmdeg] => simplify(m, dx, lvar, lmdeg)

--- Make the polynomial square-free ----
om := pushup(m, basicVar)
sqfacs := squareFree(om)
lcont := pushdown(unit sqfacs, basicVar)

--- Factorize the content ----
if ground? lcont then
  flead := constantCase(lcont, empty())
else
  flead := mFactor(lcont, dx)
factorlist := flead.factors

sqqfact := List Record(factor: P, exponent: Integer)
sqqfact := [[pushdown(ff.factor, basicVar), ff.exponent]
            for ff in factors sqfacs]

--- Factorize the primitive square-free terms ----
for fact in sqqfact repeat
  ffactor := fact.factor
  ffexp := fact.exponent
  ground? ffactor =>
    for lterm in constantCase(ffactor, empty()).factors repeat
      factorlist := cons([lterm.irr, lterm.pow * ffexp], factorlist)
lvar := variables ffactor
x : OV := lvar.1
ldeg := degree(ffactor, lvar)

--- Is the polynomial linear in one of the variables? ---
member?(1, ldeg) =>
x : OV := lvar.position(1, ldeg)
lcont := gcd coefficients(univariate(ffactor, x))
ffactor := (ffactor exquo lcont) : P
factorlist := cons([ffactor, ffexp]$MParFact, factorlist)
for lterm in mFactor(lcont, dx).factors repeat
  factorlist := cons([lterm.irr, lterm.pow * ffexp], factorlist)

varch := varChoose(ffactor, lvar, ldeg)
um := varch.npol

ldeg := ldeg.rest
lvar := lvar.rest
if varch.nvar.1 ^= x then
  lvar := varch.nvar
x := lvar.1
lvar := lvar.rest
pc := gcd coefficients um
if pc ^= 1 then
\[
\text{um} := \text{um exquo pc} :: \text{SUP P}
\]
\[
\text{ffactor} := \text{multivariate(um, x)}
\]
for \text{lcterm in mFactor(pc, dx).factors} repeat
\[
\text{factorlist} := \text{cons([lcterm.irr, lcterm.pow*ffexp], factorlist)}
\]
\[
\text{ldeg} := \text{degree(ffactor, lvar)}
\]

-- should be unitNormal if unified, but for now it is easier
\[
\text{lcum} := \text{leadingCoefficient leadingCoefficient}
\]
\[
\text{lcum} \neq 1 \text{ then}
\]
\[
\text{um} := ((\text{inv lcum}) :: \text{R} :: \text{P}) * \text{um}
\]
\[
\text{flead.contp} := (\text{lcum} :: \text{R}) * \text{flead.contp}
\]

if \text{ground?(leadingCoefficient um)}
\[
\text{then}\text{lf} := \text{mconst(um, dx, lvar, ldeg)}
\]
else \text{lf} := \text{mpol(um, dx, lvar, ldeg)}
\[
\text{auxfl} := \text{[multivariate(lfp, x), ffexp]} \text{MParFact for lfp in lf]
\]
\[
\text{factorlist} := \text{append(factorlist, auxfl)}
\]
\[
\text{flead.factors} := \text{factorlist}
\]

\text{pM(lum:L SUP R) : R ==}
\[
x := \text{monomial(1, 1)}$R$
\]
for \text{i in 1..size()$F$} repeat
\[
p := x + (\text{index(i :: PositiveInteger)}$F$) :: R
\]
\[
\text{testModulus(p, lum)} \Rightarrow \text{return p}
\]
for \text{e in 2..} repeat
\[
p := (\text{createIrreduciblePoly(e :: PositiveInteger)})$\text{FFPOLY}
\]
\[
\text{testModulus(p, lum)} \Rightarrow \text{return p}
\]
while not((q := \text{nextIrreduciblePoly(p)}$\text{FFPOLY}$) case "failed") repeat
\[
p := q :: \text{SUP F}
\]
\[
\text{if testModulus(p, lum)}$\text{GenExEuclid(R, SUP R)} \text{ then return p}
\]

---- push x in the coefficient domain for a term ----
\text{pushdcoef(t:PG):P ==}
\[
\text{map((f1:F):R+ -> coerce(f1)$R, t)}$\text{MPolyCatFunctions2(OV, E,}
\[
\text{IndexedExponents OV, F, R, PG, P)}
\]

---- internal function, for testing bad cases ----
\text{intfact(um: SUP P, lvar: L OV, ldeg: L NNI,}
\[
\text{tleadpol:MFinalFact, ltry: L L R): L SUP P ==}
\]
\text{polcase:Boolean:=(not empty? tleadpol.factors )}
\text{vfchoo:Valuf:=}
\[
\text{polcase} =>
\]
\[
\text{leadpol:L P} := [ff.irr for ff in tleadpol.factors]
\]
\text{intChoose(um, lvar, tleadpol.contp, leadpol, ltry)}
\text{intChoose(um, lvar, 1, empty(), empty())}
\text{unifact:List SUP R := vfchoo.unvfact}
nfact: NNI := #unifact
nfact=1 => [um]
ltry:L L R:= vfchoo.inval
lval:L R:=first ltry
dd:= vfchoo.lu
lpol:List P:=empty()
leadval:List R:=empty()
if polcase then
  leadval := vfchoo.complead
distf := distFact(vfchoo.lu,unifact,tleadpol,lvar,lval)
distf case "failed" =>
  return intfact(um,lvar,ldeg,tleadpol,ltry)
dist := distf :: LeadFact
  -- check the factorization of leading coefficient
lpol:= dist.polfac
dd := dist.correct
unifact:=dist.corrfact
if dd=1 then
  unifact := [dd*unifact.i for i in 1..nfact]
  um := ((dd**(nfact-1)::NNI)::P)*um
(ffff:= lifting(um,lvar,unifact,lval,lpol,ldeg,pM(unifact)))
  case "failed" => intfact(um,lvar,ldeg,tleadpol,ltry)
factfin: L SUP P:=ffff :: L SUP P
if dd=1 then
  factfin:=[primitivePart ff for ff in factfin]
factfin

-- the following functions are used to "push" x in the coefficient ring -
----- push back the variable -----
pushup(f:P,x:OV) :PG ==
ground? f => pushupconst((retract f)@R,x)
rr:PG:=0
while f'=0 repeat
  lf:=leadingMonomial f
cf:=pushupconst(leadingCoefficient f,x)
  lvf:=variables lf
  rr:=rr+monomial(cf,lvf, degree(lf,lvf))$PG
  f:=reductum f
rr

----- push x in the coefficient domain for a polynomial -----
pushdown(g:PG,x:OV) : P ==
ground? g => ((retract g)@F)::R::P
rf:P:=0$P
ug:=univariate(g,x)
while ug=0 repeat
  cf:=monomial(1,degree ug)$R
  rf:=rf+cf*pushdcoef(leadingCoefficient ug)
  ug := reductum ug
rf
push x back from the coefficient domain ----

```
pushupconst(r:R,x:OV):PG ==
ground? r => (retract r)$F ::PG
rr:=0
while r^=0 repeat
  rr:=rr+monomial((leadingCoefficient r)::PG,x,degree r)$PG
  r:=reductum r
rr
```

-- This function has to be added to Euclidean domain

```
ran(k1:Z) : R ==
  --if R case Integer then random()$R rem (2*k1)-k1
  --else
  +/[monomial(random()$F,i)$R for i in 0..k1]
```

checkzero(u:SUP P,um:SUP R) : Boolean ==
  u=0 => um =0
  um = 0 => false
  degree u = degree um => checkzero(reductum u, reductum um)
  false

--- Choose the variable of least degree ---

```
varChoose(m:P,lvar:L OV,ldeg:L NNI) : NewOrd ==
k:="min"/[d for d in ldeg]
k=degree(m,first lvar) =>
  [univariate(m,first lvar),lvar,ldeg]$NewOrd
i:=position(k,ldeg)
x:=lvar.i
ldeg:=cons(k,delete(ldeg,i))
lvar:=cons(x,delete(lvar,i))
[univariate(m,x),lvar,ldeg]$NewOrd
```

```
norm(lum: L SUP R): Integer == "max"/[degree lup for lup in lum]
```

--- Choose the values to reduce to the univariate case ---

```
  -- declarations
degum:NNI := degree um
nvar1:=#lvar
range:NNI:=0
unifact:L SUP R
cf1 : R := 1
testp:Boolean :=
  -- polynomial leading coefficient
  plist = empty() => false
  true
leadcomp,leadcomp1 : L R
leadcomp:=leadcomp1:=empty()
nfatt:NNI := degum+1
```
1 lffc:R:=1
2 lffc1:=lffc
3 newunifact : L SUP R:=empty()
4 leadtest:=true --- the lc test with polCase has to be performed
5 int:L R:=empty()

-- New sets of values are chosen until we find twice the
-- same number of "univariate" factors: the set smaller in modulo is
-- is chosen.
while true repeat
  lval := [ ran(range) for i in 1..nvar1]  
  member?(lval,ltry) => range:=1+range
  ltry := cons(lval,ltry)
  leadcomp1:=[retract eval(pol,lvar,lval) for pol in plist]
  testp and or/[unit? epl for epl in leadcomp1] => range:=range+1
  newm: SUP R:=completeEval(um,lvar,lval)
  degum ^= degree newm or minimumDegree newm ^=0 => range:=range+1
  lffc1:=content newm
  newm:=(newm exquo lffc1)::SUP R
  testp and leadtest and ^ polCase(lffc1*clc,#plist,leadcomp1)
  => range:=range+1
  Dnewm := differentiate newm
  D2newm := map(differentiate, newm)
  degree(gcd [newm,Dnewm,D2newm])^=0 => range:=range+1
  -- if R has Integer then luniv:=henselFact(newm,false)$
  -- else
  lcnm:F:=1
  -- should be unitNormal if unified, but for now it is easier
  if (lcnm:=leadingCoefficient leadingCoefficient newm)^=1 then 
    newm:=((inv lcnm)::R)*newm
  dx:="max"/[degree uc for uc in coefficients newm]
  luniv:=generalTwoFactor(newm)$TwoFactorize(F)
  lunivf:= factors luniv
  nf:= #lunivf
  nf=0 or nf>nfatt => "next values"  --- pretest failed ---

  --- the univariate polynomial is irreducible ---
  if nf=1 then leave (unifact:=[newm])
  lffc1:=lcnm * retract(unit luniv)@R * lffc1

-- the new integer give the same number of factors
  nfatt = nf =>
  -- if this is the first univariate factorization with polCase=true
  -- or if the last factorization has smaller norm and satisfies
  -- polCase
  if leadtest or 
    ((norm unifact > norm [ff.factor for ff in lunivf]) and
    ("testp or polCase(lffc1*clc,#plist,leadcomp1))) then
unifact:=[uf.factor for uf in lunivf]
int:=lval
lffc:=lffc1
if testp then leadcomp:=leadcomp1
leave "foundit"

-- the first univariate factorization, initalize
nfatt > degum =>
unifact:=[uf.factor for uf in lunivf]
lffc:=lffc1
if testp then leadcomp:=leadcomp1
int:=lval
leadtest := false
nfatt := nf

nfatt>nf => -- for the previous values there were more factors
if testp then leadtest:=^polCase(lffc*clc,#plist,leadcomp)
else leadtest:= false
-- if polCase=true we can consider the univariate decomposition
if ^leadtest then
unifact:=[uf.factor for uf in lunivf]
lffc:=lffc1
if testp then leadcomp:=leadcomp1
int:=lval
nfatt := nf
[cons(int,ltry),unifact,lffc,leadcomp]$Valuf

constantCase(m:P,factorlist:List MParFact) : MFinalFact ==
--if R case Integer then [const m,factorlist]$MFinalFact
--else
lunm:=distdfact((retract m)@R,false)$DistinctDegreeFactorize(F,R)
[(lunm.cont)::R, append(factorlist,
[[(pp.irr)::P,pp.pow] for pp in lunm.factors])$MFinalFact

---- The polynomial has mindeg>0 ----

simplify(m:P,dm:Z,lvar:L OV,lmdeg:L NNI):MFinalFact ==
factorlist:L MParFact:=empty() 
poll:P:= 1$P
for x in lvar repeat
i := lmdeg.(position(x,lvar))
i=0 => "next value"
poll:=poll*monomial(1$P,x,i)
factorlist:=cons([x::P,i]$MParFact,factorlist)
m := (m exquo poll)::P
ground? m => constantCase(m,factorlist)
flead:=mFactor(m,dm)
flead.factors:=append(factorlist,flead.factors)
flead
--- m square-free, primitive, lc constant ---

\[
\text{mfconst}(um: \mathbb{P}, dm: \mathbb{Z}, lvar: \mathbb{O}, ldeg: \mathbb{N}) : \mathbb{P} ==
\]

\[
\text{nsign}: \mathbb{B}
\]

\[
\text{factfin}: \mathbb{P} := \text{empty}()
\]

\[
\text{empty?} lvar =>
\]

\[
\text{um1}: \mathbb{R} := \text{map} (\text{ground},
\]

\[
\text{um}) \text{UPCF2}(\mathbb{P}, \mathbb{R}, \mathbb{R})
\]

\[
lum := \text{generalTwoFactor}(\text{um1}) \text{TwoFactorize}(\mathbb{F})
\]

\[
[\text{map} (\text{coerce}, \text{lumf.factor}) \text{UPCF2}(\mathbb{R}, \mathbb{R}, \mathbb{P})
\]

\[
\text{for lumf in factors lum}
\]

\[
\text{intfact}(\text{um}, lvar, ldeg, [0, \text{empty()}]) \text{MFinalFact}, \text{empty}()
\]

--- m is square-free, primitive, lc is a polynomial ---

\[
\text{mfpol}(um: \mathbb{P}, dm: \mathbb{Z}, lvar: \mathbb{O}, ldeg: \mathbb{N}) : \mathbb{P} ==
\]

\[
dist : \text{LeadFact}
\]

\[
\text{tleadpol} := \text{mFactor(leadingCoefficient um, dm)}
\]

\[
\text{intfact} (\text{um}, lvar, ldeg, \text{tleadpol}, \text{empty}())
\]

\[
\text{factor}(m: \mathbb{P}) : \text{Factored PG} ==
\]

\[
\text{lv} := \text{variables m}
\]

\[
\text{lv} = \text{empty}() \Rightarrow \text{makeFR}(m, \text{empty}())
\]

\[
\text{-- reduce to multivariate over SUP}
\]

\[
\text{ld} := [\text{degree}(m, x) \text{ for } x \text{ in } \text{lv}]
\]

\[
\text{dx} := \text{"min"}/\text{ld}
\]

\[
\text{basicVar} := \text{lv(position(dx, ld))}
\]

\[
\text{cm} := \text{pushdown}(m, \text{basicVar})
\]

\[
\text{flist} := \text{mFactor}(\text{cm}, dx)
\]

\[
\text{pushupconst}(\text{flist.contp, basicVar}) *
\]

\[
(*/[\text{primeFactor}(\text{pushup}(u.\text{irr}, \text{basicVar}), u.\text{pow})
\]

\[
\text{for } u \text{ in flist.factors}]
\]


---

--- MFINFACT.dotabb ---

"MFINFACT" [color="#FF4488", href="bookvol10.4.pdf#nameddest=MFINFACT"]

"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]

"MFINFACT" -> "PFECAT"

package MMAP MultipleMap

--- MultipleMap.input ---
```plaintext
)set break resume
)sys rm -f MultipleMap.output
)spool MultipleMap.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultipleMap
--R
--R MultipleMap(R1: IntegralDomain,UP1: UnivariatePolynomialCategory(R1),UPUP1: UnivariatePolynomialCategory(R1),UPUP2: UnivariatePolynomialCategory(Fraction(UP2))) is a package constructor
--R Abbreviation for MultipleMap is MMAP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MMAP
--R
--R--------------------------------------------- Operations -----------------------------------
--R map : ((R1 -> R2),UPUP1) -> UPUP2
--R
--E 1

)spool
)lisp (bye)

----------

-- MultipleMap.help --

=======================================
MultipleMap examples
=======================================

Lifting of a map through 2 levels of polynomials;

See Also:
  o )show MultipleMap

----------
MultipleMap (MMAP)

Exports:
map

— package MMAP MultipleMap —

)abbrev package MMAP MultipleMap
++ Author: Manuel Bronstein
++ Date Created: May 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Lifting of a map through 2 levels of polynomials;

MultipleMap(R1,UP1,UPUP1,R2,UP2,UPUP2): Exports == Implementation where
R1 : IntegralDomain
UP1 : UnivariatePolynomialCategory R1
UPUP1: UnivariatePolynomialCategory Fraction UP1
R2 : IntegralDomain
UP2 : UnivariatePolynomialCategory R2
UPUP2: UnivariatePolynomialCategory Fraction UP2

Q1 ==> Fraction UP1
Q2 ==> Fraction UP2

Exports ==> with
  map: (R1 -> R2, UPUP1) -> UPUP2
  ++ map(f, p) lifts f to the domain of p then applies it to p.

Implementation ==> add
  import UnivariatePolynomialCategoryFunctions2(R1, UP1, R2, UP2)

  rfmap: (R1 -> R2, Q1) -> Q2
  rfmap(f, q) == map(f, numer q) / map(f, denom q)

  map(f, p) ==
map(x -> rfmap(f,x), p)$UnivariatePolynomialCategoryFunctions2(Q1, UPUP1, Q2, UPUP2)


package MCALCFN MultiVariableCalculusFunctions

--- MultiVariableCalculusFunctions.input ---

)set break resume
)sys rm -f MultiVariableCalculusFunctions.output
)spool MultiVariableCalculusFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultiVariableCalculusFunctions

--R MultiVariableCalculusFunctions(S: SetCategory,F: PartialDifferentialRing(S),FLAF: FiniteLinearAggregate(F),FLAS: FiniteLinearAggregate(S) with

--R finiteAggregate is a package constructor
--R Abbreviation for MultiVariableCalculusFunctions is MCALCFN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MCALCFN
--R
--R----------------------------------- Operations -----------------------------------
--R divergence : (FLAF,FLAS) -> F gradient : (F,FLAS) -> Vector(F)
--R hessian : (F,FLAS) -> Matrix(F) jacobian : (FLAF,FLAS) -> Matrix(F)
--R laplacian : (F,FLAS) -> F
--R bandedHessian : (F,FLAS,NonNegativeInteger) -> Matrix(F)
--R bandedJacobian : (FLAF,FLAS,NonNegativeInteger,NonNegativeInteger) -> Matrix(F)
--R
--E 1

)spool
)lisp (bye)
MultiVariableCalculusFunctions (MCALCFN)

Exports:
- bandedHessian
- bandedJacobian
- divergence
- gradient
- hessian
- jacobian
- laplacian

---

Abbreviation for package MCALCFN MultiVariableCalculusFunctions

Author: Themos Tsikas, Grant Keady
Date Created: December 1992
Date Last Updated: June 1993

Description:

\spadtype{MultiVariableCalculusFunctions} Package provides several functions for multivariable calculus.
These include gradient, hessian and jacobian, divergence and laplacian. Various forms for banded and sparse storage of matrices are included.

These include gradient, hessian and jacobian, divergence and laplacian. Various forms for banded and sparse storage of matrices are included.

\begin{verbatim}
MultiVariableCalculusFunctions(S,F,FLAF,FLAS) : Exports == Implementation where
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
S: SetCategory
F: PartialDifferentialRing(S)
FLAS: FiniteLinearAggregate(S)
with finiteAggregate
FLAF: FiniteLinearAggregate(F)

Exports ==> with
  gradient: (F,FLAS) -> Vector F
    \spad{gradient(v,xlist)}
    computes the gradient, the vector of first partial derivatives,
    of the scalar field \emph{v},
    \emph{v} a function of the variables listed in \emph{xlist}.

divergence: (FLAF,FLAS) -> F
    \spad{divergence(vf,xlist)}
    computes the divergence of the vector field \emph{vf},
    \emph{vf} a vector function of the variables listed in \emph{xlist}.

laplacian: (F,FLAS) -> F
    \spad{laplacian(v,xlist)}
    computes the laplacian of the scalar field \emph{v},
    \emph{v} a function of the variables listed in \emph{xlist}.

hessian: (F,FLAS) -> Matrix F
    \spad{hessian(v,xlist)}
    computes the hessian, the matrix of second partial derivatives,
    of the scalar field \emph{v},
    \emph{v} a function of the variables listed in \emph{xlist}.

bandedHessian: (F,FLAS,NNI) -> Matrix F
    \spad{bandedHessian(v,xlist,k)}
    computes the hessian, the matrix of second partial derivatives,
    of the scalar field \emph{v},
    \emph{v} a function of the variables listed in \emph{xlist},
    \emph{k} is the semi-bandwidth, the number of nonzero subdiagonals,
    2*\emph{k}+1 being actual bandwidth.
    Stores the nonzero band in lower triangle in a matrix,
    \emph{k}+1 by \#\emph{xlist},
    whose rows are the vectors formed by diagonal, subdiagonal, etc.
    of the real, full-matrix, hessian.
    (The notation conforms to LAPACK/NAG-F07 conventions.)

-- At one stage it seemed a good idea to help the ASP\<n> domains
-- with the types of their input arguments and this led to the
-- standard Gradient|Hessian|Jacobian functions.
--standardJacobian: (Vector(F),List(S)) -> Matrix F
--    \spad{jacobian(vf,xlist)}
--    computes the jacobian, the matrix of first partial derivatives,
\end{verbatim}
-- ++ of the vector field vf,
-- ++ vf a vector function of the variables listed in xlist.
jacobian: (FLAF,FLAS) -> Matrix F
++ \spad{jacobian(vf,xlist)}
++ computes the jacobian, the matrix of first partial derivatives,
++ of the vector field vf,
++ vf a vector function of the variables listed in xlist.
bandedJacobian: (FLAF,FLAS,NNI,NNI) -> Matrix F
++ \spad{bandedJacobian(vf,xlist,kl,ku)}
++ computes the jacobian, the matrix of first partial derivatives,
++ of the vector field vf,
++ vf a vector function of the variables listed in xlist,
++ kl is the number of nonzero subdiagonals,
++ ku is the number of nonzero superdiagonals,
++ kl+ku+1 being actual bandwidth.
++ Stores the nonzero band in a matrix,
++ dimensions kl+ku+1 by #xlist.
++ The upper triangle is in the top ku rows,
++ the diagonal is in row ku+1,
++ the lower triangle in the last kl rows.
++ Entries in a column in the band store correspond to entries
++ in same column of full store.
++ (The notation conforms to LAPACK/NAG-F07 conventions.)

Implementation ==> add
localGradient(v:F,xlist:List(S)):Vector(F) ==
  vector([D(v,x) for x in xlist])
gradient(v,xflas) ==
  -- xlist:List(S) := [xflas(i) for i in 1 .. maxIndex(xflas)]
  xlist:List(S) := parts(xflas)
  localGradient(v,xlist)
localDivergence(vf:Vector(F),xlist:List(S)):F ==
  i: PI
  n: NNI
  ans: F
  -- Perhaps should report error if two args of min different
  n:= min(#(xlist),((maxIndex(vf))::NNI)$NNI)
  ans:= 0
  for i in 1 .. n repeat ans := ans + D(vf(i),xlist(i))
  ans
divergence(vf,xflas) ==
  xlist:List(S) := parts(xflas)
  i: PI
  n: NNI
  ans: F
  -- Perhaps should report error if two args of min different
  n:= min(#(xlist),((maxIndex(vf))::NNI)$NNI)
  ans:= 0
  for i in 1 .. n repeat ans := ans + D(vf(i),xlist(i))
  ans
laplacian(v,xflas) ==
xlist:List(S) := parts(xflas)
gv:Vector(F) := localGradient(v,xlist)
localDivergence(gv,xlist)
hessian(v,xflas) ==
xlist:List(S) := parts(xflas)
matrix([[D(v,[x,y]) for x in xlist] for y in xlist])
--standardJacobian(vf,xlist) ==
-- i: PI
-- matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
jacobian(vf,xflas) ==
xlist:List(S) := parts(xflas)
i: PI
matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
bandedHessian(v,xflas,k) ==
xlist:List(S) := parts(xflas)
j,iw: PI
n: NNI
bandM: Matrix F
n:= #(xlist)
bandM:= new(k+1,n,0)
for j in 1 .. n repeat setelt(bandM,1,j,D(v,xlist(j),2))
for iw in 2 .. (k+1) repeat (_
for j in 1 .. (n-iw+1) repeat (_
setelt(bandM,iw,j,D(v,[xlist(j),xlist(j+iw-1)])) ) )
bandM
jacobian(vf,xflas) ==
xlist:List(S) := parts(xflas)
i: PI
matrix([[D(vf(i),x) for x in xlist] for i in 1 .. maxIndex(vf)])
bandedJacobian(vf,xflas,kl,ku) ==
xlist:List(S) := parts(xflas)
j,iw: PI
n: NNI
bandM: Matrix F
n:= #(xlist)
bandM:= new(kl+ku+1,n,0)
for j in 1 .. n repeat setelt(bandM,ku+1,j,D(vf(j),xlist(j)))
for iw in (ku+2) .. (ku+kl+1) repeat (_
for j in 1 .. (n-iw+ku+1) repeat (_
setelt(bandM,iw,j,D(vf(j+iw-1-ku),xlist(j)))) )
for iw in 1 .. ku repeat (_
for j in (ku+2-iw) .. n repeat (_
setelt(bandM,iw,j,D(vf(j+iw-1-ku),xlist(j)))) )
bandM
package MULTFACT MultivariateFactorize

-- MultivariateFactorize.input --

)set break resume
)sys rm -f MultivariateFactorize.output
)spool MultivariateFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultivariateFactorize

--R MultivariateFactorize(OV: OrderedSet,E: OrderedAbelianMonoidSup,R: Join(EuclideanDomain,CharacteristicZero),P: PolynomialCategory(R,E,OV)) is a package constructor

--R Abbreviation for MultivariateFactorize is MULTFACT

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MULTFACT

--R

--R----------------------------------- Operations -----------------------------------

--R factor : P -> Factored(P)

--R factor : SparseUnivariatePolynomial(P) -> Factored(SparseUnivariatePolynomial(P))

--E 1

)spool
)lisp (bye)

---

--- MultivariateFactorize.help ---

====================================================================
MultivariateFactorize examples
====================================================================
This is the top level package for doing multivariate factorization over basic domains like Integer or Fraction Integer.

See Also:
o )show MultivariateFactorize

---

MultivariateFactorize (MULTFACT)

Exports:
factor

--- package MULTFACT MultivariateFactorize ---

)abbrev package MULTFACT MultivariateFactorize
++ Author: P. Gianni
++ Date Created: 1983
++ Date Last Updated: Sept. 1990
++ Description:
++ This is the top level package for doing multivariate factorization
++ over basic domains like \spadtype{Integer} or \spadtype{Fraction Integer}.

MultivariateFactorize(OV,E,R,P) : C == T
where
  R : Join(EuclideanDomain, CharacteristicZero)
    -- with factor on R[x]
  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  P : PolynomialCategory(R,E,OV)
  Z ==> Integer
  MParFact ==> Record(irr:P,pow:Z)
  USP ==> SparseUnivariatePolynomial P
SUParFact  ==>  Record(irr:USP,pow:Z)
SUPFinalFact  ==>  Record(contp:R,factors:List SUParFact)
MFinalFact  ==>  Record(contp:R,factors:List MParFact)

--  contp  =  content,
--  factors  =  List of irreducible factors with exponent
L  ==>  List

C == with
  factor  :  P  ->  Factored P
  ++  factor(p)  factors  the  multivariate  polynomial  p  over  its  coefficient
  ++  domain
factor  :  USP  ->  Factored USP
  ++  factor(p)  factors  the  multivariate  polynomial  p  over  its  coefficient
  ++  domain  where  p  is  represented  as  a  univariate  polynomial  with
  ++  multivariate  coefficients
T == add
factor(p:P)  :  Factored P  ==
  R  is  Fraction  Integer  =>
      factor(p)$MRationalFactorize(E,OV,Integer,P)
  R  is  Fraction  Complex  Integer  =>
      factor(p)$MRationalFactorize(E,OV,Complex  Integer,P)
  R  is  Fraction  Polynomial  Integer  and  OV  has  convert:  %  ->  Symbol  =>
      factor(p)$MPolyCatRationalFunctionFactorizer(E,OV,Integer,P)
      factor(p,factor$GenUFactorize(R))$InnerMultFact(OV,E,R,P)

factor(up:USP)  :  Factored USP  ==
  factor(up,factor$GenUFactorize(R))$InnerMultFact(OV,E,R,P)

package  MLIFT  MultivariateLifting

— MultivariateLifting.input —
MultivariateLifting examples

See Also:
  o )show MultivariateLifting

\pagehead{MultivariateLifting}{MLIFT}
\pagepic{ps/v104multivariatelifting.ps}{MLIFT}{1.00}
++ Author: P. Gianni.
++ Description:
++ This package provides the functions for the multivariate "lifting", using
++ an algorithm of Paul Wang.
++ This package will work for every euclidean domain R which has property
++ F, i.e. there exists a factor operation in \texttt{spad(R[x])}.

MultivariateLifting(E, OV, R, P) : C == T
where
OV : OrderedSet
E : OrderedAbelianMonoidSup
R : EuclideanDomain -- with property "F"
Z ==> Integer
BP ==> SparseUnivariatePolynomial R
P : PolynomialCategory(R, E, OV)
SUP ==> SparseUnivariatePolynomial P
NNI ==> NonNegativeInteger
Term ==> Record(expt:NNI, pcoef:P)
VTerm ==> List Term
Table ==> Vector List BP
L ==> List

C == with
corrPoly: (SUP, L OV, L R, L NNI, L SUP, Table, R) -> Union(L SUP, "failed")
++ corrPoly(u, lv, lr, ln, lu, t, r) \undocumented
lifting: (SUP, L OV, L BP, L R, L P, L NNI, R) -> Union(L SUP, "failed")
++ lifting(u, lv, lu, lr, lp, ln, r) \undocumented
lifting1: (SUP, L OV, L SUP, L R, L L P, L VTerm, L NNI, Table, R) ->
Union(L SUP, "failed")
++ lifting1(u,lv, lu, lr, lp, lt, ln, t, r) \undocumented

T == add
GenExEuclid(R, BP)
NPCoef(BP, E, OV, R, P)
IntegerCombinatoricFunctions(Z)

SUPF2 ==> SparseUnivariatePolynomialFunctions2

DetCoef ==> Record(deter:L SUP, dterm:L VTerm, nfacts:L BP,
nlead:L P)

--- local functions ---

normalDerivM : (P, Z, OV) -> P
normalDeriv : (SUP, Z) -> SUP
subslead : (SUP, P) -> SUP
subscoef : (SUP, L Term) -> SUP
maxDegree : (SUP, OV) -> NonNegativeInteger

corrPoly(m: SUP, lvar: L OV, fval: L R, ld: L NNI, flist: L SUP,
table:Table,pmod:R):Union(L SUP,"failed") ==
-- The correction coefficients are evaluated recursively.
-- Extended Euclidean algorithm for the multivariate case.

-- the polynomial is univariate --
#lvar=0 =>
lp:=solveid(map(ground,m)$SUPF2(P,R),pmod,table)
if lp case "failed" then return "failed"
lcoef:= [map(coerce,mp)$SUPF2(R,P) for mp in lp::L BP]
diff,ddiff,pol,polc:SUP
listpoly,listcong:L SUP
deg1:NNI:= ld.first
np:NNI:= #flist
a:P:= fval.first ::P
y:OV:=lvar.first
lvar:=lvar.rest
listpolv:L SUP := [map((p1:P):P +-> eval(p1,y,a),f1) for f1 in flist]
um:=map((p1:P):P ++-> eval(p1,y,a),m)
flcoef:=corrPoly(num,lvar,flvar.rest,ld,rest,listpolv,table,pmod)
if flcoef case "failed" then return "failed"
else lcoef:=flcoef :: L SUP
listcong:=[*/[flist.i for i in 1..np | i^=l] for l in 1..np]
polc:SUP:= (monomial(1,y,1) - a)::SUP
pol := 1$SUP
diff:=m- +/
for l in 1..deg1 repeat
  if diff=0 then return lcoef
  pol := pol*polic
  (ddiff:=map((p:P):P +-> eval(normalDerivM(p,l,y,a),y,a),diff)) = 0
  => "next l"
  fbeta := corrPoly(ddiff,lvar,flvar.rest,ld,rest,listpolv,table,pmod)
  if fbeta case "failed" then return "failed"
  else beta:=fbeta :: L SUP
  lcoef := [lcoef.i+beta.i*polic for i in 1..np]
  diff:=diff- +/[listcong.i*beta.i for i in 1..np]*pol
  lcoef

lifting1(m:SUP,lvar:L OV,plist:L SUP,vlist:L R,tlist:L P,_
  coeflist:L VTerm,listdeg:L NNI,table:Table,pmod:R) :Union(L SUP,"failed") ==
-- The factors of m (multivariate) are determined,
-- We suppose to know the true univariate factors
-- some coefficients are determined
conglist:L SUP:=empty()
nvar : NNI:= #lvar
pol,polc:P
mc,mj:SUP
testp:Boolean:= (not empty?(tlist))
lalpha : L SUP := empty()
tlv: L P:=empty()
subsvar: L OV:=empty()
subsval: L R:=empty()
li: L OV := lvar
ldeg: L NNI:=empty()
clv: L VTerm:=empty()
-- j = #variables, i = #factors
for j in 1..nvar repeat
    x := li.first
    li := rest li
    conglist:= plist
    v := vlist.first
    vlist := rest vlist
    degj := listdeg.j
    ldeg := cons(degj,ldeg)
    subsvar:=cons(x,subsvar)
    subsval:=cons(v,subsval)
    -- substitute the determined coefficients
    if testp then
        if j<nvar then
            tlv:=[eval(p,li,vlist) for p in tlist]
            clv:=[[term.expt,eval(term.pcoef,li,vlist)]$Term
              for term in clist] for clist in coeflist
        else (tlv,clv):=(tlist,coeflist)
        plist :=[subslead(p,lcp) for p in plist for lcp in tlv]
        if not(empty? coeflist) then
            plist:=[subscoef(tpol,clist)
              for tpol in plist for clist in clv]
        mj := map((p1:P):P+->eval(p1,li,vlist),m) --m(x1,..,xj,aj+1,..,an
        polc := x::P - v::P --(xj-aj)
        pol:= 1$P
        -- Construction of Rik, k in 1..right degree for xj+1
        for k in 1..degj repeat -- I can exit before
            pol := pol*polc
            mc := */[term for term in plist]-mj
            if mc=0 then leave "next var"
            -- Modulus Dk
            mc:=map((p1:P):P+-> normalDerivM(p1,k,x),mc)
            (mc := map((p1:P):P+-> eval(p1,[x],[v]),mc))=0 => "next k"
            flalpha:=corrPoly(mc,subsvar.rest,subsval.rest,
              ldeg.rest,conglist,table,pmod)
            if flalpha case "failed" then return "failed"
            else lalpha:=flalpha :: L SUP
            plist:=[term-alpha*pol for term in plist for alpha in lalpha]
            -- PGCD may call with a smaller value of degj
        idegj:Integer:=maxDegree(m,x)
        for term in plist repeat idegj:=idegj -maxDegree(term,x)
        idegj < 0 => return "failed"
        plist
-- There are not extraneous factors

maxDegree(um: SUP, x: OV): NonNegativeInteger ==
    ans: NonNegativeInteger := 0
    while um ^= 0 repeat
        ans := max(ans, degree(leadingCoefficient um, x))
        um := reductum um
    ans

    -- The factors of m (multivariate) are determined, when the
    -- univariate true factors are known and some coefficient determined
    nplist: List SUP := [map(coerce, pp)$SUPF2(R, P) for pp in plist]
    listdet := L SUP := []
    coeflist: L VTerm := []
    if not(empty? tlist) then
        ldcoef : DetCoef := npcoef(um, plist, tlist)
        if not empty?(listdet := ldcoef.deter) then
            if #listdet = #plist then return listdet
            plist := ldcoef.nfacts
            nplist := [map(coerce, pp)$SUPF2(R, P) for pp in plist]
            um := (um exquo */[pol for pol in listdet]):SUP
            tlist := ldcoef.nlead
            coeflist := ldcoef.dterm
        tab := tablePow(degree um, pmod, plist)
        tab case "failed" => error "Table construction failed in MLIFT"
        table: Table := tab
        ffl := lifting1(um, lvar, nplist, vlist, tlist, coeflist, listdeg, tab, pmod)
        if ffl case "failed" then return "failed"
        append(listdet, ffl:: L SUP)

    -- normalDerivM(f, m, x) = the normalized (divided by m!) m-th
    -- derivative with respect to x of the multivariate polynomial f
    normalDerivM(g: P, m: Z, x: OV) : P ==
        multivariate(normalDeriv(univariate(g, x), m), x)

    normalDeriv(f: SUP, m: Z) : SUP ==
        (n1: Z := degree f) < m => 0$SUP
        n1 = m => leadingCoefficient f :: SUP
        k := binomial(n1, m)
        ris := SUP := 0$SUP
        n := n1
        while n >= m repeat
            while n > m repeat
                k := (k*(n1-m)) quo n1
                n1 := n1-1
                ris := ris+monomial(k*leadingCoefficient f, (n-m)::NNI)
            f := reductum f
            n := degree f
ris

subslead(m:SUP,pol:P):SUP ==
  dm:NNI:=degree m
  monomial(pol,dm)+reductum m

subscosf(um:SUP,lterm:L Term):SUP ==
  dm:NNI:=degree um
  new:=monomial(leadingCoefficient um,dm)
  for k in dm-1..0 by -1 repeat
    i:NNI:=k::NNI
    empty?(lterm) or lterm.first.expt^=i =>
      new:=new+monomial(coefficient(um,i),i)
    new:=new+monomial(lterm.first.pcoef,i)
    lterm:=lterm.rest
  new

— MLIFT.dotabb —

"MLIFT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=MLIFT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"MLIFT" -> "PFECAT"

———

package MULTSQFR MultivariateSquareFree

— MultivariateSquareFree.input —

)set break resume
)sys rm -f MultivariateSquareFree.output
)spool MultivariateSquareFree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show MultivariateSquareFree

--R
--R MultivariateSquareFree(E: OrderedAbelianMonoidSup,OV: OrderedSet,R: EuclideanDomain,P: PolynomialCategory(R,E,OV)) is a package constructor
--R Abbreviation for MultivariateSquareFree is MULTSQFR
--R This constructor is not exposed in this frame.
MultivariateSquareFree.help

This package provides the functions for the computation of the square free decomposition of a multivariate polynomial. It uses the package GenExEuclid for the resolution of the equation \(Af + Bg = h\) and its generalization to \(n\) polynomials over an integral domain and the package MultivariateLifting for the "multivariate" lifting.

See Also:
- \( )\)show MultivariateSquareFree
MultivariateSquareFree (MULTSQFR)

Exports:
check    coefChoose  compdegd  consnewpol  intChoose
lift     myDegree   normDeriv2  nsqfree   squareFree
squareFreePrim  univcase

— package MULTSQFR MultivariateSquareFree —

)abbrev package MULTSQFR MultivariateSquareFree
++ Author : P.Gianni
++ Description:
++ This package provides the functions for the computation of the square
++ free decomposition of a multivariate polynomial.
++ It uses the package GenExEuclid for the resolution of
++ the equation \(Af + Bg = h\) and its generalization to \(n\) polynomials
++ over an integral domain and the package \(\)MultivariateLifting\(\)
++ for the "multivariate" lifting.

MultivariateSquareFree (E,OV,R,P) : C == T where
Z  ==> Integer
NNI ==> NonNegativeInteger
R  : EuclideanDomain
OV : OrderedSet
E  : OrderedAbelianMonoidSup
P  : PolynomialCategory(R,E,OV)
SUP ==> SparseUnivariatePolynomial P

BP    ==> SparseUnivariatePolynomial(R)
fUnion ==> Union(\"nil\",\"sqfr\",\"irred\",\"prime\")
ffSUP ==> Record(flg:fUnion,fctr:SUP,xpnt:Integer)
ffP   ==> Record(flg:fUnion,fctr:P,xpnt:Integer)
FFE   ==> Record(factor:BP,exponent:Z)
FEFP  ==> Record(factor:P,exponent:Z)
FFES  ==> Record(factor:SUP,exponent:Z)
Choice ==> Record(upol:BP,Lval:List(R),Lfact:List FFE,ctpol:R)
squareForm ==> Record(unitPart:P,suPart:List FFES)
Twopol ==> Record(pol:SUP,polval:BP)
UPCF2 ==> UnivariatePolynomialCategoryFunctions2

C == with
  squareFree : P -> Factored P
    ++ squareFree(p) computes the square free
    ++ decomposition of a multivariate polynomial p.
  squareFree : SUP -> Factored SUP
    ++ squareFree(p) computes the square free
    ++ decomposition of a multivariate polynomial p presented as
    ++ a univariate polynomial with multivariate coefficients.
  squareFreePrim : P -> Factored P
    ++ squareFreePrim(p) compute the square free decomposition
    ++ of a primitive multivariate polynomial p.

  ---- local functions ----
  compdegd : List FFE -> Z
    ++ compdegd should be local
  univcase : (P,OV) -> Factored(P)
    ++ univcase should be local
  consnewpol : (SUP,BP,Z) -> Twopol
    ++ consnewpol should be local
  nsqfree : (SUP,List(OV),List List R) -> squareForm
    ++ nsqfree should be local
  intChoose : (SUP,List(OV),List List R) -> Choice
    ++ intChoose should be local
  coefChoose : (Z,Factored P) -> P
    ++ coefChoose should be local
  check : (List(FFE),List(FFE)) -> Boolean
    ++ check should be local
  lift : (SUP,BP,BP,P,List(OV),List(NNI),List(R)) -> Union(List(SUP),"failed")
    ++ lift should be local
  myDegree : (SUP,List OV,NNI) -> List NNI
    ++ myDegree should be local
  normDeriv2 : (BP,Z) -> BP
    ++ normDeriv2 should be local

T == add

pmod:R := (prevPrime(2**26)$IntegerPrimesPackage(Integer))::R

import GenExEuclid()
import MultivariateLifting(E,OV,R,P)
import PolynomialGcdPackage(E,OV,R,P)
import FactoringUtilities(E,OV,R,P)
import IntegerCombinatoricFunctions(Z)

---- Are the univariate square-free decompositions consistent? ----

---- new square-free algorithm for primitive polynomial ----
nsqfree(oldf:SUP,lvar:List(OV),ltry:List List R) : squareForm ==
f:=oldf
univPol := intChoose(f,lvar,ltry)
-- debug msg
-- if not empty? ltry then output("ltry =", (ltry::OutputForm))$OutputPackage
f0:=univPol.upol
-- the polynomial is square-free
f0=1$BP => [1$P,[[f,1]$FFES]]$squareForm
lfact:List(FFE):=univPol.Lfact
lval:=univPol.Lval
ctf:=univPol.ctpol
leadpol:Boolean:=false
sqdec:List FFES := empty()
exp0:Z:=0
unitsq:P:=1
lcf:P:=leadingCoefficient f
if ctf"=1 then
  f0:=ctf*f0
  f:=(ctf::P)*f
  lcf:=ctf*lcf
sqlead:List FFEP:= empty()
sqlc:Factored P:=1
if lcf"=1$P then
  leadpol:=true
  sqlc:=squareFree lcf
  unitsq:=unitsq*(unit sqlc)
  sqlead:= factors sqlc
lfact:=sort((z1:FFE,z2:FFE):Boolean +-> z1.exponent > z2.exponent,lfact)
while lfact^=[] repeat
  pfact:=lfact.first
  (g0,exp0):=(pfact.factor,pfact.exponent)
  lfact:=lfact.rest
  lfact[]= and exp0 =1 =>
    f := (f exquo (ctf::P)):SUP
    gg := unitNormal leadingCoefficient f
    sqdec:=cons([gg.associate*f,exp0],sqdec)
    return [gg.unit, sqdec]$squareForm
if ctf"=1 then g0:=ctf*g0
npol:=consnewpol(f,f0,exp0)
(d,d0):=(npol.pol,npol.polval)
if leadpol then lcoef:=coefficientChoose(exp0,sqlc)
  else lcoef:=1$P
ldeg:=myDegree(f,lvar,exp0::NNI)
result := lift(d,g0,(d0 exquo g0)::BP,lcoef,lvar,ldeg,lval)
result case "failed" => return nsqfree(oldf,lvar,ltry)
result0: SUP := (result::List SUP).1
r1: SUP := result0**(exp0: NNI)
if (h:=f exquo r1) case "failed" then return nsqfree(oldf,lvar,empty())
sqdec := cons([result0,exp0],sqdec)
f := h:: SUP
f0 := completeEval(h,lvar,lval)
lcr: P := leadingCoefficient result0
if leadpol and lcr = 1$P then
  for lpfact in sqlead while lcr = 1 repeat
    ground? lcr =>
      unitsq := (unitsq exquo lcr):: P
      lcr := 1$P
    (h1 := lcr exquo lpfact.factor) case "failed" => "next"
    lcr := h1:: P
    lpfact.exponent := (lpfact.exponent) - exp0
    \[((retract f) exquo ctf)::P,sqdec\]$squareForm
squareFree(f: SUP) : Factored SUP ==
  degree f = 0 =>
    fu := squareFree retract f
    makeFR((unit fu):: SUP,[["sqfr", ff.fctr::SUP, ff.xpnt]
      for ff in factorList fu])
lvar := "setUnion" /[variables cf for cf in coefficients f]  
empty? lvar => -- the polynomial is univariate
  upol := map(ground, f)$UPCF2(P, SUP, R, BP)
  usqfr := squareFree upol
  makeFR(map(coerce, unit usqfr)$UPCF2(R, BP, P, SUP),
    [["sqfr", map(coerce, ff.fctr)$UPCF2(R, BP, P, SUP), ff.xpnt]
      for ff in factorList usqfr])
lcf := content f
f := (f exquo lcf) :: SUP
lcSq := squareFree lcf
lfs: List ffSUP := [["sqfr", ff.fctr::SUP, ff.xpnt]
  for ff in factorList lcSq]
partSq := nsqfree(f,lvar,empty())

lfs := append([["sqfr", fu.factor, fu.exponent]$ffSUP
  for fu in partSq.suPart], lfs)
makeFR((unit lcSq * partSq.unitPart) :: SUP, lfs)
squareFree(f: P) : Factored P == 
  ground? f => makeFR(f,[])  --- the polynomial is constant ---
lvar: List(0V) := variables(f)
result1: List ffP := empty()
\begin{verbatim}
lmdeg := minimumDegree(f, lvar)  --- is the mindeg > 0 ? ---
p:=1\times P
for im in 1..#lvar repeat
  (n:=lmdeg.im)=0 => "next im"
  y:=lvar.im
  p:=p\times monomial(1\times P, y, n)
  result1:=cons(["sqfr", y::P, n], result1)
if p^=1\times P then
  f := (f exquo p)::P
if ground? f then return makeFR(f, result1)

lvar:=variables(f)

#lvar=1 =>  --- the polynomial is univariate ---
  result:=univcase(f, lvar.first)
  makeFR(unit result, append(result1, factorList result))

ldeg:=degree(f, lvar)  --- general case ---
m:="min"/[j for j in ldeg[j]=0]
i:1
for j in ldeg while j>m repeat i:=i+1
x:=lvar.i
lvar:=delete(lvar, i)
f0:=univariate (f, x)
lcont:=content f0
nsqfftot:=nsqfree((f0 exquo lcont)::SUP, lvar, empty())
nsqff:=List ffP:=["sqfr",multivariate(fu.factor, x), fu.exponent]$ffP
  for fu in nsqfftot.suPart]
result1:=append(result1, nsqff)
ground? lcont => makeFR(lcont*nsqfftot.unitPart, result1)
sqlead:=squareFree(lcont)
makeFR(unit sqlead*nsqfftot.unitPart, append(result1, factorList sqlead))

-- Choose the integer for the evaluation. --
-- If the polynomial is square-free the function returns upol=1. --

intChoose(f:SUP, lvar:List(OV), ltry:List List R):Choice ==
  degf:= degree f
  try:NNI:=0
  nvr:=#lvar
  range:Z:=10
  lfact1:List(FFE):=[]
  lval1:List R := []
  lfact:List(FFE)
  ctf1:R:=1
  f1:BP:=1\times BP
  d1:Z
  while range<10000000000 repeat
    range:=2\times range
    lval:=[ran(range) for i in 1..nvr]
\end{verbatim}
member?(lval, ltry) => "new integer"
ltry := cons(lval, ltry)
f0 := completeEval(f, lvar, lval)
degree f0 ^= defg => "new integer"
ctf := content f0
lfact : List(FFE) := factors(squareFree((f0 exquo (ctf::R)::BP)::BP))

---- the univariate polynomial is square-free ----
if # lfact = 1 and (lfact.1).exponent = 1 then
  return [1$BP, lval, lfact, 1$R]$Choice
d0 := compdegd lfact
    ---- inizialize lfact1 ----
  try = 0 =>
    f1 := f0
    lfact1 := lfact
ctf1 := ctf
lval1 := lval
d1 := d0
try := 1
d0 = d1 =>
  return [f1, lval1, lfact1, ctf1]$Choice
d0 < d1 =>
  try := 1
  f1 := f0
  lfact1 := lfact
ctf1 := ctf
lval1 := lval
d1 := d0

---- Choose the leading coefficient for the lifting ----
coefChoose(exp: Z, sqlead: Factored(P)) : P ==
  lcoef := unit(sqlead)
  for term in factors(sqlead) repeat
    texp := term. exponent
    texp < exp => "next term"
    texp := texp => lcoef := lcoef * term.factor
    lcoef := lcoef * (term.factor)**(texp quo exp)::NNI
  lcoef

---- Construction of the polynomials for the lifting ----
consnewpol(g: SUP, g0: BP, deg: Z): Twopol ==
  deg = 1 => [g, g0] $Twopol
  deg := deg - 1
  [normalDeriv(g, deg), normDeriv2(g0, deg)] $Twopol

---- lift the univariate square-free factor ----
lift(ud: SUP, g0: BP, g1: BP, lcoef: P, lvar: List(OV),
    ldeg: List(NNI), lval: List(R)) : Union(List SUP, "failed") ==
leadpol:Boolean:=false
lcd:P:=leadingCoefficient ud
leadlist:List(P):=empty()

if ^ground?(leadingCoefficient ud) then
  leadpol:=true
  ud:=lcoef*ud
  lc00:=leadingCoefficient g0
  if ground? lcoef then g0:=retract(lcoef) quo lc00 * g0
     else g0:=(retract(eval(lcoef,lvar,lval)) quo lc00) * g0
  g1:=lc00*g1
  leadlist:=[lcoef,lcd]
plist:=lifting(ud,lvar,[g0,g1],lval,leadlist,ldeg,pmod)
pplist case "failed" => "failed"
(p0:UP,p1:UP):=((plist::List SUP).1,(plist::List SUP).2)
if completeEval(p0,lvar,lval) ^= g0 then (p0,p1):=(p1,p0)
[primitivePart p0,primitivePart p1]

---- the polynomial is univariate ----
univcase(f:P,x:OV) : Factored(P) ==
  uf := univariate f
  cf:=content uf
  uf :=(uf exquo cf)::BP
  result:Factored BP:=squareFree uf
  makeFR(multivariate(cf*unit result,x),
     [["sqfr",multivariate(term.factor,x),term.exponent]
      for term in factors result])

-- squareFreePrim(p:P) : Factored P ==
-- -- p is content free
-- -- ground? p => makeFR(p,[]) --- the polynomial is constant ---
-- -- lvar:List(OV):=variables p
-- -- #lvar=1 => --- the polynomial is univariate ---
-- -- nsqfree(p,lvar.first)
-- nsqfree(p,lvar,1)

compdegd(lfact:List(FFE)) : Z ==
  ris:Z:=0
  for pfact in lfact repeat
    ris:=ris+(pfact.exponent -1)*degree pfact.factor
  ris

normDeriv2(f:BP,m:Z) : BP ==
  (n1:Z:=degree f) < m => 0$BP
  n1=m => (leadingCoefficient f)::BP
  k:=binomial(n1,m)
  ris:BP:=0$BP
  n:Z:=n1
  while n>= m repeat
while n1>n repeat
    k:=(k*(n1-m)) quo n1
    n1:=n1-1
    ris:=ris+monomial(k*leadingCoefficient f,(n-m)::NNI)
    f:=reductum f
    n:=degree f
    ris

myDegree(f:SUP,lvar:List OV,exp:NNI) : List NNI==
[n quo exp for n in degree(f,lvar)]
Chapter 15

Chapter N

package NAGF02 NagEigenPackage

--- NagEigenPackage.input ---

)set break resume
)sys rm -f NagEigenPackage.output
)spool NagEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 164
)show NagEigenPackage
--R
--R NagEigenPackage is a package constructor
--R Abbreviation for NagEigenPackage is NAGF02
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGF02
--R
--R-------------------------------- Operations --------------------------------
--R f02aaf : (Integer,Integer,Matrix(DoubleFloat),Integer) -> Result
--R f02abf : (Matrix(DoubleFloat),Integer,Integer,Integer,Integer) -> Result
--R f02adf : (Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02aef : (Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02aff : (Integer,Integer,Matrix(DoubleFloat),Integer) -> Result
--R f02agf : (Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer) -> Result
--R f02ajf : (Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02akf : (Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02awf : (Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02axf : (Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer,Integer,Integer,Integer,Integer) -> Result
--R f02bbf : (Integer,Integer,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Integer) -> Result

1791
--R f02bjf : (Integer,Integer,Integer,DoubleFloat,Boolean,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02fjf : (Integer,Integer,DoubleFloat,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp27(DOT)),Union(fn: FileName,fp: Asp28(IMAGE))) -> Result
--R f02fjf : (Integer,Integer,DoubleFloat,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp27(DOT)),Union(fn: FileName,fp: Asp28(IMAGE)),FileName) -> Result
--R f02wef : (Integer,Integer,Integer,Integer,Integer,Boolean,Integer,Boolean,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f02xef : (Integer,Integer,Integer,Integer,Integer,Boolean,Integer,Boolean,Integer,Matrix(Complex(DoubleFloat)),Matrix(Complex(DoubleFloat)),Integer) -> Result

)clear all

--S 2 of 164
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 2

--S 3 of 164
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 3

--S 4 of 164
ia:=4
--R
--R
--R (3) 4
--R Type: PositiveInteger
--E 4

--S 5 of 164
n:=4
--R
--R
--R (4) 4
--R Type: PositiveInteger
--E 5

--S 6 of 164
a:Matrix SF:=
[[ 0.5 , 0.0 , 2.3 , -2.6 ],
 [ 0.0 , 0.5 , -1.4 , -0.7 ],
 [ 2.3 , -1.4 , 0.5 , 0.0 ],
 [-2.6 , -0.7 , 0.0 , 0.5 ]]
--R (5)
--R [[0.5,0.0,2.3,0.0,-2.6],[0.0,0.5,-1.4,-0.7,0.0],[2.3,-1.4,0.5,0.0],
--R [-2.6,-0.7,0.0,0.5]]
--R Type: Matrix(DoubleFloat)
--E 6

--S 7 of 164
-- result:=f02aaf(ia,n,a,-1)
--E 7

)clear all

--S 8 of 164
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 8

--S 9 of 164
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 9

--S 10 of 164
a:Matrix SF:=
[ [ 0.5 , 0.0 , 2.3 , -2.6 ],
 [ 0.0 , 0.5 , -1.4 , -0.7 ],
 [ 2.3 , -1.4 , 0.5 , 0.0 ],
 [-2.6 , -0.7 , 0.0 , 0.5 ]]
--R
--R
--R (3)
--R [[0.5,0.0,2.3,0.0,-2.6],[0.0,0.5,-1.4,-0.7,0.0],[2.3,-1.4,0.5,0.0],
--R [-2.6,-0.7,0.0,0.5]]
--R Type: Matrix(DoubleFloat)
--E 10

--S 11 of 164
ia:=4
--R
--R
n:=4

iv:=4

result:=f02abf(a,ia,n,iv,-1)

)clear all

ia:=4
ib:=4
Type: PositiveInteger

n:=4
Type: PositiveInteger

a:Matrix SF:=[
[0.5 , 1.5 , 6.6 , 4.8 ],
[1.5 , 6.5 ,16.2 , 8.6 ],
[6.6 ,16.2 ,37.6 , 9.8 ],
[4.8 , 8.6 , 9.8 ,-17.1 ]]
Type: Matrix(DoubleFloat)

b:Matrix SF:=[
[1 , 3 , 4 , 1 ],
[3 ,13 ,16 ,11 ],
[4 ,16 ,24 ,18 ],
[1 ,11 ,18 ,27 ]]
```
--R  |4. 16. 24. 18.| Type: Matrix(DoubleFloat)
--R   |
--R  +1. 11. 18. 27.+
--E 21

--S 22 of 164
-- result:=f02adf(ia,ib,n,a,b,-1)
--E 22
)
clear all

--S 23 of 164
showArrayValues true
--R
--R
--R (1) true
--R
--E 23

--S 24 of 164
showScalarValues true
--R
--R
--R (2) true
--R
--E 24

--S 25 of 164
ia:=4
--R
--R
--R (3) 4
--R
--E 25

--S 26 of 164
ib:=4
--R
--R
--R (4) 4
--R
--E 26

--S 27 of 164
n:=4
--R
--R
--R (5) 4
--R
```
iv := 4

a := Matrix SF :=
[[0.5, 1.5, 6.6, 4.8],
[1.5, 6.5, 16.2, 8.6],
[6.6, 16.2, 37.6, 9.8],
[4.8, 8.6, 9.8, -17.1]]

Type: Matrix(DoubleFloat)

b := Matrix SF :=
[[1, 3, 4, 1],
[3, 13, 16, 11],
[4, 16, 24, 18],
[1, 11, 18, 27]]

Type: Matrix(DoubleFloat)
--S 31 of 164
-- result:=f02ae5(ia,ib,n,iv,a,b,-1)
--E 31

)clear all

--S 32 of 164
showArrayValues true
--R
--R
--R (1) true
--R
--E 32

--S 33 of 164
showScalarValues true
--R
--R
--R (2) true
--R
--E 33

--S 34 of 164
ia:=4
--R
--R
--R (3) 4
--R
--E 34

--S 35 of 164
n:=4
--R
--R
--R (4) 4
--R
--E 35

--S 36 of 164
a:Matrix SF:=
[[ 1.5, 0.1, 4.5, -1.5 ],
 [-22.5, 3.5, 12.5, -2.5 ],
 [-2.5, 0.3, 4.5, -2.5 ],
 [-2.5, 0.1, 4.5, 2.5 ]]
--R
--R
--R + 1.5 0.0000000000000001 4.5 - 1.5+
--R |
--R | - 22.5 3.5 12.5 - 2.5|
--R (5) |
--R | - 2.5  0.29999999999999999  4.5  - 2.5 |
--R |                                      |
--R + 2.5  0.10000000000000001  4.5   2.5 +
 Type: Matrix(DoubleFloat)
--E 36

--S 37 of 164
-- result:=f02aff(ia,n,a,-1)
--E 37

)clear all

--S 38 of 164
showArrayValues true
--R
--R
--R (1) true
--R
--E 38

 Type: Boolean

--S 39 of 164
showScalarValues true
--R
--R
--R (2) true
--R
--E 39

 Type: Boolean

--S 40 of 164
ia:=4
--R
--R
--R (3) 4
--R
--E 40

 Type: PositiveInteger

--S 41 of 164
n:=4
--R
--R
--R (4) 4
--R
--E 41

 Type: PositiveInteger

--S 42 of 164
ivr:=4
--R
--R
--R (5) 4
--R

 Type: PositiveInteger
ivi:=4

\[
\begin{bmatrix}
1.5 & 0.1 & 4.5 & -1.5 \\
-22.5 & 3.5 & 12.5 & -2.5 \\
-2.5 & 0.3 & 4.5 & -2.5 \\
-2.5 & 0.1 & 4.5 & 2.5
\end{bmatrix}
\]

result:=f02agf(ia,n,ivr,ivi,a,-1)

\(1\) true

\(2\) true
package nageigenpackage

iar := 4

--R (3) 4

Type: PositiveInteger

--E 48

iai := 4

--R (4) 4

Type: PositiveInteger

--E 49

n := 4

--R (5) 4

Type: PositiveInteger

--E 50

ar: Matrix SF :=

\[
\begin{bmatrix}
-21.0 & 0.0 & 13.6 & 0.0 \\
0.0 & 26.0 & 7.5 & 2.5 \\
-2.0 & 1.68 & 4.5 & 1.5 \\
0.0 & -2.6 & -2.7 & 2.5 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

--E 51

ai: Matrix SF :=

\[
\begin{bmatrix}
-5.0 & 24.6 & 10.2 & 4.0 \\
22.5 & -5.0 & -10.0 & 0.0 \\
1.5 & 2.24 & -5.0 & 2.0 \\
-2.5 & 0.0 & 3.6 & -5.0 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

--E 52
```
--R  |
--R  [22.5  - 5.  - 10.  0.]
--R  (7)
--R  |  1.5  2.2400000000000002  - 5.  2.|
--R  |
--R  + 2.5  0.  3.5999999999999996  - 5.+
--R  Type: Matrix(DoubleFloat)

--S 53 of 164
-- result:=f02ajf(iar,iai,n,ar,ai,-1)
-- E 53
)
clear all

--S 54 of 164
showArrayValues true
--R  
--R  
--R  (1) true
--R  Type: Boolean
--E 54

--S 55 of 164
showScalarValues true
--R  
--R  
--R  (2) true
--R  Type: Boolean
--E 55

--S 56 of 164
iar:=4
--R  
--R  
--R  (3) 4
--R  Type: PositiveInteger
--E 56

--S 57 of 164
iai:=4
--R  
--R  
--R  (4) 4
--R  Type: PositiveInteger
--E 57

--S 58 of 164
n:=4
```
PACKAGENAGF02 NAGEIGENPACKAGEN

ivr:=4  ivi:=4  ar:Matrix SF:=
[[-21.0, 0.0, 13.6, 0.0 ],
 [0.0, 26.0, 7.5, 2.5 ],
 [-2.0, 1.68, 4.5, 1.5 ],
 [0.0, -2.6, -2.7, 2.5 ]]

ai:Matrix SF:=
[[-5.0, 24.6, 10.2, 4.0 ],
 [22.5, -5.0, -10.0, 0.0 ],
 [1.5, 2.24, -5.0, 2.0 ],
 [-2.5, 0.0, 3.6, -5.0 ]]
--R \begin{pmatrix} 22.5 & -5. & -10. & 0. \\ 1.5 & 2.2400000000000002 & -5. & 2. \\ +2.5 & 0. & 3.5999999999999996 & 5.+ \end{pmatrix} \text{ Type: Matrix(DoubleFloat)}

--S 63 of 164
-- result:=f02akf(iar,iai,n,ivr,ivi,ar,ai,-1)
-- E 63

)clear all

--S 64 of 164
showArrayValues true
--R
--R
--R (1) true
--R
--E 64

Type: Boolean

--S 65 of 164
showScalarValues true
--R
--R
--R (2) true
--R
--E 65

Type: Boolean

--S 66 of 164
iar:=4
--R
--R
--R (3) 4
--R
--E 66

Type: PositiveInteger

--S 67 of 164
iai:=4
--R
--R
--R (4) 4
--R
--E 67

Type: PositiveInteger

--S 68 of 164
n:=4
--R
--R
ar\text{:} Matrix SF:=
\begin{bmatrix}
0.5 & 0.0 & 1.84 & 2.08 \\
0.0 & 0.5 & 1.12 & -0.56 \\
1.84 & 1.12 & 0.5 & 0.0 \\
2.08 & -0.56 & 0.0 & 0.5 \\
\end{bmatrix}

ai\text{:} Matrix SF:=
\begin{bmatrix}
0.0 & 0.0 & 1.38 & -1.56 \\
0.0 & 0.0 & 0.84 & 0.42 \\
-1.38 & -0.84 & 0.0 & 0.0 \\
1.56 & -0.42 & 0.0 & 0.0 \\
\end{bmatrix}

result:=f02awf(iar,iai,n,ar,ai,-1)
showScalarValues true

(2) true

Type: Boolean

Type: Matrix(DoubleFloat)

(3)

Type: Matrix(DoubleFloat)

(4) 4

Type: PositiveInteger

Type: Matrix(DoubleFloat)
n := 4

ivr := 4

ivi := 4

result := f02axf(ar, iar, aiai, n, ivr, ivi, -1)
\text{ia:=4} \quad \text{Type: PositiveInteger} \\
\text{n:=4} \quad \text{Type: PositiveInteger} \\
\text{alb:=-2.0} \quad \text{Type: Float} \\
\text{ub:=3.0} \quad \text{Type: Float} \\
\text{m:=3} \quad \text{Type: PositiveInteger} \\
\text{iv:=4} \quad \text{Type: PositiveInteger}
a: Matrix SF:=
  [[0.5, 0.0, 2.3, -2.6],
   [0.0, 0.5, -1.4, -0.7],
   [2.3, -1.4, 0.5, 0.0],
   [-2.6, -0.7, 0.0, 0.5]]

result:= f02bbf(ia,n,alb,ub,m,iv,a,-1)

n:=4

ia:=4
--R (4) 4
--R
--E 95

--S 96 of 164
ib:=4
--R
--R
--R (5) 4
--R
--E 96

--S 97 of 164
eps1:SF:=1.0e-4
--R
--R
--R (6) 9.9999999999999991E-5
--R
--E 97

--S 98 of 164
matv:=true
--R
--R
--R (7) true
--R
--E 98

--S 99 of 164
iv:=4
--R
--R
--R (8) 4
--R
--E 99

--S 100 of 164
a:Matrix SF:=

[[3.9 ,12.5,-34.5,-0.5 ],
 [4.3,21.5,-47.5,7.5 ],
 [4.3,21.5,-43.5,3.5 ],
 [4.4,26.0,-46.0,6.0 ]]
--E 100

--S 101 of 164
b:Matrix SF:=
[[1 ,2 ,-3 ,1 ],_
 [1 ,3 ,-5 ,4 ],_
 [1 ,3 ,-4 ,3 ],_
 [1 ,3 ,-4 ,4 ]]
--R

--R  +1. 2. - 3. 1. +
--R  | |  |
--R  |1. 3. - 5. 4.|  |
--R  |(10) | |  |
--R  |1. 3. - 4. 3.|  |
--R  | |  |
--R  +1. 3. - 4. 4. +
--E 101

--S 102 of 164
-- result:=f02bjf(n,ia,ib,eps1,matv,iv,a,b,-1)
-- E 102

)clear all

--S 103 of 164
showArrayValues true
--R
--R
--R (1) true
--R
--E 103

--S 104 of 164
showScalarValues true
--R
--R
--R (2) true
--R
--E 104

--S 105 of 164
n : Integer := 16;
--R
--R
--R
--E 105
--S 106 of 164
k : Integer := 6
--R
--R
--R (4) 6
--R
Type: Integer
--E 106

--S 107 of 164
tol : DoubleFloat := 0.0001
--R
--R
--R (5) 9.999999999999999E-5
--R
Type: DoubleFloat
--E 107

--S 108 of 164
novecs : Integer := 0
--R
--R
--R (6) 0
--R
Type: Integer
--E 108

--S 109 of 164
nrx : Integer := n
--R
--R
--R (7) 16
--R
Type: Integer
--E 109

--S 110 of 164
lwork : Integer := 86
--R
--R
--R (8) 86
--R
Type: Integer
--E 110

--S 111 of 164
lrwork : Integer := 1;
--R
--R
--R
Type: Integer
--E 111

--S 112 of 164
liwork : Integer := 1;
noits: Integer := 1000

m: Integer := 4;

x: Matrix SF := new(nrx, k, 0.0);

ifail: Integer := -1

a: Matrix FRAC INT := new(n, n, 0);

a(1, 1) := 1;
\[ a(1,2) := -\frac{1}{4}; \]
\[ a(1,5) := -\frac{1}{4}; \]
\[ a(i,i-1) := -\frac{1}{4}; \]
\[ a(i,i) := 1; \]
\[ a(i,i+1) := -\frac{1}{4}; \]
\[ a(i,i+4) := -\frac{1}{4}; \]
\[ a(16,16) := 1; \]
a(16,15) := -1/4;
Type: Fraction(Integer)

a(16,12) := -1/4;
Type: Fraction(Integer)

b:Matrix FRAC INT:= new(n,n,0);
Type: Matrix(Fraction(Integer))

b(1,1) := 1
Type: Fraction(Integer)

b(1,2) := -1/2
Type: Fraction(Integer)

for i in 2..n-1 repeat
  b(i,i-1) := -1/2
  b(i,i) := 1
  b(i,i+1) := -1/2
Type: Void
b(16,15) := -1/2

b(16,16) := 1

c : Matrix MachineFloat := (inverse (a))*b;

bb := b :: Matrix MachineFloat
clear all

showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean

m := 5

(3) 5

Type: PositiveInteger

n := 3

(4) 3

Type: PositiveInteger

a : Matrix :=
[[ 2.0, 2.5, 2.5],
 [ 2.0, 2.5, 2.5],
 [ 1.6,-0.4, 2.8],
 [ 2.0,-0.5, 0.5],
 [ 1.2,-0.3,-2.9] ]
lda := m
ncolb := 1
b : Matrix SF:= \[
  [ 1.1, 0.9, 0.6, 0.0, -0.8 ]
\]
ldb := 5
wantq := true
wantp := true

ldq := 1

ldpt := n

ifail := -1

result:=f02wef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldpt,a,b,ifail)

)clear all

showArrayValues true
showScalarValues true
--R
--R
--R (2) true
--R
--E 152

--S 153 of 164
m:=5
--R
--R
--R (3) 5
--R
--E 153

--S 154 of 164
n:=3
--R
--R
--R (4) 3
--R
--E 154

--S 155 of 164
lda:=5
--R
--R
--R (5) 5
--R
--E 155

--S 156 of 164
ncolb:=1
--R
--R
--R (6) 1
--R
--E 156

--S 157 of 164
ldb:=5
--R
--R
--R (7) 5
--R
--E 157

--S 158 of 164
wantq:=true
--R
--R  (8) true
--R             Type: Boolean
--E 158

--S 159 of 164
ldq:=5
--R
--R
--R (9) 5
--R             Type: PositiveInteger
--E 159

--S 160 of 164
wantp:=true
--R
--R
--R (10) true
--R             Type: Boolean
--E 160

--S 161 of 164
ldph:=3
--R
--R
--R (11) 3
--R             Type: PositiveInteger
--E 161

--S 162 of 164

a:Matrix Complex SF:=
[[0.5*%i , -0.5 + 1.5*%i , -1 + %i ],
 [0.4 + 0.3*%i , 0.9 + 1.3*%i , 0.2 + 1.4*%i ],
 [0.4 , -0.4 + 0.4*%i , 1.8 ],
 [0.3 - 0.4*%i , 0.1 + 0.7*%i , 0.0 ],
 [-0.3*%i , 0.3 + 0.3*%i , 2.4*%i ]]
This package uses the NAG Library to compute
* eigenvalues and eigenvectors of a matrix
* eigenvalues and eigenvectors of generalized matrix
* eigenvalue problems
* singular values and singular vectors of a matrix.
Chapter F02
Eigenvalues and Eigenvectors

1. Scope of the Chapter

This chapter is concerned with computing

- eigenvalues and eigenvectors of a matrix
- eigenvalues and eigenvectors of generalized matrix
  eigenvalue problems
- singular values and singular vectors of a matrix.

2. Background to the Problems

2.1. Eigenvalue Problems

In the most usual form of eigenvalue problem we are given a
square n by n matrix $A$ and wish to compute $(\lambda)$ (an
eigenvalue) and $x/\neq 0$ (an eigenvector) which satisfy the equation

$$Ax = (\lambda)x$$

Such problems are called 'standard' eigenvalue problems in
contrast to 'generalized' eigenvalue problems where we wish to
satisfy the equation

$$Ax = (\lambda)Bx$$

$B$ also being a square n by n matrix.

Section 2.1.1 and Section 2.1.2 discuss, respectively, standard
and generalized eigenvalue problems where the matrices involved
are dense; Section 2.1.3 discusses both types of problem in the
case where $A$ and $B$ are sparse (and symmetric).

2.1.1. Standard eigenvalue problems

Some of the routines in this chapter find all the n eigenvalues,
some find all the n eigensolutions (eigenvalues and corresponding
eigenvectors), and some find a selected group of eigenvalues
and/or eigenvectors. The matrix $A$ may be:

(i) general (real or complex)

(ii) real symmetric, or

(iii) complex Hermitian (so that if $a = (\alpha)+i(\beta)$ then $ij$
In all cases the computation starts with a similarity transformation \( S^{-1}A S = T \), where \( S \) is non-singular and is the product of fairly simple matrices, and \( T \) has an 'easier form' than \( A \) so that its eigensolutions are easily determined. The matrices \( A \) and \( T \), of course, have the same eigenvalues, and if \( y \) is an eigenvector of \( T \) then \( Sy \) is the corresponding eigenvector of \( A \).

In case (i) (general real or complex \( A \)), the selected form of \( T \) is an upper Hessenberg matrix (\( t = 0 \) if \( i-j \geq 1 \)) and \( S \) is the product of \( n-2 \) stabilised elementary transformation matrices. There is no easy method of computing selected eigenvalues of a Hessenberg matrix, so that all eigenvalues are always calculated. In the real case this computation is performed via the Francis QR algorithm with double shifts, and in the complex case by means of the LR algorithm. If the eigenvectors are required they are computed by back-substitution following the QR and LR algorithm.

In case (ii) (real and symmetric \( A \)) the selected simple form of \( T \) is a tridiagonal matrix (\( t = 0 \) if \( |i-j| \geq 1 \)), and \( S \) is the product of \( n-2 \) orthogonal Householder transformation matrices. If only selected eigenvalues are required, they are obtained by the method of bisection using the Sturm sequence property, and the corresponding eigenvectors of \( T \) are computed by inverse iteration. If all eigenvalues are required, they are computed from \( T \) via the QL algorithm (an adaptation of the QR algorithm), and the corresponding eigenvectors of \( T \) are the product of the transformations for the QL reduction. In all cases the corresponding eigenvectors of \( A \) are recovered from the computation of \( x = Sy \).

In case (iii) (complex Hermitian \( A \)) analogous transformations as in case (ii) are used. \( T \) has complex elements in off-diagonal positions, but a simple diagonal similarity transformation is then used to produce a real tridiagonal form, after which the QL algorithm and succeeding methods described in the previous paragraph are used to complete the solution.

2.1.2. Generalized eigenvalue problems

Here we distinguish as a special case those problems in which both \( A \) and \( B \) are symmetric and \( B \) is positive-definite and well-conditioned with respect to inversion (i.e., all the eigenvalues of \( B \) are significantly greater than zero). Such problems can be
satisfactorily treated by first reducing them to case (ii) of Section 2.1.1 and then using the methods described there to compute the eigensolutions. If $B$ is factorized as $LL^T$ ($L$ lower triangular), then $Ax=(\lambda)xB$ is equivalent to the standard symmetric problem $Ry=(\lambda)y$, where $R=L^{-1}A(L)$ and $y=L^Tx$. After finding an eigenvector $y$ of $R$, the required $x$ is computed by back-substitution in $y=L^Tx$.

For generalized problems of the form $Ax=(\lambda)Bx$ which do not fall into the special case, the QZ algorithm is provided.

In order to appreciate the domain in which this algorithm is appropriate we remark first that when $B$ is non-singular the problem $Ax=(\lambda)Bx$ is fully equivalent to the problem

$$(B^{-1}A)x=(\lambda)x;$$

both the eigenvalues and eigenvectors being the same. When $A$ is non-singular $Ax=(\lambda)Bx$ is equivalent to

$$(A^{-1}B)x=(\mu)x;$$

the problem $(A^{-1}B)x=(\mu)x$; the eigenvalues $(\mu)$ being the reciprocals of the required eigenvalues and the eigenvectors remaining the same. In theory then, provided at least one of the matrices $A$ and $B$ is non-singular, the generalized problem $Ax=(\lambda)Bx$ could be solved via the standard problem $Cx=(\lambda)x$ with an appropriate matrix $C$, and as far as economy of effort is concerned this is quite satisfactory. However, in practice, for this reduction to be satisfactory from the standpoint of numerical stability, one requires more than the mere non-singularity of $A$ or $B$. It is necessary that $B^{-1}A$ (or $A^{-1}B$) should not only exist but that $B$ (or $A$) should be well-conditioned with respect to inversion. The nearer $B$ (or $A$) is to singularity the more unsatisfactory $B^{-1}A$ (or $A^{-1}B$) will be as a vehicle for determining the required eigenvalues. Unfortunately one cannot counter ill-conditioning in $B$ (or $A$) by computing $B^{-1}A$ (or $A^{-1}B$) accurately to single precision using iterative refinement. Well-determined eigenvalues of the original $Ax=(\lambda)Bx$ may be poorly determined even by the correctly rounded version of $B^{-1}A$ (or $A^{-1}B$). The situation may in some instances be saved by the observation that if $Ax=(\lambda)Bx$ then

$$(A-kB)x=((\lambda)-k)xB.$$ Hence if $A-kB$ is non-singular we may solve the standard problem $[(A-kB)B]x=(\mu)x$ and for numerical
stability we require only that \((A-kB)\) be well-conditioned with respect to inversion.

In practice one may well be in a situation where no \(k\) is known for which \((A-kB)\) is well-conditioned with respect to inversion and indeed \((A-kB)\) may be singular for all \(k\). The QZ algorithm is designed to deal directly with the problem \(Ax=(\lambda)x\) itself and its performance is unaffected by singularity or near-singularity of \(A\), \(B\) or \((A-kB)\).

2.1.3. Sparse symmetric problems

If the matrices \(A\) and \(B\) are large and sparse (i.e., only a small proportion of the elements are non-zero), then the methods described in the previous Section are unsuitable, because in reducing the problem to a simpler form, much of the sparsity of the problem would be lost; hence the computing time and the storage required would be very large. Instead, for symmetric problems, the method of simultaneous iteration may be used to determine selected eigenvalues and the corresponding eigenvectors. The routine provided has been designed to handle both symmetric and generalized symmetric problems.

2.2. Singular Value Problems

The singular value decomposition of an \(m\) by \(n\) real matrix \(A\) is given by

\[
A = QDP^T,
\]

where \(Q\) is an \(m\) by \(m\) orthogonal matrix, \(P\) is an \(n\) by \(n\) orthogonal matrix and \(D\) is an \(m\) by \(n\) diagonal matrix with non-negative diagonal elements. The first \(k=\min(m,n)\) columns of \(Q\) and \(P\) are the left- and right-hand singular vectors of \(A\) and the \(k\) diagonal elements of \(D\) are the singular values.

When \(A\) is complex then the singular value decomposition is given by

\[
A = QDP^H,
\]

\[
A = QDP^H,
\]

where \(Q\) and \(P\) are unitary, \(P^H\) denotes the complex conjugate of \(P\) and \(D\) is as above for the real case.

If the matrix \(A\) has column means of zero, then \(AP\) is the matrix of principal components of \(A\) and the singular values are the square roots of the sample variances of the observations with
Routines are provided to return the singular values and vectors of a general real or complex matrix.

3. Recommendations on Choice and Use of Routines

3.1. General Discussion

There is one routine, F02FJF, which is designed for sparse symmetric eigenvalue problems, either standard or generalized. The remainder of the routines are designed for dense matrices.

3.2. Eigenvalue and Eigenvector Routines

These reduce the matrix $A$ to a simpler form by a similarity transformation $S^{-1}AS=T$ where $T$ is an upper Hessenberg or tridiagonal matrix, compute the eigensolutions of $T$, and then recover the eigenvectors of $A$ via the matrix $S$. The eigenvectors are normalised so that

\[ \sum_{r=1}^{n} \left| x_r \right|^2 = 1 \]

$x_r$ being the $r$th component of the eigenvector $x$, and so that the $r$th element of largest modulus is real if $x$ is complex. For problems of the type $Ax = \lambda Bx$ with $A$ and $B$ symmetric and $B$ positive-definite, the eigenvectors are normalised so that $x^T B x = 1$, $x$ always being real for such problems.

3.3. Singular Value and Singular Vector Routines

These reduce the matrix $A$ to real bidiagonal form, $B$ say, by orthogonal transformations $Q^T A P = B$ in the real case, and by unitary transformations $Q^* A P = B$ in the complex case, and the singular values and vectors are computed via this bidiagonal form. The singular values are returned in descending order.

3.4. Decision Trees

(i) Eigenvalues and Eigenvectors
Please see figure in printed Reference Manual

(ii) Singular Values and Singular Vectors

Please see figure in printed Reference Manual

F02 -- Eigenvalues and Eigenvectors

Chapter F02

Eigenvalues and Eigenvectors

F02AAF All eigenvalues of real symmetric matrix

F02ABF All eigenvalues and eigenvectors of real symmetric matrix

F02ADF All eigenvalues of generalized real symmetric-definite eigenproblem

F02AEF All eigenvalues and eigenvectors of generalized real symmetric-definite eigenproblem

F02AFF All eigenvalues of real matrix

F02AGF All eigenvalues and eigenvectors of real matrix

F02AJF All eigenvalues of complex matrix

F02AKF All eigenvalues and eigenvectors of complex matrix

F02AWF All eigenvalues of complex Hermitian matrix

F02AXF All eigenvalues and eigenvectors of complex Hermitian matrix

F02BBF Selected eigenvalues and eigenvectors of real symmetric matrix

F02BJF All eigenvalues and optionally eigenvectors of generalized eigenproblem by QZ algorithm, real matrices

F02FJF Selected eigenvalues and eigenvectors of sparse symmetric eigenproblem

F02WEF SVD of real matrix

F02XEF SVD of complex matrix
F02 -- Eigenvalue and Eigenvectors
F02AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AAF calculates all the eigenvalues of a real symmetric matrix.

2. Specification

SUBROUTINE F02AAF (A, IA, N, R, E, IFAIL)
INTEGER IA, N, IFAIL
DOUBLE PRECISION A(IA,N), R(N), E(N)

3. Description

This routine reduces the real symmetric matrix A to a real symmetric tridiagonal matrix using Householder’s method. The eigenvalues of the tridiagonal matrix are then determined using the QL algorithm.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the lower triangle of the n by n symmetric matrix A. The elements of the array above the diagonal need not be set. On exit: the elements of A below the diagonal are overwritten, and the rest of the array is unchanged.

2: IA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F02AAF is called.
   Constraint: IA >= N.

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.

4: R(N) -- DOUBLE PRECISION array Output
On exit: the eigenvalues in ascending order.

5: E(N) -- DOUBLE PRECISION array Workspace

6: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   Failure in F02AVF(*) indicating that more than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the eigenvalues depends on the sensitivity of the matrix to rounding errors produced in tridiagonalisation. For a detailed error analysis see Wilkinson and Reinsch [1] pp 222 and 235.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the real symmetric matrix:

\[
\begin{pmatrix}
0.5 & 0.0 & 2.3 & -2.6 \\
0.0 & 0.5 & -1.4 & -0.7 \\
2.3 & -1.4 & 0.5 & 0.0 \\
-2.6 & -0.7 & 0.0 & 0.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02ABF calculates all the eigenvalues and eigenvectors of a real symmetric matrix.

2. Specification

```plaintext
SUBROUTINE F02ABF (A, IA, N, R, V, IV, E, IFAIL)
INTEGER IA, N, IV, IFAIL
DOUBLE PRECISION A(IA,N), R(N), V(IV,N), E(N)
```

3. Description

This routine reduces the real symmetric matrix A to a real symmetric tridiagonal matrix by Householder’s method. The eigenvalues and eigenvectors are calculated using the QL algorithm.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input
   On entry: the lower triangle of the n by n symmetric matrix A. The elements of the array above the diagonal need not be set. See also Section 8.

2: IA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F02ABF is called.
   Constraint: IA >= N.

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.

4: R(N) -- DOUBLE PRECISION array Output
   On exit: the eigenvalues in ascending order.

5: V(IV,N) -- DOUBLE PRECISION array Output
   On exit: the normalised eigenvectors, stored by columns; the ith column corresponds to the ith eigenvalue. The
eigenvectors are normalised so that the sum of squares of the elements is equal to 1.

6: IV -- INTEGER Input
On entry:
the first dimension of the array V as declared in the (sub)program from which F02ABF is called.
Constraint: IV >= N.

7: E(N) -- DOUBLE PRECISION array Workspace

8: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
Failure in F02AMF(*) indicating that more than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy
The eigenvectors are always accurately orthogonal but the accuracy of the individual eigenvectors is dependent on their inherent sensitivity to changes in the original matrix. For a detailed error analysis see Wilkinson and Reinsch [1] pp 222 and 235.

8. Further Comments
The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your implementation, the routine may be called with the same actual array supplied for parameters A and V, in which case the eigenvectors will overwrite the original matrix. However this is not standard Fortran 77, and may not work on all systems.

9. Example
To calculate all the eigenvalues and eigenvectors of the real symmetric matrix:
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F02 -- Eigenvalue and Eigenvectors
F02ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02ADF calculates all the eigenvalues of $ Ax = (\lambda) B x $, where $ A $ is a real symmetric matrix and $ B $ is a real symmetric positive-definite matrix.

2. Specification

```fortran
SUBROUTINE F02ADF (A, IA, B, IB, N, R, DE, IFAIL)
INTEGER IA, IB, N, IFAIL
DOUBLE PRECISION A(IA,N), B(IB,N), R(N), DE(N)
```

3. Description

The problem is reduced to the standard symmetric eigenproblem using Cholesky's method to decompose $ B $ into triangular matrices, $ T $ $ B = L L^T $, where $ L $ is lower triangular. Then $ Ax = (\lambda) B x $ implies $ -T -1 T $ $ T $ $ (L A L^T) (L x) = (\lambda) (L x) $; hence the eigenvalues of $ Ax = (\lambda) B x $ are those of $ Py = (\lambda) y $ where $ P $ is the symmetric matrix $ L A L^T $. Householder's method is used to tridiagonalise the matrix $ P $ and the eigenvalues are then found using the QL algorithm.

4. References

5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the upper triangle of the n by n symmetric matrix
   A. The elements of the array below the diagonal need not be
   set. On exit: the lower triangle of the array is
   overwritten. The rest of the array is unchanged.

2: IA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the
   (sub)program from which F02ADF is called.
   Constraint: IA >= N.

3: B(IB,N) -- DOUBLE PRECISION array Input/Output
   On entry: the upper triangle of the n by n symmetric
   positive-definite matrix B. The elements of the array below
   the diagonal need not be set. On exit: the elements below
   the diagonal are overwritten. The rest of the array is
   unchanged.

4: IB -- INTEGER Input
   On entry: the first dimension of the array B as declared in the
   (sub)program from which F02ADF is called.
   Constraint: IB >= N.

5: N -- INTEGER Input
   On entry: n, the order of the matrices A and B.

6: R(N) -- DOUBLE PRECISION array Output
   On exit: the eigenvalues in ascending order.

7: DE(N) -- DOUBLE PRECISION array Workspace

8: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   Failure in F01AEF(*); the matrix B is not positive-definite.
possibly due to rounding errors.

IFAIL= 2
Failure in F02AVF(*), more than 30*N iterations are required
to isolate all the eigenvalues.

7. Accuracy

In general this routine is very accurate. However, if B is ill-
conditioned with respect to inversion, the eigenvalues could be
inaccurately determined. For a detailed error analysis see

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the general symmetric
eigenproblem Ax=(lambda) Bx where A is the symmetric matrix:

\[
\begin{pmatrix}
0.5 & 1.5 & 6.6 & 4.8 \\
1.5 & 6.5 & 16.2 & 8.6 \\
6.6 & 16.2 & 37.6 & 9.8 \\
4.8 & 8.6 & 9.8 & -17.1
\end{pmatrix}
\]

and B is the symmetric positive-definite matrix:

\[
\begin{pmatrix}
1 & 3 & 4 & 1 \\
3 & 13 & 16 & 11 \\
4 & 16 & 24 & 18 \\
1 & 11 & 18 & 27
\end{pmatrix}
\]

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
F02AEF calculates all the eigenvalues and eigenvectors of $Ax=(\lambda)x$, where $A$ is a real symmetric matrix and $B$ is a real symmetric positive-definite matrix.

2. Specification

```
SUBROUTINE F02AEF (A, IA, B, IB, N, R, V, IV, DL, E, IFAIL)
INTEGER IA, IB, N, IV, IFAIL
DOUBLE PRECISION A(IA,N), B(IB,N), R(N), V(IV,N), DL(N), E
1 (N)
```

3. Description

The problem is reduced to the standard symmetric eigenproblem using Cholesky's method to decompose $B$ into triangular matrices $T$

$$B=LL^T,$$

where $L$ is lower triangular. Then $Ax=(\lambda)x$ implies $-1 -T T$

$$(L AL^{-1})x=(\lambda)(L x);$$ hence the eigenvalues of $Ax=(\lambda)x$ are those of $Py=(\lambda)y$, where $P$ is the symmetric matrix $L AL^{-1}$.

Householder's method is used to tridiagonalise the matrix $P$ and the eigenvalues are found using the QL algorithm. An eigenvector $z$ of the derived problem is related to an eigenvector $x$ of the original problem by $z=L x$. The eigenvectors $z$ are determined using the QL algorithm and are normalised so that $z^T z=1$; the eigenvectors of the original problem are then determined by solving $L x=z$, and are normalised so that $x^T B x=1$.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output

On entry: the upper triangle of the n by n symmetric matrix $A$. The elements of the array below the diagonal need not be set. On exit: the lower triangle of the array is overwritten. The rest of the array is unchanged. See also Section 8.
2: IA -- INTEGER
   On entry: the first dimension of the array A as declared in the (sub)program from which F02AEF is called.
   Constraint: IA >= N.

3: B(IB,N) -- DOUBLE PRECISION array
   On entry: the upper triangle of the n by n symmetric positive-definite matrix B. The elements of the array below the diagonal need not be set. On exit: the elements below the diagonal are overwritten. The rest of the array is unchanged.

4: IB -- INTEGER
   On entry: the first dimension of the array B as declared in the (sub)program from which F02AEF is called.
   Constraint: IB >= N.

5: N -- INTEGER
   On entry: n, the order of the matrices A and B.

6: R(N) -- DOUBLE PRECISION array
   On exit: the eigenvalues in ascending order.

7: V(IV,N) -- DOUBLE PRECISION array
   On exit: the normalised eigenvectors, stored by columns; the ith column corresponds to the ith eigenvalue. The T eigenvectors x are normalised so that x Bx=1. See also Section 8.

8: IV -- INTEGER
   On entry: the first dimension of the array V as declared in the (sub)program from which F02AEF is called.
   Constraint: IV >= N.

9: DL(N) -- DOUBLE PRECISION array
10: E(N) -- DOUBLE PRECISION array
   Workspace

11: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
Failure in F01AEF(*); the matrix B is not positive-definite, possibly due to rounding errors.

IFAIL= 2
Failure in F02AMF(*); more than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

In general this routine is very accurate. However, if B is ill-conditioned with respect to inversion, the eigenvectors could be inaccurately determined. For a detailed error analysis see Wilkinson and Reinsch [1] pp 310, 222 and 235.

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your implementation, the routine may be called with the same actual array supplied for parameters A and V, in which case the eigenvectors will overwrite the original matrix A. However this is not standard Fortran 77, and may not work on all systems.

9. Example

To calculate all the eigenvalues and eigenvectors of the general symmetric eigenproblem Ax=(lambda) Bx where A is the symmetric matrix:

\[
\begin{pmatrix}
0.5 & 1.5 & 6.6 & 4.8 \\
1.5 & 6.5 & 16.2 & 8.6 \\
6.6 & 16.2 & 37.6 & 9.8 \\
4.8 & 8.6 & 9.8 & -17.1
\end{pmatrix}
\]

and B is the symmetric positive-definite matrix:

\[
\begin{pmatrix}
1 & 3 & 4 & 1 \\
3 & 13 & 16 & 11 \\
4 & 16 & 24 & 18 \\
1 & 11 & 18 & 27
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F02 -- Eigenvalue and Eigenvectors

F02AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AFF calculates all the eigenvalues of a real unsymmetric matrix.

2. Specification

SUBROUTINE F02AFF (A, IA, N, RR, RI, INTGER, IFAIL)
INTEGER IA, N, INTGER(N), IFAIL
DOUBLE PRECISION A(IA,N), RR(N), RI(N)

3. Description

The matrix A is first balanced and then reduced to upper Hessenberg form using stabilised elementary similarity transformations. The eigenvalues are then found using the QR algorithm for real Hessenberg matrices.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the n by n matrix A. On exit: the array is overwritten.

2: IA -- INTEGER Input
   On entry: the dimension of the array A as declared in the (sub)program from which F02AFF is called.
   Constraint: IA >= N.

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.

4: RR(N) -- DOUBLE PRECISION array Output
On exit: the real parts of the eigenvalues.

5: RI(N) -- DOUBLE PRECISION array
   On exit: the imaginary parts of the eigenvalues.

6: INTEGER(N) -- INTEGER array
   On exit: INTEGER(i) contains the number of iterations used
   to find the ith eigenvalue. If INTEGER(i) is negative, the i
   th eigenvalue is the second of a pair found simultaneously.

Note that the eigenvalues are found in reverse order,
starting with the nth.

7: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   More than 30*N iterations are required to isolate all the
   eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and
the multiplicity of the roots. For a detailed error analysis see

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the real matrix:

\[
\begin{pmatrix}
1.5 & 0.1 & 4.5 & -1.5 \\
-22.5 & 3.5 & 12.5 & -2.5 \\
-2.5 & 0.3 & 4.5 & -2.5 \\
-2.5 & 0.1 & 4.5 & 2.5 \\
\end{pmatrix}
\]

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors
F02AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AGF calculates all the eigenvalues and eigenvectors of a real unsymmetric matrix.

2. Specification

SUBROUTINE F02AGF (A, IA, N, RR, RI, VR, IVR, VI, IVI, INTGER, IFAIL)
INTEGER IA, N, IVR, IVI, INTGER(N), IFAIL
DOUBLE PRECISION A(IA,N), RR(N), RI(N), VR(IVR,N), VI (IVI,N)

3. Description

The matrix A is first balanced and then reduced to upper Hessenberg form using real stabilised elementary similarity transformations. The eigenvalues and eigenvectors of the Hessenberg matrix are calculated using the QR algorithm. The eigenvectors of the Hessenberg matrix are back-transformed to give the eigenvectors of the original matrix A.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
          On entry: the n by n matrix A. On exit: the array is overwritten.

2: IA -- INTEGER Input
          On entry: the first dimension of the array A as declared in the (sub)program from which F02AGF is called.
3: N -- INTEGER
   On entry: n, the order of the matrix A.

4: RR(N) -- DOUBLE PRECISION array
   On exit: the real parts of the eigenvalues.

5: RI(N) -- DOUBLE PRECISION array
   On exit: the imaginary parts of the eigenvalues.

6: VR(IVR,N) -- DOUBLE PRECISION array
   On exit: the real parts of the eigenvectors, stored by columns. The ith column corresponds to the ith eigenvalue. The eigenvectors are normalised so that the sum of the squares of the moduli of the elements is equal to 1 and the element of largest modulus is real. This ensures that real eigenvalues have real eigenvectors.

7: IVR -- INTEGER
   On entry: the first dimension of the array VR as declared in the (sub)program from which F02AGF is called.
   Constraint: IVR >= N.

8: VI(IVI,N) -- DOUBLE PRECISION array
   On exit: the imaginary parts of the eigenvectors, stored by columns. The ith column corresponds to the ith eigenvalue.

9: IVI -- INTEGER
   On entry: the first dimension of the array VI as declared in the (sub)program from which F02AGF is called.
   Constraint: IVI >= N.

10: INTGER(N) -- INTEGER array
    On exit: INTGER(i) contains the number of iterations used to find the ith eigenvalue. If INTGER(i) is negative, the ith eigenvalue is the second of a pair found simultaneously.

   Note that the eigenvalues are found in reverse order, starting with the nth.

11: IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1

More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 352 and 390.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues and eigenvectors of the real matrix:

\[
\begin{pmatrix}
1.5 & 0.1 & 4.5 & -1.5 \\
-22.5 & 3.5 & 12.5 & -2.5 \\
-2.5 & 0.3 & 4.5 & -2.5 \\
-2.5 & 0.1 & 4.5 & 2.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors
F02AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AJF calculates all the eigenvalues of a complex matrix.

2. Specification
SUBROUTINE F02AJF (AR, IAR, AI, IAI, N, RR, RI, INTGER,
1     INTEGER IAR, IAI, N, INTGER(N), IFAIL
DOUBLE PRECISION AR(IAR,N), AI(IAI,N), RR(N), RI(N)

3. Description

The complex matrix $A$ is first balanced and then reduced to upper
Hessenberg form using stabilised elementary similarity
transformations. The eigenvalues are then found using the
modified LR algorithm for complex Hessenberg matrices.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input/Output
   On entry: the real parts of the elements of the $n$ by $n$
   complex matrix $A$. On exit: the array is overwritten.

2: IAR -- INTEGER Input
   On entry: the first dimension of the array AR as declared in the
   (sub)program from which F02AJF is called.
   Constraint: IAR $\geq N$.

3: AI(IAI,N) -- DOUBLE PRECISION array Input/Output
   On entry: the imaginary parts of the elements of the $n$ by $n$
   complex matrix $A$. On exit: the array is overwritten.

4: IAI -- INTEGER Input
   On entry: the first dimension of the array AI as declared in the
   (sub)program from which F02AJF is called.
   Constraint: IAI $\geq N$.

5: N -- INTEGER Input
   On entry: $n$, the order of the matrix $A$.

6: RR(N) -- DOUBLE PRECISION array Output
   On exit: the real parts of the eigenvalues.

7: RI(N) -- DOUBLE PRECISION array Output
   On exit: the imaginary parts of the eigenvalues.

8: INTGER(N) -- INTEGER array Workspace
9: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy

The accuracy of the results depends on the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 352 and 401.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the complex matrix:

\[
\begin{pmatrix}
-21.0-5.0i & 24.60i & 13.6+10.2i & 4.0i \\
22.5i & 26.00-5.00i & 7.5-10.0i & 2.5 \\
-2.0+1.5i & 1.68+2.24i & 4.5-5.0i & 1.5+2.0i \\
-2.5i & -2.60 & -2.7+3.6i & 2.5-5.0i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F02 -- Eigenvalue and Eigenvectors

F02AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

F02AKF calculates all the eigenvalues and eigenvectors of a complex matrix.

2. Specification

```plaintext
SUBROUTINE F02AKF (AR, IAR, AI, IAI, N, RR, RI, VR, IVR,
   1       VI, IVI, INTEGER, IFAIL)
   INTEGER IAR, IAI, N, IVR, IVI, INTEGER(N), IFAIL
   DOUBLE PRECISION AR(IAR,N), AI(IAI,N), RR(N), RI(N), VR
   1       (IVR,N), VI(IVI,N)
```

3. Description

The complex matrix $A$ is first balanced and then reduced to upper Hessenberg form by stabilised elementary similarity transformations. The eigenvalues and eigenvectors of the Hessenberg matrix are calculated using the LR algorithm. The eigenvectors of the Hessenberg matrix are back-transformed to give the eigenvectors of the original matrix.

4. References


5. Parameters

1: $AR(IAR,N)$ -- DOUBLE PRECISION array Input/Output
   On entry: the real parts of the elements of the $n$ by $n$ complex matrix $A$. On exit: the array is overwritten.

2: $IAR$ -- INTEGER Input
   On entry: the first dimension of the array $AR$ as declared in the (sub)program from which F02AKF is called.
   Constraint: $IAR >= N$.

3: $AI(IAI,N)$ -- DOUBLE PRECISION array Input/Output
   On entry: the imaginary parts of the elements of the $n$ by $n$ complex matrix $A$. On exit: the array is overwritten.

4: $IAI$ -- INTEGER Input
   On entry: the first dimension of the array $AI$ as declared in the (sub)program from which F02AKF is called.
   Constraint: $IAI >= N$.

5: $N$ -- INTEGER Input
On entry: n, the order of the matrix A.

6: RR(N) -- DOUBLE PRECISION array Output
On exit: the real parts of the eigenvalues.

7: RI(N) -- DOUBLE PRECISION array Output
On exit: the imaginary parts of the eigenvalues.

8: VR(IVR,N) -- DOUBLE PRECISION array Output
On exit: the real parts of the eigenvectors, stored by columns. The ith column corresponds to the ith eigenvalue.
The eigenvectors are normalised so that the sum of squares of the moduli of the elements is equal to 1 and the element of largest modulus is real.

9: IVR -- INTEGER Input
On entry:
the first dimension of the array VR as declared in the (sub)program from which F02AKF is called.
Constraint: IVR >= N.

10: VI(IVI,N) -- DOUBLE PRECISION array Output
On exit: the imaginary parts of the eigenvectors, stored by columns. The ith column corresponds to the ith eigenvalue.

11: IVI -- INTEGER Input
On entry:
the first dimension of the array VI as declared in the (sub)program from which F02AKF is called.
Constraint: IVI >= N.

12: INTEGER(N) -- INTEGER array Workspace

13: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy
The accuracy of the results depends on the conditioning of the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 352 and 390.

8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues and eigenvectors of the complex matrix:

\[
\begin{pmatrix}
-21.0 - 5.0i & 24.60i & 13.6 + 10.2i & 4.0i \\
22.5i & 26.00 - 5.00i & 7.5 - 10.0i & 2.5 \\
-2.0 + 1.5i & 1.68 + 2.24i & 4.5 - 5.0i & 1.5 + 2.0i \\
-2.5i & -2.60 & -2.7 + 3.6i & 2.5 - 5.0i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
F02 -- Eigenvalue and Eigenvectors
F02AWF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02AWF calculates all the eigenvalues of a complex Hermitian matrix.

2. Specification

```
SUBROUTINE F02AWF (AR, IAR, AI, IAI, N, R, WK1, WK2, WK3, IFAIL)
INTEGER IAR, IAI, N, IFAIL
DOUBLE PRECISION AR(IAR,N), AI(IAI,N), R(N), WK1(N), WK2(N), WK3(N)
```

3. Description

The complex Hermitian matrix A is first reduced to a real
tridiagonal matrix by n-2 unitary transformations, and a subsequent diagonal transformation. The eigenvalues are then derived using the QL algorithm, an adaptation of the QR algorithm.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input/Output
   On entry: the real parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. On exit: the array is overwritten.

2: IAR -- INTEGER Input
   On entry: the first dimension of the array AR as declared in the (sub)program from which F02AWF is called.
   Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array Input/Output
   On entry: the imaginary parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. On exit: the array is overwritten.

4: IAI -- INTEGER Input
   On entry: the first dimension of the array AI as declared in the (sub)program from which F02AWF is called.
   Constraint: IAI >= N.

5: N -- INTEGER Input
   On entry: n, the order of the complex Hermitian matrix, A.

6: R(N) -- DOUBLE PRECISION array Output
   On exit: the eigenvalues in ascending order.

7: WK1(N) -- DOUBLE PRECISION array Workspace

8: WK2(N) -- DOUBLE PRECISION array Workspace

9: WK3(N) -- DOUBLE PRECISION array Workspace
10: IFAIL -- INTEGER  
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
More than 30*N iterations are required to isolate all the eigenvalues.

7. Accuracy


8. Further Comments

The time taken by the routine is approximately proportional to n

9. Example

To calculate all the eigenvalues of the complex Hermitian matrix:

\[
\begin{pmatrix}
0.50 & 0.00 & 1.84+1.38i & 2.08-1.56i \\
0.00 & 0.50 & 1.12+0.84i & -0.56+0.42i \\
1.84-1.38i & 1.12-0.84i & 0.50 & 0.00 \\
2.08+1.56i & -0.56-0.42i & 0.00 & 0.50 \\
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

F02AXF calculates all the eigenvalues and eigenvectors of a complex Hermitian matrix.

2. Specification

```fortran
SUBROUTINE F02AXF (AR, IAR, AI, IAI, N, R, VR, IVR, VI,
1      IVI, WK1, WK2, WK3, IFAIL)
INTEGER IAR, IAI, N, IVR, IVI, IFAIL
DOUBLE PRECISION AR(IAR,N), AI(IAI,N), R(N), VR(IVR,N), VI
1      (IVI,N), WK1(N), WK2(N), WK3(N)
```

3. Description

The complex Hermitian matrix is first reduced to a real tridiagonal matrix by n-2 unitary transformations and a subsequent diagonal transformation. The eigenvalues and eigenvectors are then derived using the QL algorithm, an adaptation of the QR algorithm.

4. References


5. Parameters

1: AR(IAR,N) -- DOUBLE PRECISION array Input
   On entry: the real parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. See also Section 8.

2: IAR -- INTEGER Input
   On entry: the first dimension of the array AR as declared in the (sub)program from which F02AXF is called.
   Constraint: IAR >= N.

3: AI(IAI,N) -- DOUBLE PRECISION array Input
   On entry: the imaginary parts of the elements of the lower triangle of the n by n complex Hermitian matrix A. Elements of the array above the diagonal need not be set. See also Section 8.

4: IAI -- INTEGER Input
On entry:
the first dimension of the array AI as declared in the
(sub)program from which F02AXF is called.
Constraint: IAI >= N.

5:  N -- INTEGER
    On entry: n, the order of the matrix, A.

6:  R(N) -- DOUBLE PRECISION array
    On exit: the eigenvalues in ascending order.

7:  VR(IVR,N) -- DOUBLE PRECISION array
    On exit: the real parts of the eigenvectors, stored by
    columns. The ith column corresponds to the ith eigenvector.
    The eigenvectors are normalised so that the sum of the
    squares of the moduli of the elements is equal to 1 and the
    element of largest modulus is real. See also Section 8.

8:  IVR -- INTEGER
    On entry:
    the first dimension of the array VR as declared in the
    (sub)program from which F02AXF is called.
    Constraint: IVR >= N.

9:  VI(IVI,N) -- DOUBLE PRECISION array
    On exit: the imaginary parts of the eigenvectors, stored by
    columns. The ith column corresponds to the ith eigenvector.
    See also Section 8.

10: IVI -- INTEGER
    On entry:
    the first dimension of the array VI as declared in the
    (sub)program from which F02AXF is called.
    Constraint: IVI >= N.

11: WK1(N) -- DOUBLE PRECISION array
    Workspace

12: WK2(N) -- DOUBLE PRECISION array
    Workspace

13: WK3(N) -- DOUBLE PRECISION array
    Workspace

14: IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
More than 30*N iterations are required to isolate all the eigenvalues.

IFAIL= 2
The diagonal elements of AI are not all zero, i.e., the complex matrix is not Hermitian.

7. Accuracy

The eigenvectors are always accurately orthogonal but the accuracy of the individual eigenvalues and eigenvectors is dependent on their inherent sensitivity to small changes in the original matrix. For a detailed error analysis see Peters [1] page 3 and [2] page 3.

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the implementation document, the routine may be called with the same actual array supplied for parameters AR and VR, and for AI and VI, in which case the eigenvectors will overwrite the original matrix A. However this is not standard Fortran 77, and may not work on all systems.

9. Example

To calculate the eigenvalues and eigenvectors of the complex Hermitian matrix:

\[
\begin{bmatrix}
0.50 & 0.00 & 1.84+1.38i & 2.08-1.56i \\
0.00 & 0.50 & 1.12+0.84i & -0.56+0.42i \\
1.84-1.38i & 1.12-0.84i & 0.50 & 0.00 \\
2.08+1.56i & -0.56-0.42i & 0.00 & 0.50
\end{bmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F02BBF calculates selected eigenvalues and eigenvectors of a real symmetric matrix by reduction to tridiagonal form, bisection and inverse iteration, where the selected eigenvalues lie within a given interval.

2. Specification

```fortran
SUBROUTINE F02BBF (A, IA, N, ALB, UB, M, MM, R, IV, D, E, E2, X, G, C, ICOUNT, IFAIL)
    INTEGER IA, N, M, MM, IV, ICOUNT(M), IFAIL
    DOUBLE PRECISION A(IA,N), ALB, UB, R(M), V(IV,M), D(N), E(N), E2(N), X(N,7), G(N)
    LOGICAL C(N)
```

3. Description

The real symmetric matrix A is reduced to a symmetric tridiagonal matrix T by Householder’s method. The eigenvalues which lie within a given interval \([l,u]\), are calculated by the method of bisection. The corresponding eigenvectors of T are calculated by inverse iteration. A back-transformation is then performed to obtain the eigenvectors of the original matrix A.

4. References


5. Parameters

1: A(IA,N) -- DOUBLE PRECISION array Input/Output
   On entry: the lower triangle of the n by n symmetric matrix A. The elements of the array above the diagonal need not be set. On exit: the elements of A below the diagonal are overwritten, and the rest of the array is unchanged.

2: IA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F02BBF is called.
   Constraint: IA >= N.

3: N -- INTEGER Input
   On entry: n, the order of the matrix A.
4: ALB -- DOUBLE PRECISION Input
5: UB -- DOUBLE PRECISION Input
   On entry: l and u, the lower and upper end-points of the
   interval within which eigenvalues are to be calculated.
6: M -- INTEGER Input
   On entry: an upper bound for the number of eigenvalues
   within the interval.
7: MM -- INTEGER Output
   On exit: the actual number of eigenvalues within the
   interval.
8: R(M) -- DOUBLE PRECISION array Output
   On exit: the eigenvalues, not necessarily in ascending
   order.
9: V(IV,M) -- DOUBLE PRECISION array Output
   On exit: the eigenvectors, stored by columns. The ith
   column corresponds to the ith eigenvalue. The eigenvectors
   are normalised so that the sum of the squares of the
   elements are equal to 1.
10: IV -- INTEGER Input
    On entry:
    the first dimension of the array V as declared in the
    (sub)program from which F02BBF is called.
    Constraint: IV >= N.
11: D(N) -- DOUBLE PRECISION array Workspace
12: E(N) -- DOUBLE PRECISION array Workspace
13: E2(N) -- DOUBLE PRECISION array Workspace
14: X(N,7) -- DOUBLE PRECISION array Workspace
15: G(N) -- DOUBLE PRECISION array Workspace
16: C(N) -- LOGICAL array Workspace
17: ICOUNT(M) -- INTEGER array Output
    On exit: ICOUNT(i) contains the number of iterations for
    the ith eigenvalue.
18: IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
CHAPTER 15. CHAPTER N

Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
M is less than the number of eigenvalues in the given interval. On exit MM contains the number of eigenvalues in the interval. Rerun with this value for M.

IFAIL= 2
More than 5 iterations are required to determine any one eigenvector.

7. Accuracy

There is no guarantee of the accuracy of the eigenvectors as the results depend on the original matrix and the multiplicity of the roots. For a detailed error analysis see Wilkinson and Reinsch [1] pp 222 and 436.

8. Further Comments

The time taken by the routine is approximately proportional to n

This subroutine should only be used when less than 25% of the eigenvalues and the corresponding eigenvectors are required. Also this subroutine is less efficient with matrices which have multiple eigenvalues.

9. Example

To calculate the eigenvalues lying between -2.0 and 3.0, and the corresponding eigenvectors of the real symmetric matrix:

\[
\begin{pmatrix}
0.5 & 0.0 & 2.3 & -2.6 \\
0.0 & 0.5 & -1.4 & -0.7 \\
2.3 & -1.4 & 0.5 & 0.0 \\
-2.6 & -0.7 & 0.0 & 0.5
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

F02BJF calculates all the eigenvalues and, if required, all the eigenvectors of the generalized eigenproblem $Ax = \lambda Bx$ where $A$ and $B$ are real, square matrices, using the QZ algorithm.

2. Specification

```
SUBROUTINE F02BJF (N, A, IA, B, IB, EPS1, ALFR, ALFI,  
1      BETA, MATV, V, IV, ITER, IFAIL)
INTEGER N, IA, IB, IV, ITER(N), IFAIL
DOUBLE PRECISION A(IA,N), B(IB,N), EPS1, ALFR(N), ALFI(N),  
1      BETA(N), V(IV,N)
LOGICAL MATV
```

3. Description

All the eigenvalues and, if required, all the eigenvectors of the generalized eigenproblem $Ax = \lambda Bx$ where $A$ and $B$ are real, square matrices, are determined using the QZ algorithm. The QZ algorithm consists of 4 stages:

(a) $A$ is reduced to upper Hessenberg form and at the same time $B$ is reduced to upper triangular form.

(b) $A$ is further reduced to quasi-triangular form while the triangular form of $B$ is maintained.

(c) The quasi-triangular form of $A$ is reduced to triangular form and the eigenvalues extracted. This routine does not actually produce the eigenvalues $(\lambda_j)$, but instead returns $(\alpha_j)$ and $(\beta_j)$ such that

$$
\lambda_j = \frac{\alpha_j}{\beta_j}, \quad j=1,2,\ldots,n
$$

The division by $(\beta_j)$ becomes the responsibility of the user's program, since $(\beta_j)$ may be zero indicating an
infinite eigenvalue. Pairs of complex eigenvalues occur with \((\alpha)_j / (\beta)_j\) and \((\alpha)_{j+1} / (\beta)_{j+1}\) complex conjugates, even though \((\alpha)_j\) and \((\alpha)_{j+1}\) are not conjugate.

(d) If the eigenvectors are required (MATV = .TRUE.), they are obtained from the triangular matrices and then transformed back into the original co-ordinate system.

4. References


5. Parameters

1: \(N\) -- INTEGER Input
   On entry: \(n\), the order of the matrices \(A\) and \(B\).

2: \(A(IA,N)\) -- DOUBLE PRECISION array Input/Output
   On entry: the \(n\) by \(n\) matrix \(A\). On exit: the array is overwritten.

3: \(IA\) -- INTEGER Input
   On entry: the first dimension of the array \(A\) as declared in the (sub)program from which F02BJF is called.
   Constraint: \(IA \geq N\).

4: \(B(IB,N)\) -- DOUBLE PRECISION array Input/Output
   On entry: the \(n\) by \(n\) matrix \(B\). On exit: the array is overwritten.

5: \(IB\) -- INTEGER Input
   On entry: the first dimension of the array \(B\) as declared in the (sub)program from which F02BJF is called.
   Constraint: \(IB \geq N\).

6: \(EPS1\) -- DOUBLE PRECISION Input
On entry: the tolerance used to determine negligible elements. If EPS1 > 0.0, an element will be considered negligible if it is less than EPS1 times the norm of its matrix. If EPS1 <= 0.0, machine precision is used in place of EPS1. A positive value of EPS1 may result in faster execution but less accurate results.

7: ALFR(N) -- DOUBLE PRECISION array Output
8: ALFI(N) -- DOUBLE PRECISION array Output
   On exit: the real and imaginary parts of \((alpha)\) , for
   \(j =\, 1, 2, \ldots, n\).
9: BETA(N) -- DOUBLE PRECISION array Output
   On exit: \((beta)\) , for \(j = 1, 2, \ldots, n\).
10: MATV -- LOGICAL Input
    On entry: MATV must be set .TRUE. if the eigenvectors are required, otherwise .FALSE..

11: V(IV,N) -- DOUBLE PRECISION array Output
    On exit: if MATV = .TRUE., then
    (i) if the \(j\)th eigenvalue is real, the \(j\)th column of V contains its eigenvector;
    
    (ii) if the \(j\)th and \((j+1)\)th eigenvalues form a complex pair, the \(j\)th and \((j+1)\)th columns of V contain the real and imaginary parts of the eigenvector associated with the first eigenvalue of the pair. The conjugate of this vector is the eigenvector for the conjugate eigenvalue.
    
    Each eigenvector is normalised so that the component of largest modulus is real and the sum of squares of the moduli equal one.
    
    If MATV = .FALSE., V is not used.

12: IV -- INTEGER Input
    On entry: the first dimension of the array V as declared in the (sub)program from which FO2BJF is called.
    Constraint: IV \(\geq N\).

13: ITER(N) -- INTEGER array Output
    On exit: ITER(\(j\)) contains the number of iterations needed to obtain the \(j\)th eigenvalue. Note that the eigenvalues are obtained in reverse order, starting with the \(n\)th.
14: IFAIL -- INTEGER

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= i

More than 30*N iterations are required to determine all the diagonal 1 by 1 or 2 by 2 blocks of the quasi-triangular form in the second step of the QZ algorithm. IFAIL is set to the index i of the eigenvalue at which this failure occurs. If the soft failure option is used, \((alpha)_j\) and \((beta)_j\) are correct for \(j=i+1,i+2,...,n\), but \(V\) does not contain any correct eigenvectors.

7. Accuracy

The computed eigenvalues are always exact for a problem \((A+E)x=(lambda)(B+F)x\) where \(||E||/||A||\) and \(||F||/||B||\) are both of the order of max(EPS1,(epsilon)), EPS1 being defined as in Section 5 and (epsilon) being the machine precision.

Note: interpretation of results obtained with the QZ algorithm often requires a clear understanding of the effects of small changes in the original data. These effects are reviewed in Wilkinson [3], in relation to the significance of small values of \((alpha)_j\) and \((beta)_j\). It should be noted that if \((alpha)_j\) and \((beta)_j\) are both small for any \(j\), it may be that no reliance can be placed on any of the computed eigenvalues \((lambda)_j\). The user is recommended to study [3] and, if in difficulty, to seek expert advice on determining the sensitivity of the eigenvalues to perturbations in the data.

8. Further Comments

The time taken by the routine is approximately proportional to \(n^3\) and also depends on the value chosen for parameter EPS1.

9. Example
To find all the eigenvalues and eigenvectors of $Ax = \lambda Bx$
where

$$
\begin{pmatrix}
3.9 & 12.5 & -34.5 & -0.5 \\
4.3 & 21.5 & -47.5 & 7.5
\end{pmatrix}
\begin{pmatrix}
4.3 & 21.5 & -43.5 & 3.5 \\
4.4 & 26.0 & -46.0 & 6.0
\end{pmatrix}
$$

and

$$
\begin{pmatrix}
1 & 2 & -3 & 1 \\
1 & 3 & 5 & 4 \\
1 & 3 & 4 & 3 \\
1 & 3 & 4 & 4
\end{pmatrix}
$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
\[ Cx = (\lambda)x \]  
where \( C \) is an \( n \) by \( n \) matrix such that

\[ T \]
\[ BC = C B \]  
for a given positive-definite matrix \( B \). \( C \) is said to be \( B \)-symmetric. Different specifications of \( C \) allow for the solution of a variety of eigenvalue problems. For example, when

\[ T \]
\[ C = A \text{ and } B = I \text{ where } A = A \]

the routine finds the \( m \) eigenvalues of largest absolute magnitude for the standard symmetric eigenvalue problem

\[ Ax = (\lambda)x. \]

The routine is intended for the case where \( A \) is sparse.

As a second example, when

\[ -1 \]
\[ C = B A \]

where

\[ T \]
\[ A = A \]

the routine finds the \( m \) eigenvalues of largest absolute magnitude for the generalized symmetric eigenvalue problem

\[ Ax = (\lambda)Bx. \]

The routine is intended for the case where \( A \) and \( B \) are sparse.

The routine does not require \( C \) explicitly, but \( C \) is specified via a user-supplied routine \( IMAGE \) which, given an \( n \) element vector \( z \), computes the image \( w \) given by

\[ w = Cz. \]

For instance, in the above example, where \( C = B A \), routine \( IMAGE \) will need to solve the positive-definite system of equations

\[ Bw = Az \text{ for } w. \]
To find the m eigenvalues of smallest absolute magnitude of (3) 
we can choose C=A and hence find the reciprocals of the 
required eigenvalues, so that IMAGE will need to solve Aw=z for 
-1 
w, and correspondingly for (4) we can choose C=A \ B and solve 
Aw=Bz for w.

A table of examples of choice of IMAGE is given in Table 3.1. It 
should be remembered that the routine also returns the 
corresponding eigenvectors and that B is positive-definite. 
Throughout A is assumed to be symmetric and, where necessary, 
non-singularity is also assumed.

<table>
<thead>
<tr>
<th>Eigenvalues Required</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ax=(\lambda)x (B=I)Ax=(\lambda)x Bx=(\lambda)x</td>
<td>Compute Solve Compute</td>
</tr>
<tr>
<td>Largest w=Az Bw=Az w=ABz</td>
<td></td>
</tr>
<tr>
<td>Smallest (Find 1/(\lambda)) Aw=z Ax=Bz Av=z Bw=(\nu)</td>
<td>Solve Solve Solve</td>
</tr>
<tr>
<td>Furthest from (sigma) (Find (\lambda)-(sigma)) w=(A-(\sigma)I)z Bw=(A-\sigma I)z w=(AB-\sigma I)z</td>
<td>Compute Solve Compute</td>
</tr>
<tr>
<td>Closest to (sigma) (Find 1/((\lambda)-(sigma))) (A-(\sigma)I)w=z (A-(\sigma)I)w=Bz (AB-(\sigma)I)w=z</td>
<td>Solve Solve Solve</td>
</tr>
</tbody>
</table>

Table 3.1

The Requirement of IMAGE for Various Problems

The matrix B also need not be supplied explicitly, but is 
specified via a user-supplied routine DOT which, given n element 
\mathbf{T} vectors z and w, computes the generalized dot product w Bz.

F02FJF is based upon routine SMITZ (see Nikolai [1]), which is 
itself a derivative of the Algol procedure ritzit (see
Rutishauser [4]), and uses the method of simultaneous (subspace) iteration. (See Parlett [2] for description, analysis and advice on the use of the method.)

The routine performs simultaneous iteration on \( k > m \) vectors. Initial estimates to \( p \leq k \) eigenvectors, corresponding to the \( p \) eigenvalues of \( C \) of largest absolute value, may be supplied by the user to F02FJF. When possible \( k \) should be chosen so that the \( k \)th eigenvalue is not too close to the \( m \) required eigenvalues, but if \( k \) is initially chosen too small then F02FJF may be re-entered, supplying approximations to the \( k \) eigenvectors found so far and with \( k \) then increased.

At each major iteration F02FJF solves an \( r \) by \( r \) (\( r \leq k \)) eigenvalue sub-problem in order to obtain an approximation to the eigenvalues for which convergence has not yet occurred. This approximation is refined by Chebyshev acceleration.

4. References


5. Parameters

1: \( \text{N} \) -- INTEGER
   Input
   On entry: \( n \), the order of the matrix \( C \). Constraint: \( N \geq 1 \).

2: \( \text{M} \) -- INTEGER
   Input/Output
   On entry: \( m \), the number of eigenvalues required.
   On exit: \( m \), the number of eigenvalues actually found. It is equal to \( m \) if \( IFAIL = 0 \) on exit, and is less than \( m \) if \( IFAIL = 2, 3 \) or 4. See Section 6 and Section 8 for further information.

3: \( \text{K} \) -- INTEGER
   Input
   On entry: the number of simultaneous iteration vectors to be used. Too small a value of \( K \) may inhibit convergence, while a larger value of \( K \) incurs additional storage and additional
work per iteration. Suggested value: \( K = M + 4 \) will often be a reasonable choice in the absence of better information. Constraint: \( M < K \leq N \).

4: NOITS -- INTEGER Input/Output
On entry: the maximum number of major iterations (eigenvalue sub-problems) to be performed. If NOITS <= 0, then the value 100 is used in place of NOITS. On exit: the number of iterations actually performed.

5: TOL -- DOUBLE PRECISION Input
On entry: a relative tolerance to be used in accepting eigenvalues and eigenvectors. If the eigenvalues are required to about \( t \) significant figures, then TOL should be \( 10^{-t} \).

\( TOL \) is accepted as an eigenvalue as soon as two successive approximations to \( d \) differ by less than \( (|d| \times TOL)/10 \), where \( d \) is the latest approximation to \( d \).

Once an eigenvalue has been accepted, then an eigenvector is accepted as soon as \( (d_f)/(d - d_k) < TOL \), where \( f \) is the normalised residual of the current approximation to the eigenvector (see Section 8 for further information). The values of the \( f \) and \( d \) can be printed from routine MONIT.

If TOL is supplied outside the range \((\text{epsilon}), 1.0\)\), where \( \text{epsilon} \) is the machine precision, then the value \( \text{epsilon} \) is used in place of TOL.

6: DOT -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure

\( T \)

DOT must return the value \( w^T z \) for given vectors \( w \) and \( z \).

For the standard eigenvalue problem, where \( B = I \), DOT must return the dot product \( w^T z \).

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION DOT (IFLAG, N, Z, W, 1 RWORK, LRWORK, 2 IWORK, LIWORK)}\]

\[
\text{INTEGER IFLAG, N, LRWORK, IWORK(LIWORK), LIWORK}\]

\[
\text{DOUBLE PRECISION Z(N), W(N), RWORK(LRWORK)}\]
1: IFLAG -- INTEGER Input/Output
   On entry: IFLAG is always non-negative. On exit: IFLAG
   may be used as a flag to indicate a failure in the
   computation of $w B z$. If IFLAG is negative on exit from
   DOT, then F02FJF will exit immediately with IFAIL set
   to IFLAG. Note that in this case DOT must still be
   assigned a value.

2: N -- INTEGER Input
   On entry: the number of elements in the vectors $z$ and $w$
   and the order of the matrix $B$.

3: Z(N) -- DOUBLE PRECISION array Input
   On entry: the vector $z$ for which $w B z$ is required.

4: W(N) -- DOUBLE PRECISION array Input
   On entry: the vector $w$ for which $w B z$ is required.

5: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace

6: LRWORK -- INTEGER Input

7: IWORK(LIWORK) -- INTEGER array User Workspace

8: LIWORK -- INTEGER Input
   DOT is called from F02FJF with the parameters RWORK,
   LRWORK, IWORK and LIWORK as supplied to F02FJF. The
   user is free to use the arrays RWORK and IWORK to
   supply information to DOT and to IMAGE as an
   alternative to using COMMON.
   DOT must be declared as EXTERNAL in the (sub)program
   from which F02FJF is called. Parameters denoted as
   Input must not be changed by this procedure.

7: IMAGE -- SUBROUTINE, supplied by the user.
   External Procedure
   IMAGE must return the vector $w = C z$ for a given vector $z$.
   Its specification is:

   SUBROUTINE IMAGE (IFLAG, N, Z, W, RWORK, LRWORK,
   1   IWORK, LIWORK)
   INTEGER IFLAG, N, LRWORK, IWORK(LIWORK),
   1   LIWORK
DOUBLE PRECISION Z(N), W(N), RWORK(LRWORK)

1:  IFLAG -- INTEGER         Input/Output
   On entry: IFLAG is always non-negative. On exit: IFLAG
   may be used as a flag to indicate a failure in the
   computation of w. If IFLAG is negative on exit from
   IMAGE, then F02FJF will exit immediately with IFAIL set
   to IFLAG.

2:  N -- INTEGER             Input
   On entry: n, the number of elements in the vectors w
   and z, and the order of the matrix C.

3:  Z(N) -- DOUBLE PRECISION array      Input
   On entry: the vector z for which Cz is required.

4:  W(N) -- DOUBLE PRECISION array      Output
   On exit: the vector w=Cz.

5:  RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace

6:  LRWORK -- INTEGER            Input

7:  IWORK(LIWORK) -- INTEGER array    User Workspace

8:  LIWORK -- INTEGER            Input
   IMAGE is called from F02FJF with the parameters RWORK,
   LRWORK, IWORK and LIWORK as supplied to F02FJF. The
   user is free to use the arrays RWORK and IWORK to
   supply information to IMAGE and DOT as an alternative
   to using COMMON.
   IMAGE must be declared as EXTERNAL in the (sub)program
   from which F02FJF is called. Parameters denoted as
   Input must not be changed by this procedure.

8:  MONIT -- SUBROUTINE, supplied by the user.
   External Procedure
   MONIT is used to monitor the progress of F02FJF. MONIT may
   be the dummy subroutine F02FJZ if no monitoring is actually
   required. (F02FJZ is included in the NAG Foundation Library
   and so need not be supplied by the user. The routine name
   F02FJZ may be implementation dependent: see the Users’ Note
   for your implementation for details.) MONIT is called after
   the solution of each eigenvalue sub-problem and also just
   prior to return from F02FJF. The parameters ISTATE and
   NEXTIT allow selective printing by MONIT.

   Its specification is:

   SUBROUTINE MONIT (ISTATE, NEXTIT, NEVALS,
INTEGER ISTATE, NEXTIT, NEVALS, NEVECS, K
DOUBLE PRECISION F(K), D(K)

1: ISTATE -- INTEGER Input
On entry: ISTATE specifies the state of F02FJF and will have values as follows:
ISTATE = 0
   No eigenvalue or eigenvector has just been accepted.
ISTATE = 1
   One or more eigenvalues have been accepted since the last call to MONIT.
ISTATE = 2
   One or more eigenvectors have been accepted since the last call to MONIT.
ISTATE = 3
   One or more eigenvalues and eigenvectors have been accepted since the last call to MONIT.
ISTATE = 4
   Return from F02FJF is about to occur.

2: NEXTIT -- INTEGER Input
On entry: the number of the next iteration.

3: NEVALS -- INTEGER Input
On entry: the number of eigenvalues accepted so far.

4: NEVECS -- INTEGER Input
On entry: the number of eigenvectors accepted so far.

5: K -- INTEGER Input
On entry: k, the number of simultaneous iteration vectors.

6: F(K) -- DOUBLE PRECISION array Input
On entry: a vector of error quantities measuring the state of convergence of the simultaneous iteration vectors. See the parameter TOL of F02FJF above and Section 8 for further details. Each element of F is initially set to the value 4.0 and an element remains at 4.0 until the corresponding vector is tested.

7: D(K) -- DOUBLE PRECISION array Input
On entry: D(i) contains the latest approximation to the
absolute value of the ith eigenvalue of C.
MONIT must be declared as EXTERNAL in the (sub)program from which F02FJF is called. Parameters denoted as Input must not be changed by this procedure.

9: NOVECS -- INTEGER Input
On entry: the number of approximate vectors that are being supplied in X. If NOVECS is outside the range (0,K), then the value 0 is used in place of NOVECS.

10: X(NRX,K) -- DOUBLE PRECISION array Input/Output
On entry: if 0 < NOVECS <= K, the first NOVECS columns of X must contain approximations to the eigenvectors corresponding to the NOVECS eigenvalues of largest absolute value of C. Supplying approximate eigenvectors can be useful when reasonable approximations are known, or when the routine is being restarted with a larger value of K. Otherwise it is not necessary to supply approximate vectors, as simultaneous iteration vectors will be generated randomly by the routine. On exit: if IFAIL = 0, 2, 3 or 4, the first m' columns contain the eigenvectors corresponding to the eigenvalues returned in the first m' elements of D (see below); and the next k-m'-1 columns contain approximations to the eigenvectors corresponding to the approximate eigenvalues returned in the next k-m'-1 elements of D. Here m' is the value returned in M (see above), the number of eigenvalues actually found. The kth column is used as workspace.

11: NRX -- INTEGER Input
On entry: the first dimension of the array X as declared in the (sub)program from which F02FJF is called. Constraint: NRX >= N.

12: D(K) -- DOUBLE PRECISION array Output
On exit: if IFAIL = 0, 2, 3 or 4, the first m' elements contain the first m' eigenvalues in decreasing order of magnitude; and the next k-m'-1 elements contain approximations to the next k-m'-1 eigenvalues. Here m' is the value returned in M (see above), the number of eigenvalues actually found. D(k) contains the value e where (-e,e) is the latest interval over which Chebyshev acceleration is performed.

13: WORK(LWORK) -- DOUBLE PRECISION array Workspace

14: LWORK -- INTEGER Input
On entry: the length of the array WORK, as declared in the (sub)program from which F02FJF is called. Constraint:
CHAPTER 15. CHAPTER N

LWORK$\geq 3K + \max(KK, 2N)$.

15: RWORK(LRWORK) -- DOUBLE PRECISION array User Workspace
RWORK is not used by F02FJF, but is passed directly to
routines DOT and IMAGE and may be used to supply information
to these routines.

16: LRWORK -- INTEGER Input
On entry: the length of the array RWORK, as declared in the
(sub)program from which F02FJF is called. Constraint: LRWORK
$\geq 1$.

17: IWORK(LIWORK) -- INTEGER array User Workspace
IWORK is not used by F02FJF, but is passed directly to
routines DOT and IMAGE and may be used to supply information
to these routines.

18: LIWORK -- INTEGER Input
On entry: the length of the array IWORK, as declared in the
(sub)program from which F02FJF is called. Constraint: LIWORK
$\geq 1$.

19: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /= 0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit. To suppress
the output of an error message when soft failure occurs, set
IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL< 0
A negative value of IFAIL indicates an exit from F02FJF
because the user has set IFLAG negative in DOT or IMAGE. The
value of IFAIL will be the same as the user's setting of
IFLAG.

IFAIL= 1
On entry N < 1,
or $M < 1$,
or $M \geq K$,
or $K \geq N$,
or $NRX < N$,
or $LWORK < 3(K+max(K+K*N))$,
or $LRWORK < 1$,
or $LIWORK < 1$.

**IFAIL= 2**
Not all the requested eigenvalues and vectors have been obtained. Approximations to the $r$th eigenvalue are oscillating rapidly indicating that severe cancellation is occurring in the $r$th eigenvector and so $M$ is returned as $(r-1)$. A restart with a larger value of $K$ may permit convergence.

**IFAIL= 3**
Not all the requested eigenvalues and vectors have been obtained. The rate of convergence of the remaining eigenvectors suggests that more than NOITS iterations would be required and so the input value of $M$ has been reduced. A restart with a larger value of $K$ may permit convergence.

**IFAIL= 4**
Not all the requested eigenvalues and vectors have been obtained. NOITS iterations have been performed. A restart, possibly with a larger value of $K$, may permit convergence.

**IFAIL= 5**
This error is very unlikely to occur, but indicates that convergence of the eigenvalue sub-problem has not taken place. Restarting with a different set of approximate vectors may allow convergence. If this error occurs the user should check carefully that F02FJF is being called correctly.

7. Accuracy

Eigenvalues and eigenvectors will normally be computed to the accuracy requested by the parameter TOL, but eigenvectors corresponding to small or to close eigenvalues may not always be computed to the accuracy requested by the parameter TOL. Use of the routine MONIT to monitor acceptance of eigenvalues and
eigenvectors is recommended.

8. Further Comments

The time taken by the routine will be principally determined by the time taken to solve the eigenvalue sub-problem and the time taken by the routines DOT and IMAGE. The time taken to solve an eigenvalue sub-problem is approximately proportional to $nk^2$. It is important to be aware that several calls to DOT and IMAGE may occur on each major iteration.

As can be seen from Table 3.1, many applications of F02FJF will require routine IMAGE to solve a system of linear equations. For example, to find the smallest eigenvalues of $Ax = (\lambda)Bx$, IMAGE needs to solve equations of the form $Av = Bw$ for $w$ and routines from Chapters F01 and F04 of the NAG Foundation Library will frequently be useful in this context. In particular, if $A$ is a positive-definite variable band matrix, F04MCF may be used after $A$ has been factorized by F01MCF. Thus factorization need be performed only once prior to calling F02FJF. An illustration of this type of use is given in the example program in Section 9.

An approximation $d$, to the $i$th eigenvalue, is accepted as soon as $d$ and the previous approximation differ by less than $|d| \times 10^{-TOL/10}$. Eigenvectors are accepted in groups corresponding to clusters of eigenvalues that are equal, or nearly equal, in absolute value and that have already been accepted. If $d$ is the last eigenvalue in such a group and we define the residual $r$ as

$$r = Cx - y$$

where $y$ is the projection of $Cx$, with respect to $B$, onto the space spanned by $x_1, x_2, \ldots, x_r$ and $x$ is the current approximation to the $j$th eigenvector, then the value $f$ returned in MONIT is given by

$$f = 2 T$$
\[ f = \max_{i \leq j} \| r \| / \| C x \| \| x \| = x B x \]

and each vector in the group is accepted as an eigenvector if

\[
\frac{|d_f|}{|d_i| - e} < \text{TOL}
\]

where \( e \) is the current approximation to \( |d_i| \). The values of the \( f_k \) are systematically increased if the convergence criteria \( i \) appear to be too strict. See Rutishauser [4] for further details.

The algorithm implemented by F02FJF differs slightly from SIMITZ (Nikolai [1]) in that the eigenvalue sub-problem is solved using the singular value decomposition of the upper triangular matrix \( R^T \) of the Gram-Schmidt factorization of \( C x \), rather than forming \( R R^T \).

9. Example

To find the four eigenvalues of smallest absolute value and corresponding eigenvectors for the generalized symmetric eigenvalue problem \( A x = (\lambda) B x \), where \( A \) and \( B \) are the 16 by 16 matrices

\[
A = \begin{pmatrix}
1 & a & a & a \\
a & 1 & a & a \\
a & a & 1 & a \\
a & a & a & 1 \\
a & a & a & 1 \\
a & a & 1 & a \\
a & a & a & a \\
a & a & a & a
\end{pmatrix}
\]

where \( a = -\frac{4}{3} \)
where $b = \frac{-1}{2}$

TOL is taken as 0.0001 and 6 iteration vectors are used. F01MAF is used to factorize the matrix $A$, prior to calling F02FJF, and F04MAF is used within IMAGE to solve the equations $Aw = Bz$ for $w$. Details of the factorization of $A$ are passed from F01MAF to F04MAF by means of the COMMON block BLOCK1.

Output from MONIT occurs each time ISTATE is non-zero. Note that the required eigenvalues are the reciprocals of the eigenvalues returned by F02FJF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**F02 -- Eigenvalue and Eigenvectors**

**F02WEF**

F02WEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. **Purpose**

F02WEF returns all, or part, of the singular value decomposition of a general real matrix.

2. **Specification**
SUBROUTINE F02WEF (M, N, A, LDA, NCOLB, B, LDB, WANTQ, Q, 
1     LDQ, SV, WANTP, PT, LDPT, WORK, IFAIL)
INTEGER M, N, LDA, NCOLB, LDB, LDQ, LDPT, IFAIL
DOUBLE PRECISION A(LDA,*), B(LDB,*), Q(LDQ,*), SV(*), PT 
1     (LDPT,*), WORK(*)
LOGICAL WANTQ, WANTP

3. Description

The m by n matrix A is factorized as

\[ T \]
\[ A = QDP, \]

where

\[
\begin{align*}
(S) & \\
D &= (0), & m &> n, \\
D &= S, & m &= n, \\
D &= (S \ 0), & m &< n, \\
\end{align*}
\]

Q is an m by m orthogonal matrix, P is an n by n orthogonal matrix and S is a min(m,n) by min(m,n) diagonal matrix with non-negative diagonal elements, \( s_1, s_2, \ldots, s_{\min(m,n)} \), ordered such that

\[ s_1 \geq s_2 \geq \ldots \geq s_{\min(m,n)} \geq 0. \]

The first \( \min(m,n) \) columns of Q are the left-hand singular vectors of A, the diagonal elements of S are the singular values of A and the first \( \min(m,n) \) columns of P are the right-hand singular vectors of A.

Either or both of the left-hand and right-hand singular vectors of A may be requested and the matrix C given by

\[ T \]
\[ C = QB, \]

where B is an m by ncolb given matrix, may also be requested.

The routine obtains the singular value decomposition by first reducing A to upper triangular form by means of Householder transformations, from the left when \( m \geq n \) and from the right when \( m < n \). The upper triangular form is then reduced to bidiagonal form.
by Givens plane rotations and finally the QR algorithm is used to obtain the singular value decomposition of the bidiagonal form.

Good background descriptions to the singular value decomposition are given in Dongarra et al [1], Hammarling [2] and Wilkinson [3] DSVDC.

Note that if \( K \) is any orthogonal diagonal matrix so that

\[
T
K K = I,
\]

(so that \( K \) has elements +1 or -1 on the diagonal)

then

\[
T
A = (QK)D(PK)
\]

is also a singular value decomposition of \( A \).

4. References


5. Parameters

1: \( M \) -- INTEGER 
Input
On entry: the number of rows, \( m \), of the matrix \( A \).
Constraint: \( M \geq 0 \).

When \( M = 0 \) then an immediate return is effected.

2: \( N \) -- INTEGER 
Input
On entry: the number of columns, \( n \), of the matrix \( A \).
Constraint: \( N \geq 0 \).

When \( N = 0 \) then an immediate return is effected.

3: \( A (LDA,*) \) -- DOUBLE PRECISION array 
Input/Output
Note: the second dimension of the array \( A \) must be at least \( \max (1,N) \).
On entry: the leading m by n part of the array A must contain the matrix A whose singular value decomposition is required. On exit: if M >= N and WANTQ = .TRUE., then the leading m by n part of A will contain the first n columns of the orthogonal matrix Q.

If M < N and WANTP = .TRUE., then the leading m by n part of A will contain the first m rows of the orthogonal matrix P.

If M >= N and WANTQ = .FALSE. and WANTP = .TRUE., then the leading n by n part of A will contain the first n rows of the orthogonal matrix P.

Otherwise the leading m by n part of A is used as internal workspace.

4: LDA -- INTEGER
On entry: the first dimension of the array A as declared in the (sub)program from which F02WEF is called.
Constraint: LDA >= max(1,M).

5: NCOLB -- INTEGER
On entry: ncolb, the number of columns of the matrix B. When NCOLB = 0 the array B is not referenced. Constraint: NCOLB >= 0.

6: B(LDB,*) -- DOUBLE PRECISION array
On entry: if NCOLB > 0, the leading m by ncolb part of the array B must contain the matrix to be transformed. On exit: B is overwritten by the m by ncolb matrix Q B.

7: LDB -- INTEGER
On entry: the first dimension of the array B as declared in the (sub)program from which F02WEF is called.
Constraint: if NCOLB > 0 then LDB >= max(1,M).

8: WANTQ -- LOGICAL
On entry: WANTQ must be .TRUE., if the left-hand singular vectors are required. If WANTQ = .FALSE., then the array Q is not referenced.

9: Q(LDQ,*) -- DOUBLE PRECISION array
Note: the second dimension of the array Q must be at least
max(1,M).
On exit: if M < N and WANTQ = .TRUE., the leading m by m part of the array Q will contain the orthogonal matrix Q. Otherwise the array Q is not referenced.

10: LDQ -- INTEGER Input
On entry:
the first dimension of the array Q as declared in the (sub)program from which F02WEF is called.
Constraint: if M < N and WANTQ = .TRUE., LDQ >= max(1,M).

11: SV(*) -- DOUBLE PRECISION array Output
Note: the length of SV must be at least min(M,N). On exit:
the min(M,N) diagonal elements of the matrix S.

12: WANTP -- LOGICAL Input
On entry: WANTP must be .TRUE. if the right-hand singular vectors are required. If WANTP = .FALSE., then the array PT is not referenced.

13: PT(LDPT,*) -- DOUBLE PRECISION array Output
Note: the second dimension of the array PT must be at least max(1,N).
On exit: if M >= N and WANTQ and WANTP are .TRUE., the leading n by n part of the array PT will contain the orthogonal matrix P. Otherwise the array PT is not referenced.

14: LDPT -- INTEGER Input
On entry:
the first dimension of the array PT as declared in the (sub)program from which F02WEF is called.
Constraint: if M >= N and WANTQ and WANTP are .TRUE., LDPT >= max(1,N).

15: WORK(*) -- DOUBLE PRECISION array Output
Note: the length of WORK must be at least max(1,lwork), where lwork must be as given in the following table:

\[
\begin{align*}
M & \geq N \\
\text{WANTQ is .TRUE. and WANTP = .TRUE.} & \Rightarrow lwork=\max(N+5*(N-1),N+NCOLB,4) \\
\text{WANTQ = .TRUE. and WANTP = .FALSE.} & \Rightarrow lwork=\max(N+4*(N-1),N+NCOLB,4) \\
\text{WANTQ = .FALSE. and WANTP = .TRUE.} & \Rightarrow lwork=\max(N+3*(N-1),N+NCOLB,4)
\end{align*}
\]
\[\text{lwork}\text{=}\max(3(N-1),2) \text{ when NCOLB} = 0\]
\[\text{lwork}\text{=}\max(5(N-1),2) \text{ when NCOLB} > 0\]
\[\text{WANTQ} = \text{.FALSE. and WANTP} = \text{.FALSE.}\]
\[\text{lwork}\text{=}\max(2(N-1),2) \text{ when NCOLB} = 0\]
\[\text{lwork}\text{=}\max(3(N-1),2) \text{ when NCOLB} > 0\]
\[M < N\]
\[\text{WANTQ} = \text{.TRUE. and WANTP} = \text{.TRUE.}\]
\[2\]
\[\text{lwork}\text{=}\max(M +5(M-1),2)\]
\[\text{WANTQ} = \text{.TRUE. and WANTP} = \text{.FALSE.}\]
\[\text{lwork}\text{=}\max(3(M-1),1)\]
\[\text{WANTQ} = \text{.FALSE. and WANTP} = \text{.TRUE.}\]
\[2\]
\[\text{lwork}\text{=}\max(M +3(M-1),2) \text{ when NCOLB} = 0\]
\[2\]
\[\text{lwork}\text{=}\max(M +5(M-1),2) \text{ when NCOLB} > 0\]
\[\text{WANTQ} = \text{.FALSE. and WANTP} = \text{.FALSE.}\]
\[\text{lwork}\text{=}\max(2(M-1),1) \text{ when NCOLB} = 0\]
\[\text{lwork}\text{=}\max(3(M-1),1) \text{ when NCOLB} > 0\]

On exit: \(\text{WORK}(\min(M,N))\) contains the total number of iterations taken by the \(R\) algorithm.

The rest of the array is used as workspace.

16: IFAIL -- INTEGER
Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL=-1

One or more of the following conditions holds:
M < 0,
N < 0,
LDA < M,
NCOIB < 0,
LDB < M and NCOIB > 0,
LDQ < M and M < N and WANTQ = .TRUE.,
LDPT < N and M >= N and WANTQ = .TRUE., and WANTP = .TRUE.

IFAIL > 0
The QR algorithm has failed to converge in 50*min(m,n) iterations. In this case SV(1), SV(2),..., SV(IFAIL) may not have been found correctly and the remaining singular values may not be the smallest. The matrix A will nevertheless have been factorized as $A = QEP$, where the leading min(m,n) by min(m,n) part of E is a bidiagonal matrix with SV(1), SV(2), ..., SV(min(m,n)) as the diagonal elements and WORK(1), WORK(2), ..., WORK(min(m,n)-1) as the super-diagonal elements.

This failure is not likely to occur.

7. Accuracy

The computed factors Q, D and P satisfy the relation

$$T QDP = A + E,$$

where

$$||E|| <= c(\epsilon)||A||,$$

(\epsilon) being the machine precision, c is a modest function of m and n and ||.|| denotes the spectral (two) norm. Note that ||A|| = sv .

8. Further Comments

Following the use of this routine the rank of A may be estimated by a call to the INTEGER FUNCTION F06KLF(*). The statement:

IRANK = F06KLF(MIN(M, N), SV, 1, TOL)
returns the value \((k-1)\) in IRANK, where \(k\) is the smallest integer for which \(SV(k) < \text{tol} \times SV(1)\), where \(\text{tol}\) is the tolerance supplied in TOL, so that IRANK is an estimate of the rank of \(S\) and thus also of \(A\). If TOL is supplied as negative then the machine precision is used in place of TOL.

9. Example

9.1. Example 1

To find the singular value decomposition of the 5 by 3 matrix

\[
\begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{pmatrix}
\]

\(A\)

and to find the vector \(Qb\) for the vector

\[
\begin{pmatrix}
1.1 \\
0.9 \\
0.6 \\
0.0 \\
-0.8
\end{pmatrix}
\]

\(b\)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

9.2. Example 2

To find the singular value decomposition of the 3 by 5 matrix

\[
\begin{pmatrix}
2.0 & 2.0 & 1.6 & 2.0 & 1.2 \\
2.5 & 2.5 & -0.4 & -0.5 & -0.3 \\
2.5 & 2.5 & 2.8 & 0.5 & -2.9
\end{pmatrix}
\]

\(A\)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

F02XEF returns all, or part, of the singular value decomposition
of a general complex matrix.

2. Specification

SUBROUTINE F02XEF (M, N, A, LDA, NCOLB, B, LDB, WANTQ, Q,
1    LDQ, SV, WANTP, PH, LDPH, RWORK, CWORK,
2    IFAIL)
    INTEGER M, N, LDA, NCOLB, LDB, LDQ, LDPH,
1    IFAIL
    DOUBLE PRECISION SV(*), RWORK(*)
    COMPLEX(KIND=KIND(1.0D0)) A(LDA,*), B(LDB,*), Q(LDQ,*),
1    PH(LDPH,*), CWORK(*)
    LOGICAL WANTQ, WANTP

3. Description

The m by n matrix A is factorized as

\[ H \]

\[ A = QDP \]

where

\[
\begin{align*}
  (S) & \quad m > n, \\
  D = (0) & \quad m = n, \\
  D = S & \quad m < n,
\end{align*}
\]

Q is an m by m unitary matrix, P is an n by n unitary matrix and
S is a min(m,n) by min(m,n) diagonal matrix with real non-
negative diagonal elements, sv , sv ,..., sv , ordered such
1 2 \[ \text{min}(m,n) \]
that

\[ sv \geq sv \geq \ldots \geq sv \geq 0. \]
1 2 \[ \text{min}(m,n) \]

The first min(m,n) columns of Q are the left-hand singular
vectors of A, the diagonal elements of S are the singular values
of A and the first min(m,n) columns of P are the right-hand
singular vectors of A.
Either or both of the left-hand and right-hand singular vectors of A may be requested and the matrix C given by

\[ H \]
\[ C = Q B, \]

where B is an m by ncolb given matrix, may also be requested.

The routine obtains the singular value decomposition by first reducing A to upper triangular form by means of Householder transformations, from the left when m\( \geq \)n and from the right when m\( < \)n. The upper triangular form is then reduced to bidiagonal form by Givens plane rotations and finally the QR algorithm is used to obtain the singular value decomposition of the bidiagonal form.

Good background descriptions to the singular value decomposition are given in Dongarra et al [1], Hammarling [2] and Wilkinson [3] ZSVDC.

Note that if K is any unitary diagonal matrix so that

\[ H \]
\[ KK = I, \]

then

\[ H \]
\[ A = (QK)D(PK) \]

is also a singular value decomposition of A.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number of rows, m, of the matrix A.
   Constraint: M \( \geq \) 0.
When $M = 0$ then an immediate return is effected.

2: $N$ -- INTEGER  
   Input
   On entry: the number of columns, $n$, of the matrix $A$.  
   Constraint: $N >= 0$.

When $N = 0$ then an immediate return is effected.

3: $A(LDA,*$) -- COMPLEX(KIND(1.0D)) array  
   Input/Output 
   Note: the second dimension of the array $A$ must be at least 
   max(1,$N$).
   On entry: the leading $m$ by $n$ part of the array $A$ must 
   contain the matrix $A$ whose singular value decomposition is 
   required. On exit: if $M >= N$ and WANTQ = .TRUE., then the 
   leading $m$ by $n$ part of $A$ will contain the first $n$ columns of 
   the unitary matrix $Q$.
   If $M < N$ and WANTP = .TRUE., then the leading $m$ by $n$ part of 
   $A$ will contain the first $m$ rows of the unitary matrix $P$.
   Otherwise the leading $m$ by $n$ part of $A$ is used as internal workspace.

4: $LDA$ -- INTEGER  
   Input 
   On entry: the first dimension of the array $A$ as declared in the 
   (sub)program from which F02XEF is called.
   Constraint: $LDA >= max(1,M)$.

5: $NCOLB$ -- INTEGER  
   Input 
   On entry: $ncolb$, the number of columns of the matrix $B$.
   When $NCOLB = 0$ the array $B$ is not referenced. Constraint: 
   $NCOLB >= 0$.

6: $B(LDB,*$) -- COMPLEX(KIND(1.0D)) array  
   Input/Output 
   Note: the second dimension of the array $B$ must be at least 
   max(1,$NCOLB$).
   On entry: if $NCOLB > 0$, the leading $m$ by $ncolb$ part of the 
   array $B$ must contain the matrix to be transformed. On exit: 
   $B$ is overwritten by the $m$ by $ncolb$ matrix $Q B$.

7: $LDB$ -- INTEGER  
   Input 
   On entry: the first dimension of the array $B$ as declared in the 
   (sub)program from which F02XEF is called.
   Constraint: if $NCOLB > 0$, then $LDB >= max(1,M)$.
8: WANTQ -- LOGICAL
On entry: WANTQ must be .TRUE. if the left-hand singular
vectors are required. If WANTQ = .FALSE. then the array Q is
not referenced.

9: Q(LDQ,*) -- COMPLEX(KIND(1.0D)) array
Note: the second dimension of the array Q must be at least
max(1,M).
On exit: if M < N and WANTQ = .TRUE., the leading m by m
part of the array Q will contain the unitary matrix Q.
Otherwise the array Q is not referenced.

10: LDQ -- INTEGER
On entry:
the first dimension of the array Q as declared in the
(sub)program from which F02XEF is called.
Constraint: if M < N and WANTQ = .TRUE., LDQ >= max(1,M).

11: SV(*) -- DOUBLE PRECISION array
Note: the length of SV must be at least min(M,N). On exit:
the min(m,n) diagonal elements of the matrix S.

12: WANTP -- LOGICAL
On entry: WANTP must be .TRUE. if the right-hand singular
vectors are required. If WANTP = .FALSE. then the array PH
is not referenced.

13: PH(LDPH,*) -- DOUBLE PRECISION array
Note: the second dimension of the array PH must be at least
max(1,N).
On exit: if M >= N and WANTQ and WANTP are .TRUE., the
leading n by n part of the array PH will contain the unitary
H
matrix P. Otherwise the array PH is not referenced.

14: LDPH -- INTEGER
On entry:
the first dimension of the array PH as declared in the
(sub)program from which F02XEF is called.
Constraint: if M >= N and WANTQ and WANTP are .TRUE., LDPH
>= max(1,N).

15: RWORK(*) -- DOUBLE PRECISION array
Note: the length of RWORK must be at least max(1,lrwork),
where lrwork must satisfy:
lrwork=2*(min(M,N)-1) when
NCDLB = 0 and WANTQ and WANTP are .FALSE.,
lrwork=3*(min(M,N)-1) when
either NCOLB = 0 and WANTQ = .FALSE. and WANTP = .TRUE., or WANTP = .FALSE. and one or both of NCOLB > 0 and WANTQ = .TRUE.

\[ lrwork = 5 \times (\min(M,N) - 1) \]
otherwise.

On exit: RWORK(\min(M,N)) contains the total number of iterations taken by the QR algorithm.

The rest of the array is used as workspace.

16: CWORK(*) -- COMPLEX(KIND(1.0D)) array Workspace
Note: the length of CWORK must be at least \( \max(1,lcwork) \), where \( lcwork \) must satisfy:

\[ lcwork = \begin{cases} 
N + \max(N, NCOLB) & \text{when } M \geq N \text{ and } \text{WANTQ and WANTP are both .TRUE.} \\
N + \max(N + N, NCOLB) & \text{when } M \geq N \text{ and } \text{WANTQ = .TRUE., but WANTP = .FALSE.} \\
N + \max(N, NCOLB) & \text{when } M \geq N \text{ and } \text{WANTQ = .FALSE.} \\
M + M & \text{when } M < N \text{ and } \text{WANTP = .TRUE.} \\
M & \text{when } M < N \text{ and } \text{WANTP = .FALSE.}
\end{cases} \]

17: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL=-1
One or more of the following conditions holds:
\[ M < 0, \]
N < 0,
LDA < M,
NCOLB < 0,
LDB < M and NCOLB > 0,
LDQ < M and M < N and WANTQ = .TRUE.,
LDPH < N and M >= N and WANTQ = .TRUE. and WANTP = .TRUE..

IFAIL > 0
The QR algorithm has failed to converge in 50*min(m,n) iterations. In this case SV(1), SV(2),..., SV(IFAIL) may not have been found correctly and the remaining singular values may not be the smallest. The matrix A will nevertheless have been factorized as A=QEP where the leading min(m,n) by min(m,n) part of E is a bidiagonal matrix with SV(1), SV(2),..., SV(min(m,n)) as the diagonal elements and RWORK(1), RWORK(2),..., RWORK(min(m,n)-1) as the super-diagonal elements.

This failure is not likely to occur.

7. Accuracy

The computed factors Q, D and P satisfy the relation

\[ QDP = A+E, \]

where

\[ \|E\| \leq c(\epsilon) \|A\|, \]

(\(\epsilon\)) being the machine precision, \(c\) is a modest function of \(m\) and \(n\) and \(\|\cdot\|\) denotes the spectral (two) norm. Note that \(\|A\| = sv\).

8. Further Comments

Following the use of this routine the rank of A may be estimated by a call to the INTEGER FUNCTION F06KLF(*). The statement:
IRANK = F06KLF(MIN(M, N), SV, 1, TOL)

returns the value (k-1) in IRANK, where k is the smallest integer for which SV(k)<tol*SV(1), where tol is the tolerance supplied in TOL, so that IRANK is an estimate of the rank of S and thus also of A. If TOL is supplied as negative then the machine precision is used in place of TOL.

9. Example

9.1. Example 1

To find the singular value decomposition of the 5 by 3 matrix

\[
A = \begin{pmatrix}
0.5i & -0.5+1.5i & -1.0+1.0i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

H

together with the vector Q b for the vector

\[
b = \begin{pmatrix}
-0.55+1.05i \\
0.49+0.93i \\
0.56-0.16i \\
0.39+0.23i \\
1.13+0.83i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

9.2. Example 2

To find the singular value decompostition of the 3 by 5 matrix

\[
A = \begin{pmatrix}
0.5i & 0.4-0.3i & 0.4 & 0.3+0.4i & 0.3i \\
-0.5-1.5i & 0.9-1.3i & -0.4-0.4i & 0.1-0.7i & 0.3-0.3i \\
-1.0-1.0i & 0.2-1.4i & 1.8 & 0.0 & -2.4i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagEigenPackage (NAGF02)

Exports:
- f02aaf
- f02abf
- f02adf
- f02aef
- f02aff
- f02agf
- f02ajf
- f02akf
- f02awf
- f02axf
- f02bbf
- f02bjf
- f02fjf
- f02fjf
- f02wef
- f02xef

— package NAGF02 NagEigenPackage —

)abbrev package NAGF02 NagEigenPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:20 1994
++ Description:
++ This package uses the NAG Library to compute
++ eigenvalues and eigenvectors of a matrix
++ eigenvalues and eigenvectors of generalized matrix
++ eigenvalue problems
++ singular values and singular vectors of a matrix.

NagEigenPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==> with
  f02aaf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
  ++ f02aaf(ia,n,a,ifail)
  ++ calculates all the eigenvalue.
  ++ See \downlink{Manual Page}{manpageXXf02aaf}.
  f02abf : (Matrix DoubleFloat,Integer,Integer,Integer,Integer) -> Result
  ++ f02abf(a,ia,n,iv,ifail)
  ++ calculates all the eigenvalues of a real
  ++ symmetric matrix.
  ++ See \downlink{Manual Page}{manpageXXf02abf}.
  f02adf : (Integer,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
**f02adf** \( (i_a, i_b, n, a, b, ifail) \)
Calculates all the eigenvalues of \( Ax = \lambda B x \), where \( A \) is a real symmetric matrix and \( B \) is a real symmetric positive-definite matrix.
See \link{Manual Page}{manpageXXf02adf}.

**f02aef** : \((\text{Integer}, \text{Integer}, \text{Integer}, \text{Integer}, \_ \text{Matrix} \text{ DoubleFloat}, \text{Matrix} \text{ DoubleFloat}, \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues of \( Ax = \lambda B x \), where \( A \) is a real symmetric matrix and \( B \) is a real symmetric positive-definite matrix.
See \link{Manual Page}{manpageXXf02aef}.

**f02aff** : \((\text{Integer}, \text{Integer}, \text{Matrix} \text{ DoubleFloat}, \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues of a real unsymmetric matrix.
See \link{Manual Page}{manpageXXf02aff}.

**f02agf** : \((\text{Integer}, \text{Integer}, \text{Integer}, \text{Integer}, \_ \text{Matrix} \text{ DoubleFloat}, \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues of a real unsymmetric matrix.
See \link{Manual Page}{manpageXXf02agf}.

**f02ajf** : \((\text{Integer}, \text{Integer}, \text{Integer}, \text{Matrix} \text{ DoubleFloat}, \_ \text{Matrix} \text{ DoubleFloat}, \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues.
See \link{Manual Page}{manpageXXf02ajf}.

**f02akf** : \((\text{Integer}, \text{Integer}, \text{Integer}, \text{Integer}, \_ \text{Integer}, \_ \text{Matrix} \text{ DoubleFloat}, \_ \text{Matrix} \text{ DoubleFloat}, \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues of a complex matrix.
See \link{Manual Page}{manpageXXf02akf}.

**f02awf** : \((\text{Integer}, \text{Integer}, \text{Integer}, \text{Matrix} \text{ DoubleFloat}, \_ \text{Matrix} \text{ DoubleFloat}, \_ \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues of a complex Hermitian matrix.
See \link{Manual Page}{manpageXXf02awf}.

**f02axf** : \((\_ \text{Matrix} \text{ DoubleFloat}, \_ \text{Matrix} \text{ DoubleFloat}, \_ \text{Integer}, \_ \text{Integer}, \_ \text{Integer}, \_ \text{Integer}, \_ \text{Integer}, \_ \text{Integer}) \rightarrow \text{Result}\)
Calculates all the eigenvalues of a complex Hermitian matrix.
See \link{Manual Page}{manpageXXf02axf}.

**f02bbf** : \((\text{Integer}, \text{Integer}, \text{DoubleFloat}, \_ \text{DoubleFloat}, \_ \text{Integer}, \_ \text{Integer}, \text{Matrix} \text{ DoubleFloat}, \text{Integer}) \rightarrow \text{Result}\)
Calculates selected eigenvalues of a real matrix.
See \link{Manual Page}{manpageXXf02bbf}.
++ symmetric matrix by reduction to tridiagonal form, bisection and
++ inverse iteration, where the selected eigenvalues lie within a
++ given interval.
++ See \downlink{Manual Page}{manpageXXf02bbf}.

f02bjf : (Integer,Integer,Integer,DoubleFloat,_
      Boolean,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ f02bjf(n,ia,ib,eps1,matv,iv,a,b,ifail)
++ calculates all the eigenvalues and, if required, all the
++ eigenvectors of the generalized eigenproblem \(Ax=(\lambda)Bx\)
++ where A and B are real, square matrices, using the QZ algorithm.
++ See \downlink{Manual Page}{manpageXXf02bjf}.

f02fjf : (Integer,Integer,DoubleFloat,Integer,_
      Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,_
      Integer,Union(fn:FileName,fp:Asp27(DOT)),_
      Union(fn:FileName,fp:Asp28(IMAGE))) -> Result
++ f02fjf(n,k,tol,novecs,nrx,lwork,lrwork,
++ liwork,m,niots,x,ifail,dot,image)
++ finds eigenvalues of a real sparse symmetric
++ or generalized symmetric eigenvalue problem.
++ See \downlink{Manual Page}{manpageXXf02fjf}.

f02fjf : (Integer,Integer,DoubleFloat,Integer,_
      Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,_
      Integer,Union(fn:FileName,fp:Asp27(DOT)),_
      Union(fn:FileName,fp:Asp28(IMAGE)),FileName) -> Result
++ f02fjf(n,k,tol,novecs,nrx,lwork,lrwork,
++ liwork,m,niots,x,ifail,dot,image,monit)
++ finds eigenvalues of a real sparse symmetric
++ or generalized symmetric eigenvalue problem.
++ See \downlink{Manual Page}{manpageXXf02fjf}.

f02wef : (Integer,Integer,Integer,Integer,_
      Integer,Boolean,Integer,Boolean,Integer,Matrix DoubleFloat,_
      Matrix DoubleFloat,Integer) -> Result
++ f02wef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldpt,a,b,ifail)
++ returns all, or part, of the singular value decomposition
++ of a general real matrix.
++ See \downlink{Manual Page}{manpageXXf02wef}.

f02xef : (Integer,Integer,Integer,Integer,_
      Integer,Boolean,Integer,Boolean,Integer,Matrix Complex DoubleFloat,_
      Matrix Complex DoubleFloat,Integer) -> Result
++ f02xef(m,n,lda,ncolb,ldb,wantq,ldq,wantp,ldph,a,b,ifail)
++ returns all, or part, of the singular value decomposition
++ of a general complex matrix.
++ See \downlink{Manual Page}{manpageXXf02xef}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Boolean)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Complex DoubleFloat)
import AnyFunctions1(DoubleFloat)

f02aaf(iaArg:Integer,nArg:Integer,aArg:Matrix DoubleFloat,_
  ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
  "f02aaf",_
  ["r":S,"e":S]$Lisp,_,
  ,["e":S,"n":S]$Lisp]$Lisp,_,
  ["integer":S,"ia":S,"n":S,"ifail":S]$Lisp_]
  $Lisp,_
  ["r":S,"a":S,"ifail":S]$Lisp,_
  [[iaArg::Any,nArg::Any,ifailArg::Any,aArg::Any ]]
  @List Any$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

f02abf(aArg:Matrix DoubleFloat,iaArg:Integer,nArg:Integer,_
  ivArg:Integer,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
  "f02abf",_
  ["r":S,"v":S,"e":S]$Lisp,_,
  ["double":S,"a":S,"ia":S,"n":S]$Lisp_,
  ["e":S,"n":S]$Lisp]$Lisp,_,
  ]$Lisp,_
  ["r":S,"v":S,"ifail":S]$Lisp,_
  [[iaArg::Any,nArg::Any,ivArg::Any,ifailArg::Any,aArg::Any ]]
  @List Any$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

f02adf(iaArg:Integer,ibArg:Integer,nArg:Integer,_
  aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
  ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
  "f02adf",_
  ["r":S,"de":S]$Lisp,_,
  ["double":S,"a":S,"ib":S,"n":S,$Lisp,_,
  ["r":S,"a":S,"ifail":S]$Lisp,_,
  ["integer":S,"ia":S,"ib":S,ifail":S]
  ]$Lisp,_,
  ["r":S,"a":S,"ifail":S]$Lisp,_
  [[iaArg::Any,ibArg::Any,nArg::Any,ifailArg::Any,aArg::Any,bArg::Any ]]
  @List Any$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result
f02aef(iaArg:Integer,ibArg:Integer,nArg:Integer,_
  ivArg:Integer,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
  ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "f02aef",_
    ,"e"::S]$Lisp_
    ,["r"::S,"v"::S,"dl"::S,"e"::S]$Lisp_
    ,["double"::S,"r"::S,"n"::S]$Lisp,
      ["v"::S,"iv"::S,"n"::S]$Lisp_
      ,["a"::S,"ia"::S,"n"::S]$Lisp,
        ["b"::S,"ib"::S,"n"::S]$Lisp,_
        ["dl"::S,"n"::S]$Lisp,
          ["e"::S,"n"::S]$Lisp_
        ]$Lisp_
        ,["integer"::S,"ia"::S,"n"::S,"ifail"::S_
          @$List Any\)$Lisp_

pretend List (Record(key:Symbol,entry:Any))$Result

f02aff(iaArg:Integer,nArg:Integer,aArg:Matrix DoubleFloat,_
  ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "f02aff",_
    ,["double"::S,"rr"::S,"n"::S]$Lisp,
        ["ri"::S,"n"::S]$Lisp_
        ,["a"::S,"ia"::S,"n"::S]$Lisp,
          ["b"::S,"ib"::S,"n"::S]$Lisp,_
          ["integer"::S,"ia"::S,"n"::S,"ifail"::S_
            @$List Any\)$Lisp_

pretend List (Record(key:Symbol,entry:Any))$Result

f02agf(iaArg:Integer,nArg:Integer,ivrArg:Integer,_
  iviArg:Integer,aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"f02agf",_
,"a":S]$Lisp,_,
["double":S,["rr":S,"n":S]$Lisp,["ri":S,"n":S]$Lisp_,
["a":S,"ia":S,"n":S]$Lisp]$Lisp_,
,["integer":S,"ia":S,"n":S,"ivr":S,"ivi":S_,
,["integer":S,"n":S]$Lisp,"ifail":S]$Lisp_]
$Lisp_,
[[[iarArg::Any,nArg::Any,ivrArg::Any,iviArg::Any,_
ifailArg::Any,aArg::Any ]]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f02ajf(iarArg:Integer,iaiArg:Integer,nArg:Integer,_
arArg:Matrix DoubleFloat,aiArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"f02ajf",_
,"ar":S,"ai":S,"integer":S_]
$Lisp,_,
["rr":S,"ri":S,"integer":S]$Lisp,_,
["double":S,["rr":S,"n":S]$Lisp,["ri":S,"n":S]$Lisp_,
,["integer":S,"n":S]$Lisp]$Lisp)$Lisp_,
[[[iarArg::Any,iaiArg::Any,nArg::Any,ifailArg::Any,_
arArg::Any,aiArg::Any ]]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f02akf(iarArg:Integer,iaiArg:Integer,nArg:Integer,_
ivrArg:Integer,iviArg:Integer,arArg:Matrix DoubleFloat,_
aiArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"f02akf",_
,"ai":S,"integer":S]$Lisp,_,
["double":S,["rr":S,"n":S]$Lisp,["ri":S,"n":S]$Lisp_,
["integer":S,"n":S]$Lisp]$Lisp)$Lisp_,
[[[iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any,_
ifailArg::Any,aArg::Any ]]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
[(invokeNagman(NIL$Lisp, 
"f02awf", _
"wk1":S,"wk2":S, "wk3":S]$Lisp_, _
[r":S,"wk1":S,"wk2":S,"wk3":S]$Lisp_, _
["double":S,"iar":S,"n":S]$Lisp_, _
["ai":S,"iai":S,"n":S]$Lisp_, _
["r":S,"wk1":S,"wk2":S,"wk3":S]$Lisp_, _
["double":S,"iar":S,"n":S]$Lisp_, _
[iar::Any, iai::Any, n::Any, ifail::Any, ar::Any, ai::Any, 
pre pretend List (Record(key:Symbol,entry:Any)) ]_ 
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))

f02axf(ar:Matrix DoubleFloat, iar:Integer, ai:Matrix DoubleFloat, 
[(invokeNagman(NIL$Lisp, 
"f02axf", _
[iar::S,"iar":S,"n":S,"ivr":S,"ivi":S,_ 
,"wk1":S,"wk2":S,"wk3":S]$Lisp_, _
["double":S,"iar":S,"n":S]$Lisp_, _
[iar::Any, iai::Any, n::Any, ifail::Any, ar::Any, ifail::Any, _ 
ai::Any ]_ 
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))

```
[[[iarArg::Any,iaiArg::Any,nArg::Any,ivrArg::Any,iviArg::Any,_
   ifailArg::Any,arArg::Any,aiArg::Any ]]]
"@List Any"$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f02bbf(iaArg:Integer,nArg:Integer,albArg:DoubleFloat,_
ubArg:DoubleFloat,mArg:Integer,ivArg:Integer,_
aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "f02bbf",
  ["ia"::S,"n"::S,"alb"::S,"ub"::S,"m"::S_,
  ","e"::S,"e2"::S,"x"::S,"g"::S,"c"::S_]
)$Lisp,_
  [$Lisp])pretend List (Record(key:Symbol,entry:Any))$Result

f02bjf(nArg:Integer,iaArg:Integer,ibArg:Integer,_
eps1Arg:DoubleFloat,matvArg:Boolean,ivArg:Integer,_
aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "f02bjf",
  ","a"::S,"b"::S]$Lisp,_
  ["double"::S,"eps1"::S,["alfr"::S,"n"::S]$Lisp_ ,
  ["alfi"::S,"n"::S]$Lisp,["beta"::S,"n"::S]$Lisp_ ,
  ["b"::S,"ib"::S,"n"::S]$Lisp_ ]$Lisp,_
  ["double"::S,"eps1"::S,"matv"::S]$_Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
```
f02fjf(nArg:Integer,kArg:Integer,tolArg:DoubleFloat,_
    novecsArg:Integer,nrxArg:Integer,lworkArg:Integer,_
    lrworkArg:Integer,mArg:Integer,_
    noitsArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
    dotArg:Union(fn:FileName,fp:Asp27(DOT)),_
    imageArg:Union(fn:FileName,fp:Asp28(IMAGE))): Result ==
pushFortranOutputStack(dotFilename := aspFilename "dot")$FOP
if dotArg case fn
    then outputAsFortran(dotArg.fn)
    else outputAsFortran(dotArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(imageFilename := aspFilename "image")$FOP
if imageArg case fn
    then outputAsFortran(imageArg.fn)
    else outputAsFortran(imageArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
outputAsFortran()$Asp29(MONIT)
popFortranOutputStack()$FOP
[(invokeNagman([dotFilename,imageFilename,monitFilename]$Lisp,_
    "f02fjf",_
    "lwork"::S,"lrwork"::S,"m"::S,"noits"::S,_
    "work"::S,"rwork"::S,"iwork"::S_
    ]$Lisp,_
    "monit"::S]$Lisp,_
    ]$Lisp_
    ["iwork"::S,"liwork"::S]$Lisp]$Lisp]$_Lisp_
    ["nArg::Any,kArg::Any,tolArg::Any,novecsArg::Any,nrxArg::Any,_
    lworkArg::Any,lrworkArg::Any,iworkArg::Any,liworkArg::Any,_
    noitsArg::Any,ifailArg::Any,xArg::Any ]}_
@List Any]$_Lisp$_Lisp_
pretend List (Record(key:Symbol,entry:Any)))$Result
imageArg: Union(fn: FileName, fp: Asp28(IMAGE)), monitArg: FileName): Result ==
pushFortranOutputStack(dotFilename := aspFilename "dot")$FOP
if dotArg case fn
  then outputAsFortran(dotArg.fn)
  else outputAsFortran(dotArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(imageFilename := aspFilename "image")$FOP
if imageArg case fn
  then outputAsFortran(imageArg.fn)
  else outputAsFortran(imageArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
outputAsFortran(monitArg)
[(invokeNagman([dotFilename, imageFilename, monitFilename]$Lisp,
  "f02fjf",
  ["n":S,"k":S,"tol":S,"novecs":S,"nrx":S,
  "lwork":S,"lrwork":S,"liwork":S,"m":S,"noits":S,
  "work":S,"rwork":S,"iwork":S
  ]$Lisp,
  ["d":S,"work":S,"iwork":S,"dot":S,"image":S,
  "monit":S]$Lisp,
  ["double":S,"tol":S,"d":S,"k":S]$Lisp,
  ["x":S,"nrx":S,"k":S]$Lisp,["work":S,"lwork":S]$Lisp,
  ]$Lisp,
  ["integer":S,"n":S,"k":S,"novecs":S,"nrx":S,
  "lwork":S,"lrwork":S,"liwork":S,"m":S,"noits":S,"ifail":S,
  ["iwork":S,"liwork":S]$Lisp]$Lisp
]$Lisp,
["d":S,"m":S,"noits":S,"x":S,"ifail":S]$Lisp,
[[nArg::Any,kArg::Any,tolArg::Any,novecsArg::Any,nrxArg::Any,
  lworkArg::Any,lrworkArg::Any,liworkArg::Any,mArg::Any,
  noitsArg::Any,ifailArg::Any,xArg::Any ]]@List Any]$Lisp)
pretend List (Record(key: Symbol, entry: Any))$Result

f02wef(mArg: Integer, nArg: Integer, ldaArg: Integer,
aArg: Matrix DoubleFloat, bArg: Matrix DoubleFloat, ifailArg: Integer): Result ==
workLength : Integer :=
mArg >= nArg =>
  wantqArg and wantpArg =>
\[
\max(\max(nArg^2 + 5*(nArg - 1),nArg + ncolbArg),4)
\]
\[
wantqArg => \max(\max(nArg^2 + 4*(nArg - 1),nArg + ncolbArg),4)
\]
\[
wantpArg => \max(\max(3*(nArg - 1),2),\max(5*(nArg - 1),2)\max(3*(nArg - 1),2)
\]
\[
wantqArg and wantpArg => \max(mArg^2 + 5*(mArg - 1),2)
\]
\[
wantqArg => \max(3*(mArg - 1),1)
\]
\[
wantpArg => \max(3*(mArg - 1),1)
\]
\[
\]
\[
\text{-- This segment added by hand, to deal with an assumed size array GDN}
\]
\[
tem : Integer := (\min(mArg,nArg) - 1)
\]
\[
rLen : Integer := 2*tem
\]
zero? ncolbArg and wantpArg and not wantqArg => 3*tem
not wantpArg =>
ncolbArg >0 or wantqArg => 3*tem
5*tem
cLen : Integer :=
mArg >= nArg =>
wantqArg and wantpArg => 2*(nArg + max(nArg**2,ncolbArg))
wantqArg and not wantpArg => 2*(nArg + max(nArg**2+nArg,ncolbArg))
2*(nArg + max(nArg,ncolbArg))
wantpArg => 2*(mArg**2 + mArg)
2*mArg
svLength : Integer :=
min(mArg,nArg)
[(invokeNagman(NIL$Lisp,_{
  "f02xef",_,
   "b"::S,"cwork"::S]$Lisp,_,
  [["double"::S,["sv"::S,svLength]$Lisp,["rwork"::S,rlen]$Lisp_,
    ["integer"::S,["m"::S,"n"::S,"lda"::S,"ncolb"::S_,
    ["logical"::S,"wantq"::S,"wantp"::S]$Lisp_,
    ["double complex"::S,["q"::S,"ldq"::S,"m"::S]$Lisp_,
    [["double",["sv"::S,svLength]$Lisp,["rwork"::S,rlen]$Lisp_]
      $Lisp_],
    ["integer",["m"::S,"n"::S,"lda"::S,"ncolb"::S_,
    ["logical",["wantq"::S,"wantp"::S]$Lisp_],
    ["double complex",["q"::S,"ldq"::S,"m"::S]$Lisp_],
  $Lisp_],
  pretend List (Record(key:Symbol,entry:Any))]
) Result

— NAGF02.dotabb —

"NAGF02" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF02"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGE02" -> "ALIST"
package NAGE02 NagFittingPackage

--- NagFittingPackage.input ---

)set break resume
/sys rm -f NagFittingPackage.output
/spool NagFittingPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 215
)show NagFittingPackage
--R
--R NagFittingPackage is a package constructor
--R Abbreviation for NagFittingPackage is NAGE02
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGE02
--R
--R------------------------------- Operations --------------------------------
--R e02adf : (Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R e02aef : (Integer,Matrix(DoubleFloat),DoubleFloat,Integer) -> Result
--R e02agf : (Integer,Integer,Integer,DoubleFloat,DoubleFloat,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,Integer) -> Result
--R e02ahf : (Integer,DoubleFloat,DoubleFloat,Matrix(DoubleFloat),Integer,Integer,Integer,Integer,Integer) -> Result
--R e02ajf : (Integer,DoubleFloat,DoubleFloat,Matrix(DoubleFloat),Integer,Integer,DoubleFloat,Integer,Integer,Integer) -> Result
--R e02akf : (Integer,DoubleFloat,DoubleFloat,Matrix(DoubleFloat),Integer,Integer,DoubleFloat,Integer) -> Result
--R e02baf : (Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R e02bbf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),DoubleFloat,Integer) -> Result
--R e02bcf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),DoubleFloat,Integer,Integer) -> Result
--R e02bdf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R e02bef : (String,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),DoubleFloat,Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Matrix(DoubleFloat)) -> Result
--R e02daf : (Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,Integer,DoubleFloat,Matrix(DoubleFloat),Integer) -> Result
--R e02dcf : ... -> Result
--R e02ddf : (String,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),DoubleFloat,Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R e02def : (Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R e02dff : (Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,Integer) -> Result
--R e02gaf : (Integer,Integer,Integer,DoubleFloat,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R e02zaf : (Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,Integer) -> Result

-- S 2 of 215
)showArrayValues true
--R
--R

)clear all
showScalarValues true

m := 11

kplus1 := 4

nrows := 50

x: Matrix SF :=
[[1.00 ,2.10 ,3.10 ,3.90 ,4.90 ,5.80 ,_
  6.50 ,7.10 ,7.80 ,8.40 ,9.00 ],
  6.50 ,7.10 ,7.80 ,8.40 ,9.00 ]]
y: Matrix SF :=
[[10.40, 7.90, 4.70, 2.50, 1.20, 2.20, 5.10, 9.20, 16.10, 24.50, 35.30],
  10.399999999999999, 7.9000000000000004, 4.6999999999999993, 2.5, 1.2,
  2.2000000000000002, 5.0999999999999996, 9.1999999999999993,
  16.100000000000001, 24.5, 35.299999999999997]
Type: Matrix(DoubleFloat)

w: Matrix SF :=
[[1.00, 1.00, 1.00, 1.00, 1.00, 0.80, 0.80, 0.70, 0.50, 0.30, 0.20],
  1., 1., 1., 1., 1., 0.80000000000000004, 0.80000000000000004,
  0.69999999999999996, 0.5, 0.29999999999999999, 0.20000000000000001]
Type: Matrix(DoubleFloat)

result := e02adf(m,kplus1,nrows,x,y,w,-1)

)clear all
\[ n_{\text{plus1}} := 5 \]

\[ a : \text{Matrix} \text{SF:}= \begin{bmatrix} 2.0000 & 0.5000 & 0.2500 & 0.1250 & 0.0625 \end{bmatrix} \]

\[ x_{\text{cap}} := -1.0 \]

\[ \text{result} := \text{e02aef}(n_{\text{plus1}}, a, x_{\text{cap}}, -1) \]

\( \text{)clear all} \)

\[ \text{showArrayValues true} \]

\[ 1 \text{ true} \]

\[ \text{showScalarValues true} \]

\[ 2 \text{ true} \]
---S 19 of 215
m:=5
---R
---R
---R (3) 5
---R
---E 19
---E: PositiveInteger

---S 20 of 215
kplus1:=5
---R
---R
---R (4) 5
---R
---E 20
---E: PositiveInteger

---S 21 of 215
nrows:=6
---R
---R
---R (5) 6
---R
---E 21
---E: PositiveInteger

---S 22 of 215
xmin:=0.0
---R
---R
---R (6) 0.0
---R
---E 22
---E: Float

---S 23 of 215
xmax:=4.0
---R
---R
---R (7) 4.0
---R
---E 23
---E: Float

---S 24 of 215
x:Matrix SF:= [[0.5, 1.0, 2.0, 2.5, 3.0]]
---R
---R
---R (8) [0.5, 1.0, 2.0, 2.5, 3.0]
---R
---E 24
---E: Matrix(DoubleFloat)

---S 25 of 215
y:Matrix SF:= [[0.03, -0.75, -1.0, -0.1, 1.75]]
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>$\begin{bmatrix} 2.9999999999999999E-2 &amp; -0.75 &amp; -1. &amp; -9.9999999999999992E-2 &amp; 1.75 \end{bmatrix}$ Type: Matrix(DoubleFloat)</td>
</tr>
<tr>
<td>10</td>
<td>$\begin{bmatrix} 1. &amp; 1. &amp; 1. &amp; 1. &amp; 1. \end{bmatrix}$ Type: Matrix(DoubleFloat)</td>
</tr>
<tr>
<td>11</td>
<td>2 Type: PositiveInteger</td>
</tr>
<tr>
<td>12</td>
<td>$\begin{bmatrix} 0. &amp; 4. \end{bmatrix}$ Type: Matrix(DoubleFloat)</td>
</tr>
<tr>
<td>13</td>
<td>$\begin{bmatrix} 1. &amp; -2. &amp; 9. &amp; 0. &amp; 0. &amp; 0. &amp; 0. &amp; 0. &amp; 0. &amp; 0. &amp; 0. \end{bmatrix}$ Type: Matrix(DoubleFloat)</td>
</tr>
<tr>
<td>14</td>
<td>15 Type: PositiveInteger</td>
</tr>
</tbody>
</table>

**Note:** The table represents a set of mathematical operations and calculations involving matrices. Each step is described with the corresponding matrix representation and type.
ip:Matrix Integer:= [[1, 0]]  
  \[ \begin{bmatrix} 1 & 0 \end{bmatrix} \]  
  Type: Matrix(Integer)  

lwrk:=200  
\[ 200 \]  
Type: PositiveInteger  

liwrk:=12  
\[ 12 \]  
Type: PositiveInteger  

result:=e02agf(m,kplus1,nrows,xmin,xmax,x,y,w,mf,xf,yf,lyf,ip,lwrk,liwrk, -1)  
\[ \text{result} \]  

)clear all  

showArrayValues true  
\[ \text{true} \]  
Type: Boolean  

showScalarValues true  
\[ \text{true} \]  
Type: Boolean  

np1:=7  
\[ 7 \]
--R  (3) 7
--R Type: PositiveInteger
--E 37

--S 38 of 215
xmin:=-0.5
--R
--R
--R  (4) - 0.5
--R Type: Float
--E 38

--S 39 of 215
xmax:=2.5
--R
--R
--R  (5) 2.5
--R Type: Float
--E 39

--S 40 of 215
a:Matrix SF:= 
[[2.53213, 1.13032, 0.27150, 0.04434, _
  0.00547, 0.00054, 0.00004 ]]

--R
--R
--R  (6)
--R [ ]
--R [2.53213, 1.13032, 0.27149999999999996, 4.4339999999999997E-2,
--R  5.4699999999999992E-3, 5.399999999999999E-4, 3.9999999999999996E-5]
--R ]
--R Type: Matrix(DoubleFloat)
--E 40

--S 41 of 215
ia1:=1
--R
--R
--R  (7) 1
--R Type: PositiveInteger
--E 41

--S 42 of 215
la:=7
--R
--R
--R  (8) 7
--R Type: PositiveInteger
--E 42

--S 43 of 215
iadif1:=1
--R
--R
--R (9) 1
--R  Type: PositiveInteger
--E 43

--S 44 of 215
ladif:=7
--R
--R
--R (10) 7
--R  Type: PositiveInteger
--E 44

--S 45 of 215
-- result:=e02ahf(np1,xmin,xmax,a,ia1,la,iadif1,ladif, -1)
--E 45
)
clear all

--S 46 of 215
showArrayValues true
--R
--R
--R (1) true
--R  Type: Boolean
--E 46

--S 47 of 215
showScalarValues true
--R
--R
--R (2) true
--R  Type: Boolean
--E 47

--S 48 of 215
np1:=7
--R
--R
--R (3) 7
--R  Type: PositiveInteger
--E 48

--S 49 of 215
xmin:=-0.5
--R
--R
--R (4) - 0.5
--R
--E 49

--S 50 of 215
xmax:=2.5
--R
--R
--R (5) 2.5
--R
--E 50

--S 51 of 215
a:Matrix SF:=
[[2.53213, 1.13032, 0.27150, 0.04434, 0.00547, 0.00054, 0.00004]]
--R
--R
--R (6)
--R [ ]
--R [2.53213, 1.13032, 0.27149999999999996, 4.4339999999999997E-2,
--R 5.4699999999999992E-3, 5.3999999999999999E-4, 3.9999999999999999E-5]
--R ]
--R
--E 51

--S 52 of 215
ia1:=1
--R
--R
--R (7) 1
--R
--E 52

--S 53 of 215
la:=7
--R
--R
--R (8) 7
--R
--E 53

--S 54 of 215
qatm1:=0.0
--R
--R
--R (9) 0.0
--R
--E 54

--S 55 of 215
qatm1:=0.0
--R
--R
--R (9) 0.0
--R
--E 54
iaint1:=1  
\text{Type: PositiveInteger}

laint:=8  
\text{Type: PositiveInteger}

\text{result:=e02ajf(np1,xmin,xmax,a,ia1,la,qatm1,iaint1,laint, -1)}

np1:=7  
\text{Type: PositiveInteger}

xmin:=-0.5
1912

CHAPTER 15. CHAPTER N

--R (4) - 0.5
--E 61

--S 62 of 215
xmax:=2.5
--R
--R
--R (5) 2.5
--R
--E 62

--S 63 of 215
a:Matrix SF:=
[[2.53213, 1.13032, 0.27150, 0.04434, 0.00547, 0.00054, 0.00004]]
--R
--R
--R (6)
--R [
--R [2.53213, 1.13032, 0.27149999999999996, 4.4339999999999997E-2,
--R 5.4699999999999992E-3, 5.399999999999999E-4, 3.9999999999999996E-5]
--R ]
--E 63

--S 64 of 215
ia1:=1
--R
--R
--R (7) 1
--R
--E 64

--S 65 of 215
la:=7
--R
--R
--R (8) 7
--R
--E 65

--S 66 of 215
x:=-0.5
--R
--R
--R (9) - 0.5
--R
--E 66

--S 67 of 215
-- result:=e02akf(np1,xmin,xmax,a,ia1,la,x, -1)
--E 67
)
clear all

--S 68 of 215
showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 68

--S 69 of 215
showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 69

--S 70 of 215
m:=14
--R
--R (3) 14
--R Type: PositiveInteger
--E 70

--S 71 of 215
ncap7:=12
--R
--R (4) 12
--R Type: PositiveInteger
--E 71

--S 72 of 215
x:Matrix SF:=
[[0.20 , 0.47 , 0.74 , 1.09 , 1.60 , 1.90 , 2.60 , 3.10 , 4.00 , 5.15 ,
  R 6.17 , 8.00 , 10.00 , 12.00 ]]
--R
--R (5)
--R [
--R [0.20000000000000001, 0.46999999999999997, 0.73999999999999999,
--R 1.08999999999999999, 1.6000000000000001, 1.89999999999999999,
--R 2.5999999999999996, 3.0999999999999996, 4., 5.1500000000000004,
--R 6.1699999999999999, 8., 10., 12.]]
```
--R ]
--R Type: Matrix(DoubleFloat)
--E 72

--S 73 of 215
y:Matrix SF:=[
[0.00 ,2.00 ,4.00 ,6.00 ,8.00 ,8.62 ,9.10 ,8.90 ,
8.15 ,7.00 ,6.00 ,4.54 ,3.39 ,2.56]
--R
--R ]
--R Type: Matrix(DoubleFloat)

--S 74 of 215
w:Matrix SF:=[
[0.20 ,0.20 ,0.30 ,0.70 ,0.90 ,1.00 ,
1.00 ,1.00 ,0.80 ,0.50 ,0.70 ,1.00 ,1.00 ,1.00]
--R
--R ]
--R Type: Matrix(DoubleFloat)

--S 75 of 215
lambda:Matrix SF:=[
[0.0 ,0.0 ,0.0 ,1.50 ,2.60 ,4.00 ,8.00 ,0.0 ,0.0 ,0.0 ,0.0 ,1.50 ,2.60 ,4.00 ,8.00 ,0.0 ,0.0 ,0.0 ,0.0]
--R
--R ]
--R Type: Matrix(DoubleFloat)

result:=e02baf(m,ncap7,x,y,w,lamda, -1)
```

)clear all
showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean

ncap7:=11

(3) 11

Type: PositiveInteger

lambda:Matrix SF:=

[1.00 , 1.00 , 1.00 , 1.00 , 3.00 , 6.00 , 8.00 , 9.00 , 9.00 , 9.00 , 9.00 ]

(4) [1. 1. 1. 3. 6. 8. 9. 9. 9. 9.]

Type: Matrix(DoubleFloat)

c:Matrix SF:=

[1.00 , 2.00 , 4.00 , 7.00 , 6.00 , 4.00 , 3.00 , 0.00 , 0.00 , 0.00 , 0.00 ]

(5) [1. 2. 4. 7. 6. 4. 3. 0. 0. 0. 0.]

Type: Matrix(DoubleFloat)

x:=2.0

(6) 2.0
--R
--E 82

--S 83 of 215
-- result:=e02bbf(ncap7, lamda, c, x, -1)
--E 83

)clear all

--S 84 of 215
showArrayValues true
--R
--R
--R (1) true
--R
--E 84

--S 85 of 215
showScalarValues true
--R
--R
--R (2) true
--R
--E 85

--S 86 of 215
ncap7:=14
--R
--R
--R (3) 14
--R
--E 86

--S 87 of 215
lamda:Matrix SF:=
[ [0.0, 0.0, 0.0, 0.0, 1.0, 3.0, 3.0, 3.0, 4.0, 4.0, 6.0, 6.0, 6.0, 6.0] ]
--R
--R
--R (4) [0. 0. 0. 1. 3. 3. 4. 4. 6. 6. 6. 6.]
--R
--E 87

--S 88 of 215
c:Matrix SF:=
[ [10.0, 12.0, 13.0, 15.0, 22.0, 26.0, 24.0, 18.0, 14.0, 12.0, 0.0, 0.0, 0.0, 0.0] ]
--R
--R
--R (5) [10. 12. 13. 15. 22. 24. 18. 14. 12. 0. 0. 0. 0.]
--R

Type: Matrix(DoubleFloat)

--E 88

--S 89 of 215
x:=2.0
--R

--R

Type: Float

--E 89

--S 90 of 215
left:=1
--R

--R

Type: PositiveInteger

--E 90

--S 91 of 215
result:=e02bcf(ncap7,lamda,c,x,left, -1)
--E 91

)clear all

--S 92 of 215
showArrayValues true
--R

--R

Type: Boolean

--E 92

--S 93 of 215
showScalarValues true
--R

--R

Type: Boolean

--E 93

--S 94 of 215
ncap7:=14
--R

--R

Type: PositiveInteger

--E 94
\begin{verbatim}
--S 95 of 215
lambda:Matrix SF:=
[[0.0 ,0.00 ,0.00 ,1.00 ,3.00 ,3.00 ,3.00 ,4.00 ,4.00 ,6.00 ,6.00, 6.00, 6.00, 6.00]]
--R
--R
--R (4) [0. 0. 0. 0. 1. 3. 3. 3. 4. 4. 6. 6. 6. 6.]
--R
--R Type: Matrix(DoubleFloat)
--E 95

--S 96 of 215
c:Matrix SF:=
[[10.00 ,12.00 ,13.00 ,15.00 ,22.00 ,26.00 ,24.00 ,18.00 ,14.00 ,12.00 ,0.00 ,0.00, 0.00, 0.00]]
--R
--R
--R
--R Type: Matrix(DoubleFloat)
--E 96

--S 97 of 215
result:=e02bdf(ncap7,lambda,c, -1)
--E 97

)clear all

--S 98 of 215
displayArrayValues true
--R
--R
--R (1) true
--R
--R Type: Boolean
--E 98

--S 99 of 215
displayScalarValues true
--R
--R
--R (2) true
--R
--R Type: Boolean
--E 99

--S 100 of 215
clear c
--R
--R
--R (3) "c"
--R
--R Type: String
\end{verbatim}
--E 100
--S 101 of 215
m:=15
--R
--R
--R (4) 15
--R Type: PositiveInteger
--E 101

--S 102 of 215
x:Matrix SF:=
[ [0.00 ,0.50 ,1.00 ,1.50 ,2.00 ,2.50 ,3.00 ,
  4.00 ,4.50 ,5.00 ,5.50 ,6.00 ,7.00 ,7.50 ,8.00 ] ]
--R
--R
--R (5) [0. 0.5 1. 1.5 2. 2.5 3. 4. 4.5 5. 5.5 6. 7. 7.5 8. ]
--R Type: Matrix(DoubleFloat)
--E 102

--S 103 of 215
y:Matrix SF:=
[ [-1.1 , -0.372 , 0.431 , 1.69 , 2.11 , 3.10 , 4.23 , 4.35 , 4.81 ,
  4.61 , 4.79 , 5.23 , 6.35 , 7.19 , 7.97 ] ]
--R
--R
--R (6)
--R [ ]
--R Type: Matrix(DoubleFloat)
--E 103

--S 104 of 215
w:Matrix SF:=
[ [1.00 , 2.00 , 1.50 , 1.00 , 3.00 , 1.00 , 0.50 ,
  1.00 , 2.00 , 2.50 , 1.00 , 3.00 , 1.00 , 2.00 , 1.00 ] ]
--R
--R
--R (7) [1. 2. 1.5 1. 3. 1. 0.5 1. 2. 2.5 1. 3. 1. 2. 1. ]
--R Type: Matrix(DoubleFloat)
--E 104

--S 105 of 215
s:=1.0
--R
nest:=54  \hspace{1cm} \text{Type: PositiveInteger}

lwrk:=1105  \hspace{1cm} \text{Type: PositiveInteger}

n:=0  \hspace{1cm} \text{Type: NonNegativeInteger}

lamda:=new(1,54,0.0)$Matrix DoubleFloat

ifail:=-1  \hspace{1cm} \text{Type: Integer}
--S 111 of 215
wrk:=new(1,1105,0.0)$Matrix DoubleFloat
--R
--R (14)
--R [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
iwrk := new(1,54,0)$Matrix Integer

result := e02bef(start, m, x, y, w, s, nest, lwrk, n, lamda, ifail, wrk, iwrk)

clear all

showArrayValues true

result := e02bef(start, m, x, y, w, s, nest, lwrk, n, lamda, ifail, wrk, iwrk)

clear all
showScalarValues true

(2) true

Type: Boolean

m:=30

(3) 30

Type: PositiveInteger

px:=8

(4) 8

Type: PositiveInteger

py:=10

(5) 10

Type: PositiveInteger

x:Matrix SF:=

[[-0.52, -0.61, 0.93, 0.09, 0.88, -0.70, 1, 1, 0.3, -0.77, -0.23, -1, -0.26, -0.83, 0.22, 0.89, -0.80, -0.88, 0.68, -0.14, 0.67, -0.90, -0.84, 0.84, 0.15, -0.91, -0.35, -0.16, -0.35, -1]]
\[ y: \text{Matrix SF} := \begin{bmatrix} 0.60, -0.95, 0.84, 0.17, -0.87, 1, 0.1, 0.24, -0.77, _- \_0.32, 1, -0.63, -0.66, 0.93, 0.15, 0.99, -0.54, 0.44, -0.72, _- \\ 0.63, -0.40, 0.20, 0.43, 0.28, -0.24, 0.86, -0.41, -0.05, -1 \end{bmatrix} \]

\[ f: \text{Matrix SF} := \begin{bmatrix} 0.93, -1.79, 0.36, 0.52, 0.49, -1.76, 0.33, 0.48, 0.65, _- \\ -1.82, 0.92, 1, 8.88, -2.01, 0.47, 0.49, 0.84, -2.42, _- \\ 0.47, 7.15, 0.44, -3.34, 2.78, 0.44, 0.70, -6.52, 0.66, _- \\ 2.32, 1.66, -1 \end{bmatrix} \]
w: Matrix SF :=
\[
\begin{bmatrix}
10 , & 10 , & 10 , & 10 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 \[8pt]
1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 , & 1 \[8pt]
\end{bmatrix}
\]
Type: Matrix(DoubleFloat)

mu: Matrix SF :=
\[
\begin{bmatrix}
0 , & 0 , & 0 , & -0.5 , & 0.00 , & 0 , & 0 , & 0 , & 0 \[8pt]
\end{bmatrix}
\]
Type: Matrix(DoubleFloat)

point: Matrix Integer :=
\[
\begin{bmatrix}
3 , & 6 , & 4 , & 5 , & 7 , & 10 , & 8 , & 9 , & 11 , & 13 , & 12 , & 15 , & 14 , & 18 , & 16 , & 17 , & 19 , & 20 , & 21 , & 30 , & 23 , & 26 , & 24 , & 25 , & 27 , & 28 , & 20 , & 22 , & 1 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 \[8pt]
0 , & 29 , & 0 , & 0 , & 2 , & 22 , & 1 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 , & 0 \[8pt]
\end{bmatrix}
\]
Type: Matrix(Integer)

npoint := 43
Type: PositiveInteger

nc := 24
Type: PositiveInteger
nws:=1750

eps:=0.000001

lamda:Matrix SF:= [[0, 0, 0, 0, 0, 0, 0, 0]]

result:=e02daf(m,px,py,x,y,f,w,mu,point,npoint,nc,nws,eps,lamda,-1)

showArrayValues true

showScalarValues true

)clear all
---E 132
---S 133 of 215
start:="c"
---R
---R
---R (3) "c"
---R
---E 133

---S 134 of 215
mx:=11
---R
---R
---R (4) 11
---R
---R
---E 134

---S 135 of 215
x:Matrix SF:= [[0 ,0.5 ,1 ,1.5 ,2 ,2.5 ,3 ,3.5 ,4 ,4.5 ,5 ]]
---R
---R
---R (5) [0. 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5.]
---R
---R
---E 135

---S 136 of 215
my:=9
---R
---R
---R (6) 9
---R
---R
---E 136

---S 137 of 215
y:Matrix SF:= [[0 ,0.5 ,1 ,1.5 ,2 ,2.5 ,3 ,3.5 ,4 ]]
---R
---R
---R (7) [0. 0.5 1. 1.5 2. 2.5 3. 3.5 4.]
---R
---R
---E 137

---S 138 of 215
f:Matrix SF:=
[[1 ,0.88758 ,0.5403 ,0.070737 ,-0.41515 ,-0.80114 ,
  -0.97999 , -0.93446 , -0.65664 , 1.5 , 1.3564 , 0.82045 ,
  0.10611 , -0.62422 , -1.2317 , -1.485 , -1.3047 , -0.98547 ,
  2.06 ,1.7552 ,1.0806 ,0.15147 , -0.83229 , -1.6023 ,
  -1.97 , -1.8729 , -1.4073 ,2.57 ,2.124 ,1.3508 ,0.17684 ,
  -1.0404 , -2.0029 , -2.475 , -2.3511 , -1.6741 ,3 ,2.6427 ,
  -1.0404 , -2.0029 , -2.475 , -2.3511 , -1.6741 ,3 ,2.6427 ,]
```
1.6309 ,0.21221 , -1.2484 , -2.2034 , -2.97 , -2.8094 , -1.9809 , 3.5 , 3.1715 , 1.8611 , 0.24458 , -1.4565 , -2.864 , -3.265 , -3.2776 , -3.96 , -3.7958 , -2.6146 , 4.5 , 3.9391 , 2.4314 , 0.31632 , -1.8627 , -3.6351 , -4.455 , -4.2141 , -2.9314 , 5.04 , 4.3879 , 2.7515 , 0.35369 , -2.0707 , -4.0057 , -4.97 , -4.6823 , -3.2382 , 5.05 , 4.8367 , 2.9717 , 0.38505 , -2.2888 , -4.4033 , -5.445 , -5.1405 , -3.595 , 6 , 5.2755 , -3.2418 , 0.42442 , -2.4769 , -4.8169 , -5.93 , -5.6387 , -3.9319 ]]
```
nxest:=15

Type: PositiveInteger

nyest:=13

Type: PositiveInteger

lwrk:=592

Type: PositiveInteger

liwrk:=51

Type: PositiveInteger

nx:=0

Type: NonNegativeInteger

lambda:Matrix SF:=new(1,15,0.0)$Matrix SF

Type: Matrix(DoubleFloat)
ny:=0

(16) 0

Type: NonNegativeInteger

mu:Matrix SF:=new(1,13,0.0)$Matrix SF

(17) [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]

Type: Matrix(DoubleFloat)

wrk:Matrix SF:=new(1,592,0.0)$Matrix SF

(18) [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]
PACKAGE NAGE02 NAGFITTINGPACKAGE
--R
0.,
--R
0.,
--R
0.,
--R
0.,
--R
0.,
--R
0.,
--R
]
--R
--E 148

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

1931
0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,
0.]

0.,
0.,
0.,
0.,
0.,

0.,
0.,
0.,
0.,
0.,

Type: Matrix(DoubleFloat)

--S 149 of 215
iwrk:Matrix Integer:=new(1,51,0)$Matrix Integer
--R
--R
--R
(19)
--R
[
--R
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
--R
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
--R
0, 0, 0]
--R
]
--R
Type: Matrix(Integer)
--E 149
--S 150 of 215
-- result:=e02dcf(start,mx,x,my,y,f,s,nxest,nyest,lwrk,liwrk,nx,_
-lamda,ny,mu,wrk,iwrk,-1)
--E 150
)clear all

--S 151 of 215
showArrayValues true
--R
--R
--R
(1) true
--R
--E 151
--S 152 of 215
showScalarValues true
--R
--R
--R
(2) true
--R
--E 152
--S 153 of 215
start:="c"
--R

Type: Boolean

Type: Boolean


1932

CHAPTER 15. CHAPTER N

---
---
--- (3) "c"
---
--- Type: String
--- E 153
---
--- S 154 of 215
m:=30
---
--- Type: PositiveInteger
--- E 154
---
--- S 155 of 215
x:Matrix SF:=
[[11.16 ,12.85 ,19.85 ,19.72 ,15.91 ,0 ,20.87 ,3.45 ,
  14.26 ,17.43 ,22.8 ,7.58 ,25 ,0 ,9.66 ,5.22 ,17.25 ,25 ,12.13 ,22.23 ,
  11.52 ,15.2 ,7.54 ,17.32 ,2.14 ,0.51 ,22.69 ,5.47 ,21.67 ,3.31 ]]
---
--- Type: Matrix(DoubleFloat)
--- E 155
---
--- S 156 of 215
y:Matrix SF:=
[[1.24 ,3.06 ,10.72 ,1.39 ,7.74 ,20 ,20 ,12.78 ,17.87 ,3.46 ,12.39 ,
  1.98 ,11.87 ,0 ,20 ,14.66 ,19.57 ,3.87 ,10.79 ,6.21 ,8.53 ,0 ,10.69 ,
  13.78 ,15.03 ,8.37 ,19.63 ,17.13 ,14.36 ,0.33 ]]
---
--- Type: Matrix(DoubleFloat)
--- E 155
---
--- S 157 of 215

---
--- [1.24, 3.0599999999999996, 10.719999999999999, 1.3899999999999999,
--- 7.740000000000002, 20., 20., 12.779999999999999, 17.869999999999999,
--- 3.46, 12.390000000000001, 1.98, 11.869999999999999, 0., 20., 14.66,
--- 19.57, 3.870000000000001, 10.789999999999999, 6.21, 8.5299999999999994,
--- 0., 10.69, 13.779999999999999, 15.029999999999999, 8.3699999999999992,
--- 19.629999999999999, 17.129999999999999, 14.359999999999999,
--- 0.3299999999999996]
---
--- ]
--R

Type: Matrix(DoubleFloat)

--E 156

--S 157 of 215

f:Matrix SF:=[
[22.15, 22.11, 7.97, 16.83, 15.30, 34.6, 5.74, 41.24, 10.74, 18.60, 
 5.47, 29.87, 4.4, 58.2, 4.73, 40.36, 6.43, 8.74, 13.71, 10.25, 
 15.74, 21.6, 19.31, 12.11, 53.1, 49.43, 3.25, 28.63, 5.52, 44.08 ]]

--R

--S 158 of 215

w:Matrix SF:=[
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

--R

--S 159 of 215

s:=10

--R

--R (9) 10

Type: PositiveInteger

--E 159

--S 160 of 215

nxest:=14

--R
nyest := 14

liwrk := 11016

liwrk := 128

nx := 0

lambda := new(1, 14, 0.0)$Matrix SF

ny := 0
mu := new(1,14,0.0)$Matrix SF

wrk := new(1,11016,0.0)$Matrix SF;

result := e02ddf(start, m, x, y, f, w, s, nxest, nyest, lwrk, liwrk, nx, lamda, ny, mu, wrk, -1)

)clear all

m := 7
px:=11
--R
--R
--R (4) 11
--R Type: PositiveInteger
--E 173

--S 174 of 215
py:=10
--R
--R
--R (5) 10
--R Type: PositiveInteger
--E 174

--S 175 of 215
x:Matrix SF:= [[1, 1.1, 1.5, 1.6, 1.9, 1.9, 2]]
--R
--R
--R (6)
--R [ ]
--R [1., 1.1000000000000001, 1.5, 1.6000000000000001, 1.8999999999999999, 1.8999999999999999, 2.]
--R ]
--R Type: Matrix(DoubleFloat)
--E 175

--S 176 of 215
y:Matrix SF:= [[0, 0.1, 0.7, 0.4, 0.3, 0.8, 1]]
--R
--R
--R (7)
--R [ ]
--R [0., 0.10000000000000001, 0.69999999999999996, 0.40000000000000002, 0.29999999999999999, 0.80000000000000004, 1.]
--R ]
--R Type: Matrix(DoubleFloat)
--E 176

--S 177 of 215
lambda:Matrix SF:= [[1.0, 1.0, 1.0, 1.0, 1.0, 1.3, 1.6, 2, 2, 2, 2]]
--R
--R
--R (8)
--R [1. 1. 1. 1. 1.2999999999999998 1.5 1.6000000000000001 2. 2. 2. 2.]
--R ]
--R Type: Matrix(DoubleFloat)
--E 177

--S 178 of 215
mu:Matrix SF:= [[0, 0, 0, 0.4, 0.7, 1, 1, 1, 1]]
---R
---R
---R (9)
---R [0., 0., 0., 0.4000000000000002, 0.6999999999999996, 1., 1., 1., 1.]
---R Type: Matrix(DoubleFloat)
---E 178
---S 179 of 215
---R c:Matrix SF:=
[ 1 , 1.1333 , 1.3667 , 1.7 , 1.9 , 2. , 1.2 , 1.3333 , 1.5667 , 1.9 ,
 2.1 , 2.2 , 1.5833 , 1.7167 , 1.95 , 2.2833 , 2.4833 , 2.5833 ,
 2.1433 , 2.2767 , 2.51 , 2.8433 , 3.0433 , 3.1433 , 2.8667 ,
 3 , 3.2333 , 3.5667 , 3.7667 , 3.8667 , 3.4667 , 3.6 , 3.8333 ,
 4.1667 , 4.3667 , 4.4667 , 4.7 , 4.9 , 5 ]
---R
---R
---R (10)
---R [1. , 1.1333, 1.3666999999999998, 1.7, 1.8999999999999999, 2., 1.2,
---R 1.3332999999999999, 1.5666999999999999, 1.8999999999999999, 2.0999999999999996,
---R 2.0999999999999996, 2.2 , 1.5832999999999999, 1.7166999999999999, 1.95,
---R 2.2832999999999997, 2.4832999999999998, 2.5832999999999999, 2.1433,
---R 2.7666999999999999, 2.5832999999999999, 2.8433000000000002, 2.8666999999999999,
---R 3.0432999999999999, 3.1433, 2.8666999999999999, 3., 3.2332999999999998,
---R 3.5667, 3.7667000000000002, 3.8666999999999999, 3.4666999999999999,
---R 4.3666999999999999, 4.4666999999999994, 4. , 4.1333000000000002,
---R 4.3666999999999999, 4.6999999999999993, 4.9000000000000004, 5.]
---R
---R Type: Matrix(DoubleFloat)
---E 179
---S 180 of 215
---R result:=e02df(m,px,py,x,y,lambda,mu,c,-1)
---E 180

)clear all

---S 181 of 215
showArrayValues true
---R
---R
---R (1) true
---R Type: Boolean
---E 181

---S 182 of 215
showScalarValues true
---R
CHAPTER 15. CHAPTER N

--R
--R (2) true
--R
--E 182

--S 183 of 215
mx:=7
--R
--R
--R (3) 7
--R
--E 183

--S 184 of 215
my:=6
--R
--R
--R (4) 6
--R
--E 184

--S 185 of 215
px:=11
--R
--R
--R (5) 11
--R
--E 185

--S 186 of 215
py:=10
--R
--R
--R (6) 10
--R
--E 186

--S 187 of 215
x:Matrix SF:= [[1 ,1.1 ,1.3 ,1.4 ,1.5 ,1.7 ,2 ]]
--R
--R
--R (7)
--R [[]1.,1.10000000000000001,1.2999999999999999,1.3999999999999999,1.5,1.7,2.]]
--R
--E 187

--S 188 of 215
y:Matrix SF:= [[0 ,0.2 ,0.4 ,0.6 ,0.8 ,1 ]]
--R
--R
--R


```
--R (8)
--R [  
--R [0., 0.20000000000000001, 0.40000000000000002, 0.59999999999999998,  
--R 0.80000000000000004, 1.]  
--R ]
--R Type: Matrix(DoubleFloat)
--E 188

--S 189 of 215
lamda:Matrix SF:= [[1, 1, 1, 1.3, 1.5, 1.6, 2, 2, 2]]
--R
--R (9)
--R [1. 1. 1. 1.2999999999999998 1.5 1.6000000000000001 2. 2. 2. 2.]
--R Type: Matrix(DoubleFloat)
--E 189

--S 190 of 215
mu:Matrix SF:= [[0, 0, 0, 0.4, 0.7, 1, 1, 1, 1]]
--R
--R (10)
--R [0. 0. 0. 0.40000000000000002 0.6999999999999996 1. 1. 1. 1.]
--R Type: Matrix(DoubleFloat)
--E 190

--S 191 of 215
c:Matrix SF:=  
[[1, 1.1333, 1.3667, 1.7, 1.9, 2, 1.2, 1.3333, 1.5667, 1.9,  
  2.1, 2.2, 1.5833, 1.7167, 1.95, 2.2833, 2.4833, 2.5833, 2.1433, 2.2767,  
  2.51, 2.8433, 3.0433, 3.1433, 3.8667, 3.6, 3.8333, 4.1667, 4.4667, 4.6999999999999996,  
  4.7, 4.9, 5.]]
--R
--R (11)
--R [  
--R [1., 1.1333, 1.3666999999999998, 1.7, 1.8999999999999999, 2., 1.2,  
--R 1.3332999999999999, 1.5667, 1.8999999999999999, 2.0999999999999996,  
--R 2.20000000000000002, 1.5832999999999999, 1.7166999999999999, 1.95,  
--R 2.2832999999999999, 2.4832999999999999, 2.5832999999999999, 2.1433,  
--R 2.2766999999999999, 2.5099999999999999, 2.8433000000000002,  
--R 3.0432999999999999, 3.1433, 3.2332999999999999, 3.7667, 3.9, 3.8333, 4.1667, 4.4667, 4.6999999999999996,  
--R 4.7, 4.9, 5.]]
--R
--R]
--R Type: Matrix(DoubleFloat)
--E 191
```
result := e02dff(mx, my, px, py, x, y, lambda, mu, c, lwrk, liwrk, -1)

showArrayValues true

showScalarValues true

m := 5

la := 7
nplus2 := 5

a : Matrix DoubleFloat :=
[[1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[exp(0.2), exp(-0.2), 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[exp(0.4), exp(-0.4), 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[exp(0.6), exp(-0.6), 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[exp(0.8), exp(-0.8), 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
\[
\begin{bmatrix}
4.5009999999999994 & 4.3599999999999994 & 4.3330000000000002, & 4.4179999999999993, 4.625
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)
\[
\text{lambda: Matrix SF:= } \begin{bmatrix}
0 , 0 , 0 , 1.00 , 0 , 0 , 0 , 0 , 0 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
\text{mu: Matrix SF:= } \begin{bmatrix}
0 , 0 , 0 , 0.80 , 1.20 , 0 , 0 , 0 , 0 \end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
m:=10
\]

Type: PositiveInteger

\[
x: Matrix SF:= \begin{bmatrix}
0.00 , 0.70 , 1.44 , 0.21 , 1.01 , 1.84 , 0.71 , 1.00 , 0.54 , 1.531 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
y: Matrix SF:= \begin{bmatrix}
0.77 , 1.06 , 0.33 , 0.44 , 0.50 , 0.02 , 1.95 , 1.20 , 0.04 , 0.18 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)
This package uses the NAG Library to find a function which approximates a set of data points. Typically the data contain random errors, as of experimental measurement, which need to be smoothed out. To seek an approximation to the data, it is first necessary to specify for the approximating function a mathematical form (a polynomial, for example) which contains a number of unspecified coefficients: the appropriate fitting routine then derives for the coefficients the values which provide the best fit of that particular form. The package deals mainly with curve and surface fitting (i.e., fitting with functions of one and of two variables) when a polynomial or a cubic spline is used as the fitting function, since these cover the most common needs. However, fitting with other functions and/or more variables can be undertaken by means of general linear or nonlinear routines (some of which are contained in other packages) depending on whether the coefficients in the function occur linearly or nonlinearly. Cases where a graph rather than a set of data points is given can be treated simply by first reading a suitable set of points from the graph. The package also contains routines for evaluating, differentiating and integrating polynomial and spline curves and surfaces, once the numerical values of their coefficients have been determined.
Contents of this Introduction:

1. Scope of the Chapter
2. Background to the Problems
   2.1. Preliminary Considerations
      2.1.1. Fitting criteria: norms
      2.1.2. Weighting of data points
   2.2. Curve Fitting
      2.2.1. Representation of polynomials
      2.2.2. Representation of cubic splines
   2.3. Surface Fitting
      2.3.1. Bicubic splines: definition and representation
   2.4. General Linear and Nonlinear Fitting Functions
2.5. Constrained Problems
2.6. References
3. Recommendations on Choice and Use of Routines
   3.1. General
      3.1.1. Data considerations
      3.1.2. Transformation of variables
   3.2. Polynomial Curves
      3.2.1. Least-squares polynomials: arbitrary data points
      3.2.2. Least-squares polynomials: selected data points
   3.3. Cubic Spline Curves
      3.3.1. Least-squares cubic splines
3.3.2. Automatic fitting with cubic splines

3.4. Spline Surfaces

3.4.1. Least-squares bicubic splines

3.4.2. Automatic fitting with bicubic splines

3.5. General Linear and Nonlinear Fitting Functions

3.5.1. General linear functions

3.5.2. Nonlinear functions

3.6. Constraints

3.7. Evaluation, Differentiation and Integration

3.8. Index

1. Scope of the Chapter

The main aim of this chapter is to assist the user in finding a function which approximates a set of data points. Typically the data contain random errors, as of experimental measurement, which need to be smoothed out. To seek an approximation to the data, it is first necessary to specify for the approximating function a mathematical form (a polynomial, for example) which contains a number of unspecified coefficients: the appropriate fitting routine then derives for the coefficients the values which provide the best fit of that particular form. The chapter deals mainly with curve and surface fitting (i.e., fitting with functions of one and of two variables) when a polynomial or a cubic spline is used as the fitting function, since these cover the most common needs. However, fitting with other functions and/or more variables can be undertaken by means of general linear or nonlinear routines (some of which are contained in other chapters) depending on whether the coefficients in the function occur linearly or nonlinearly. Cases where a graph rather than a set of data points is given can be treated simply by first reading a suitable set of points from the graph.

The chapter also contains routines for evaluating, differentiating and integrating polynomial and spline curves and surfaces, once the numerical values of their coefficients have been determined.

2. Background to the Problems
2.1. Preliminary Considerations

In the curve-fitting problems considered in this chapter, we have a dependent variable \( y \) and an independent variable \( x \), and we are given a set of data points \((x_r, y_r)\), for \( r = 1, 2, \ldots, m \). The preliminary matters to be considered in this section will, for simplicity, be discussed in this context of curve-fitting problems. In fact, however, these considerations apply equally well to surface and higher-dimensional problems. Indeed, the discussion presented carries over essentially as it stands if, for these cases, we interpret \( x \) as a vector of several independent variables and correspondingly each \( x_r \) as a vector containing the \( r \)th data value of each independent variable.

We wish, then, to approximate the set of data points as closely as possible with a specified function, \( f(x) \) say, which is as smooth as possible -- \( f(x) \) may, for example, be a polynomial. The requirements of smoothness and closeness conflict, however, and a balance has to be struck between them. Most often, the smoothness requirement is met simply by limiting the number of coefficients allowed in the fitting function -- for example, by restricting the degree in the case of a polynomial. Given a particular number of coefficients in the function in question, the fitting routines of this chapter determine the values of the coefficients such that the 'distance' of the function from the data points is as small as possible. The necessary balance is struck by the user comparing a selection of such fits having different numbers of coefficients. If the number of coefficients is too low, the approximation to the data will be poor. If the number is too high, the fit will be too close to the data, essentially following the random errors and tending to have unwanted fluctuations between the data points. Between these extremes, there is often a group of fits all similarly close to the data points and then, particularly when least-squares polynomials are used, the choice is clear: it is the fit from this group having the smallest number of coefficients.

The above process can be seen as the user minimizing the smoothness measure (i.e., the number of coefficients) subject to the distance from the data points being acceptably small. Some of the routines, however, do this task themselves. They use a different measure of smoothness (in each case one that is continuous) and minimize it subject to the distance being less than a threshold specified by the user. This is a much more automatic process, requiring only some experimentation with the threshold.
2.1.1. Fitting criteria: norms

A measure of the above 'distance' between the set of data points and the function \( f(x) \) is needed. The distance from a single data point \((x_r, y_r)\) to the function can simply be taken as

\[
(\epsilon) = y_r - f(x_r),
\]

and is called the residual of the point. (With this definition, the residual is regarded as a function of the coefficients contained in \( f(x) \); however, the term is also used to mean the particular value of \((\epsilon)\) which corresponds to the fitted values of the coefficients.) However, we need a measure of distance for the set of data points as a whole. Three different measures are used in the different routines (which measure to select, according to circumstances, is discussed later in this sub-section). With \((\epsilon)\) defined in (1), these measures, or \( r \) norms, are

\[
\begin{align*}
\| \epsilon \|_1 & = \sum_{r=1}^{m} |\epsilon_r|, \\
\| \epsilon \|_2 & = \sqrt{\sum_{r=1}^{m} \epsilon_r^2}, \\
\| \epsilon \|_{\infty} & = \max_{r=1}^{m} |\epsilon_r|
\end{align*}
\]

respectively the \( l_1 \) norm, the \( l_2 \) norm and the \( l_{\infty} \) norm.

Minimization of one or other of these norms usually provides the fitting criterion, the minimization being carried out with respect to the coefficients in the mathematical form used for \( f(x) \): with respect to the \( b \) for example if the mathematical form is the power series in (8) below. The fit which results from minimizing (2) is known as the \( l_1 \) fit, or the fit in the \( l_1 \) norm:
that which results from minimizing (3) is the 1-fit, the well-known least-squares fit (minimizing (3) is equivalent to minimizing the square of (3), i.e., the sum of squares of residuals, and it is the latter which is used in practice), and that from minimizing (4) is the 1-infty, or minimax, fit.

Strictly speaking, implicit in the use of the above norms are the statistical assumptions that the random errors in the y are independent of one another and that any errors in the x are negligible by comparison. From this point of view, the use of the 1 norm is appropriate when the random errors in the y have a normal distribution, and the 2 norm is appropriate when they have a rectangular distribution, as when fitting a table of values rounded to a fixed number of decimal places. The 1 norm is appropriate when the error distribution has its frequency function proportional to the negative exponential of the modulus of the normalised error -- not a common situation.

However, the user is often indifferent to these statistical considerations, and simply seeks a fit which he can assess by inspection, perhaps visually from a graph of the results. In this event, the 1 norm is particularly appropriate when the data are thought to contain some 'wild' points (since fitting in this norm tends to be unaffected by the presence of a small number of such points), though of course in simple situations the user may prefer to identify and reject these points. The 1 norm should be used only when the maximum residual is of particular concern, as may be the case for example when the data values have been obtained by accurate computation, as of a mathematical function. Generally, however, a routine based on least-squares should be preferred, as being computationally faster and usually providing more information on which to assess the results. In many problems the three fits will not differ significantly for practical purposes.

Some of the routines based on the 1 norm do not minimize the 2 norm itself but instead minimize some (intuitively acceptable) measure of smoothness subject to the norm being less than a user-specified threshold. These routines fit with cubic or bicubic
The use of the above norms also assumes that the data values \( y_r \) are of equal (absolute) accuracy. Some of the routines enable an allowance to be made to take account of differing accuracies. The allowance takes the form of 'weights' applied to the \( y \)-values so that those values known to be more accurate have a greater influence on the fit than others. These weights, to be supplied by the user, should be calculated from estimates of the absolute accuracies of the \( y \)-values, these estimates being expressed as standard deviations, probable errors or some other measure which has the same dimensions as \( y \). Specifically, for each \( y \) the corresponding weight \( w_r \) should be inversely proportional to the accuracy estimate of \( y_r \). For example, if the percentage accuracy is the same for all \( y \), then the absolute accuracy of \( y_r \) is proportional to \( y_r \) (assuming \( y_r \) to be positive, as it usually is in such cases) and so \( w_r = k/y_r \), for \( r=1,2,\ldots,m \), for an arbitrary positive constant \( k \). (This definition of weight is stressed because often weight is defined as the square of that used here.)

The norms (2), (3) and (4) above are then replaced respectively by

\[
\sum_{r=1}^{m} w_r (\epsilon_r), \quad (5)
\]

\[
\sum_{r=1}^{m} w_r^2 (\epsilon_r), \quad (6)
\]

\[
\max_{r} w_r (\epsilon_r). \quad (7)
\]
Again it is the square of (6) which is used in practice rather than (6) itself.

2.2. Curve Fitting

When, as is commonly the case, the mathematical form of the fitting function is immaterial to the problem, polynomials and cubic splines are to be preferred because their simplicity and ease of handling confer substantial benefits. The cubic spline is the more versatile of the two. It consists of a number of cubic polynomial segments joined end to end with continuity in first and second derivatives at the joins. The third derivative at the joins is in general discontinuous. The x-values of the joins are called knots, or, more precisely, interior knots. Their number determines the number of coefficients in the spline, just as the degree determines the number of coefficients in a polynomial.

2.2.1. Representation of polynomials

Rather than using the power-series form

\[ f(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_k x^k \quad (8) \]

to represent a polynomial, the routines in this chapter use the Chebyshev series form

\[ f(x) = -a_0 T_0(x) + a_1 T_1(x) + a_2 T_2(x) + \ldots + a_k T_k(x), \quad (9) \]

where \( T_i(x) \) is the Chebyshev polynomial of the first kind of degree \( i \) (see Cox and Hayes [1], page 9), and where the range of \( x \) has been normalized to run from \(-1\) to \(+1\). The use of either form leads theoretically to the same fitted polynomial, but in practice results may differ substantially because of the effects of rounding error. The Chebyshev form is to be preferred, since it leads to much better accuracy in general, both in the computation of the coefficients and in the subsequent evaluation of the fitted polynomial at specified points. This form also has other advantages: for example, since the later terms in (9) generally decrease much more rapidly from left to right than do those in (8), the situation is more often encountered where the last terms are negligible and it is obvious that the degree of the polynomial can be reduced (note that on the interval \(-1\leq x\leq 1\) for all \( i \), \( T_i(x) \) attains the value unity but never exceeds it, so
that the coefficient \( a_i \) gives directly the maximum value of the term containing it).

2.2.2. Representation of cubic splines

A cubic spline is represented in the form

\[
f(x) = c_1 N_1(x) + c_2 N_2(x) + \ldots + c_p N_p(x), \quad (10)
\]

where \( N_i(x) \), for \( i = 1, 2, \ldots, p \), is a normalised cubic B-spline (see Hayes [2]). This form, also, has advantages of computational speed and accuracy over alternative representations.

2.3. Surface Fitting

There are now two independent variables, and we shall denote these by \( x \) and \( y \). The dependent variable, which was denoted by \( y \) in the curve-fitting case, will now be denoted by \( f \). (This is a rather different notation from that indicated for the general-dimensional problem in the first paragraph of Section 2.1, but it has some advantages in presentation.)

Again, in the absence of contrary indications in the particular application being considered, polynomials and splines are the approximating functions most commonly used. Only splines are used by the surface-fitting routines in this chapter.

2.3.1. Bicubic splines: definition and representation

The bicubic spline is defined over a rectangle \( R \) in the \( (x,y) \) plane, the sides of \( R \) being parallel to the \( x \)- and \( y \)-axes. \( R \) is divided into rectangular panels, again by lines parallel to the axes. Over each panel the bicubic spline is a bicubic polynomial, that is it takes the form

\[
\begin{align*}
3 & \quad 3 \\
-- & \quad -- & \quad i & \quad j \\
> & \quad > & \quad a_{ij} \quad (13) \\
-- & \quad -- & \quad ij \\
i=0 & \quad j=0
\end{align*}
\]

Each of these polynomials joins the polynomials in adjacent panels with continuity up to the second derivative. The constant \( x \)-values of the dividing lines parallel to the \( x \)-axis form the set of interior knots for the variable \( x \), corresponding precisely to the set of interior knots of a cubic spline. Similarly, the constant \( y \)-values of dividing lines parallel to the \( x \)-axis form
the set of interior knots for the variable \( y \). Instead of representing the bicubic spline in terms of the above set of bicubic polynomials, however, it is represented, for the sake of computational speed and accuracy, in the form

\[
f(x,y) = \sum_{i=1}^{p} \sum_{j=1}^{q} c_{ij} M_i(x) N_j(y),
\]

where \( M_i(x) \) for \( i=1,2,\ldots,p \), and \( N_j(y) \) for \( j=1,2,\ldots,q \), are normalised B-splines (see Hayes and Halliday [4] for further details of bicubic splines and Hayes [2] for normalised B-splines).

2.4. General Linear and Nonlinear Fitting Functions

We have indicated earlier that, unless the data-fitting application under consideration specifically requires some other type of fitting function, a polynomial or a spline is usually to be preferred. Special routines for these functions, in one and in two variables, are provided in this chapter. When the application does specify some other fitting function, however, it may be treated by a routine which deals with a general linear function, or by one for a general nonlinear function, depending on whether the coefficients in the given function occur linearly or nonlinearly.

The general linear fitting function can be written in the form

\[
f(x) = c_1 \phi_1(x) + c_2 \phi_2(x) + \ldots + c_p \phi_p(x),
\]

where \( x \) is a vector of one or more independent variables, and the \( \phi_i \) are any given functions of these variables (though they must be linearly independent of one another if there is to be the possibility of a unique solution to the fitting problem). This is not intended to imply that each \( \phi_i \) is necessarily a function of all the variables: we may have, for example, that each \( \phi_i \) is a function of a different single variable, and even that one of the \( \phi_i \) is a constant. All that is required is that a value of each \( \phi_i(x) \) can be computed when a value of each independent variable is given.
When the fitting function \( f(x) \) is not linear in its coefficients, no more specific representation is available in general than \( f(x) \) itself. However, we shall find it helpful later on to indicate the fact that \( f(x) \) contains a number of coefficients (to be determined by the fitting process) by using instead the notation \( f(x; c) \), where \( c \) denotes the vector of coefficients. An example of a nonlinear fitting function is

\[
f(x; c) = c_1 + c_2 \exp(-c_3 x) + c_4 \exp(-c_5 x),
\]

which is in one variable and contains five coefficients. Note that here, as elsewhere in this Chapter Introduction, we use the term 'coefficients' to include all the quantities whose values are to be determined by the fitting process, not just those which occur linearly. We may observe that it is only the presence of the coefficients \( c_4 \) and \( c_5 \) which makes the form (16) nonlinear.

If the values of these two coefficients were known beforehand, (16) would instead be a linear function which, in terms of the general linear form (15), has \( p=3 \) and

\[
\phi_1(x) = 1, \quad \phi_2(x) = \exp(-c_3 x), \quad \text{and} \quad \phi_3(x) = \exp(-c_5 x).
\]

We may note also that polynomials and splines, such as (9) and (14), are themselves linear in their coefficients. Thus if, when fitting with these functions, a suitable special routine is not available (as when more than two independent variables are involved or when fitting in the 1 norm), it is appropriate to use a routine designed for a general linear function.

### 2.5. Constrained Problems

So far, we have considered only fitting processes in which the values of the coefficients in the fitting function are determined by an unconstrained minimization of a particular norm. Some fitting problems, however, require that further restrictions be placed on the determination of the coefficient values. Sometimes these restrictions are contained explicitly in the formulation of the problem in the form of equalities or inequalities which the coefficients, or some function of them, must satisfy. For example, if the fitting function contains a term \( A \exp(-kx) \), it may be required that \( k \geq 0 \). Often, however, the equality or inequality constraints relate to the value of the fitting function or its derivatives at specified values of the independent variable(s), but these too can be expressed in terms
of the coefficients of the fitting function, and it is
appropriate to do this if a general linear or nonlinear routine
is being used. For example, if the fitting function is that given
in (10), the requirement that the first derivative of the
function at \( x = x_0 \) be non-negative can be expressed as

\[
0 = c_{N} N'(x_0) + c_{N} N'(x_0) + \ldots + c_{N} N'(x_0), \quad (17)
\]

where the prime denotes differentiation with respect to \( x \) and
each derivative is evaluated at \( x = x_0 \). On the other hand, if the
0 requirement had been that the derivative at \( x = x_0 \) be exactly zero,
0 the inequality sign in (17) would be replaced by an equality.

Routines which provide a facility for minimizing the appropriate
norm subject to such constraints are discussed in Section 3.6.

2.6. References

suite of algorithms for the non-specialist user. Report

Fitting. Bull Inst Math Appl. 10 144--152.

(For definition of normalised B-splines and details of
numerical methods.)

Variable. Numerical Approximation to Functions and Data. (ed

Appl. 14 89--103.

3. Recommendations on Choice and Use of Routines

3.1. General

The choice of a routine to treat a particular fitting problem
will depend first of all on the fitting function and the norm to
be used. Unless there is good reason to the contrary, the fitting
function should be a polynomial or a cubic spline (in the
appropriate number of variables) and the norm should be the l
norm (leading to the least-squares fit). If some other function is to be used, the choice of routine will depend on whether the function is nonlinear (in which case see Section 3.5.2) or linear in its coefficients (see Section 3.5.1), and, in the latter case, on whether the $l^1$ or $l^2$ norm is to be used. The latter section is appropriate for polynomials and splines, too, if the $l^1$ norm is preferred.

In the case of a polynomial or cubic spline, if there is only one independent variable, the user should choose a spline (Section 3.3) when the curve represented by the data is of complicated form, perhaps with several peaks and troughs. When the curve is of simple form, first try a polynomial (see Section 3.2) of low degree, say up to degree 5 or 6, and then a spline if the polynomial fails to provide a satisfactory fit. (Of course, if third-derivative discontinuities are unacceptable to the user, a polynomial is the only choice.) If the problem is one of surface fitting, one of the spline routines should be used (Section 3.4). If the problem has more than two independent variables, it may be treated by the general linear routine in Section 3.5.1, again using a polynomial in the first instance.

Another factor which affects the choice of routine is the presence of constraints, as previously discussed in Section 2.5. Indeed this factor is likely to be overriding at present, because of the limited number of routines which have the necessary facility. See Section 3.6.

3.1.1. Data considerations

A satisfactory fit cannot be expected by any means if the number and arrangement of the data points do not adequately represent the character of the underlying relationship: sharp changes in behaviour, in particular, such as sharp peaks, should be well covered. Data points should extend over the whole range of interest of the independent variable(s): extrapolation outside the data ranges is most unwise. Then, with polynomials, it is advantageous to have additional points near the ends of the ranges, to counteract the tendency of polynomials to develop fluctuations in these regions. When, with polynomial curves, the user can precisely choose the $x$-values of the data, the special points defined in Section 3.2.2 should be selected. With splines the choice is less critical as long as the character of the relationship is adequately represented. All fits should be tested graphically before accepting them as satisfactory.

For this purpose it should be noted that it is not sufficient to plot the values of the fitted function only at the data values of
the independent variable(s); at the least, its values at a
similar number of intermediate points should also be plotted, as
unwanted fluctuations may otherwise go undetected. Such
fluctuations are the less likely to occur the lower the number of
coefficients chosen in the fitting function. No firm guide can be
given, but as a rough rule, at least initially, the number of
coefficients should not exceed half the number of data points
(points with equal or nearly equal values of the independent
variable, or both independent variables in surface fitting,
counting as a single point for this purpose). However, the
situation may be such, particularly with a small number of data
points, that a satisfactorily close fit to the data cannot be
achieved without unwanted fluctuations occurring. In such cases,
it is often possible to improve the situation by a transformation
of one or more of the variables, as discussed in the next
paragraph: otherwise it will be necessary to provide extra data
points. Further advice on curve fitting is given in Cox and Hayes
[1] and, for polynomials only, in Hayes [3] of Section 2.7. Much
of the advice applies also to surface fitting; see also the
Routine Documents.

3.1.2. Transformation of variables

Before starting the fitting, consideration should be given to the
choice of a good form in which to deal with each of the
variables: often it will be satisfactory to use the variables as
they stand, but sometimes the use of the logarithm, square root,
or some other function of a variable will lead to a better-
behaved relationship. This question is customarily taken into
account in preparing graphs and tables of a relationship and the
same considerations apply when curve or surface fitting. The
practical context will often give a guide. In general, it is best
to avoid having to deal with a relationship whose behaviour in
one region is radically different from that in another. A steep
rise at the left-hand end of a curve, for example, can often best
be treated by curve fitting in terms of log(x+c) with some
suitable value of the constant c. A case when such a
transformation gave substantial benefit is discussed in Hayes [3]
page 60. According to the features exhibited in any particular
case, transformation of either dependent variable or independent
variable(s) or both may be beneficial. When there is a choice it
is usually better to transform the independent variable(s): if
the dependent variable is transformed, the weights attached to
the data points must be adjusted. Thus (denoting the dependent
variable by y, as in the notation for curves) if the y to be
r
fitted have been obtained by a transformation y=g(Y) from
original data values Y, with weights W, for r=1,2,...,m, we
r
must take
\[ w = W_1 / (dy/dY), \]  
where the derivative is evaluated at \( Y \). Strictly, the transformation of \( Y \) and the adjustment of weights are valid only when the data errors in the \( Y \) are small compared with the range spanned by the \( Y \), but this is usually the case.

3.2. Polynomial Curves

3.2.1. Least-squares polynomials: arbitrary data points

E02ADF fits to arbitrary data points, with arbitrary weights, polynomials of all degrees up to a maximum degree \( k \), which is at choice. If the user is seeking only a low degree polynomial, up to degree 5 or 6 say, \( k = 10 \) is an appropriate value, providing there are about 20 data points or more. To assist in deciding the degree of polynomial which satisfactorily fits the data, the routine provides the root-mean-square-residual \( s_i \) for all degrees \( i = 1, 2, \ldots, k \). In a satisfactory case, these \( s_i \) will decrease steadily as \( i \) increases and then settle down to a fairly constant value, as shown in the example

\[
\begin{array}{cc}
  i & s_i \\
  0 & 3.5215 \\
  1 & 0.7708 \\
  2 & 0.1861 \\
  3 & 0.0820 \\
  4 & 0.0554 \\
  5 & 0.0251 \\
  6 & 0.0264 \\
  7 & 0.0280 \\
  8 & 0.0277 \\
\end{array}
\]
If the $s$ values settle down in this way, it indicates that the closest polynomial approximation justified by the data has been achieved. The degree which first gives the approximately constant value of $s$ (degree 5 in the example) is the appropriate degree to select. (Users who are prepared to accept a fit higher than sixth degree, should simply find a high enough value of $k$ to enable the type of behaviour indicated by the example to be detected: thus they should seek values of $k$ for which at least 4 or 5 consecutive values of $s$ are approximately the same.) If the degree were allowed to go high enough, $s$ would, in most cases, eventually start to decrease again, indicating that the data points are being fitted too closely and that undesirable fluctuations are developing between the points. In some cases, particularly with a small number of data points, this final decrease is not distinguishable from the initial decrease in $s$. In such cases, users may seek an acceptable fit by examining the graphs of several of the polynomials obtained. Failing this, they may (a) seek a transformation of variables which improves the behaviour, (b) try fitting a spline, or (c) provide more data points. If data can be provided simply by drawing an approximating curve by hand and reading points from it, use the points discussed in Section 3.2.2.

3.2.2. Least-squares polynomials: selected data points

When users are at liberty to choose the $x$-values of data points, such as when the points are taken from a graph, it is most advantageous when fitting with polynomials to use the values $x = \cos((\pi)r/n)$, for $r=0,1,\ldots,n$ for some value of $n$, a suitable $r$ value for which is discussed at the end of this section. Note that these $x$ relate to the variable $x$ after it has been normalised so that its range of interest is $-1$ to $+1$. E02ADF may then be used as in Section 3.2.1 to seek a satisfactory fit.

3.3. Cubic Spline Curves

3.3.1. Least-squares cubic splines

E02BAF fits to arbitrary data points, with arbitrary weights, a
cubic spline with interior knots specified by the user. The choice of these knots so as to give an acceptable fit must largely be a matter of trial and error, though with a little experience a satisfactory choice can often be made after one or two trials. It is usually best to start with a small number of knots (too many will result in unwanted fluctuations in the fit, or even in there being no unique solution) and, examining the fit graphically at each stage, to add a few knots at a time at places where the fit is particularly poor. Moving the existing knots towards these places will also often improve the fit. In regions where the behaviour of the curve underlying the data is changing rapidly, closer knots will be needed than elsewhere. Otherwise, positioning is not usually very critical and equally-spaced knots are often satisfactory. See also the next section, however.

A useful feature of the routine is that it can be used in applications which require the continuity to be less than the normal continuity of the cubic spline. For example, the fit may be required to have a discontinuous slope at some point in the range. This can be achieved by placing three coincident knots at the given point. Similarly a discontinuity in the second derivative at a point can be achieved by placing two knots there. Analogy with these discontinuous cases can provide guidance in more usual cases: for example, just as three coincident knots can produce a discontinuity in slope, so three close knots can produce a rapid change in slope. The closer the knots are, the more rapid can the change be.

Figure 1
Please see figure in printed Reference Manual

An example set of data is given in Figure 1. It is a rather tricky set, because of the scarcity of data on the right, but it will serve to illustrate some of the above points and to show some of the dangers to be avoided. Three interior knots (indicated by the vertical lines at the top of the diagram) are chosen as a start. We see that the resulting curve is not steep enough in the middle and fluctuates at both ends, severely on the right. The spline is unable to cope with the shape and more knots are needed.

In Figure 2, three knots have been added in the centre, where the data shows a rapid change in behaviour, and one further out at each end, where the fit is poor. The fit is still poor, so a further knot is added in this region and, in Figure 3, disaster ensues in rather spectacular fashion.

Figure 2
Please see figure in printed Reference Manual
The reason is that, at the right-hand end, the fits in Figure 1 and Figure 2 have been interpreted as poor simply because of the fluctuations about the curve underlying the data (or what it is naturally assumed to be). But the fitting process knows only about the data and nothing else about the underlying curve, so it is important to consider only closeness to the data when deciding goodness of fit.

Thus, in Figure 1, the curve fits the last two data points quite well compared with the fit elsewhere, so no knot should have been added in this region. In Figure 2, the curve goes exactly through the last two points, so a further knot is certainly not needed here.

Figure 4 shows what can be achieved without the extra knot on each of the flat regions. Remembering that within each knot interval the spline is a cubic polynomial, there is really no need to have more than one knot interval covering each flat region.

What we have, in fact, in Figure 2 and Figure 3 is a case of too many knots (so too many coefficients in the spline equation) for the number of data points. The warning in the second paragraph of Section 2.1 was that the fit will then be too close to the data, tending to have unwanted fluctuations between the data points. The warning applies locally for splines, in the sense that, in localities where there are plenty of data points, there can be a lot of knots, as long as there are few knots where there are few points, especially near the ends of the interval. In the present example, with so few data points on the right, just the one extra knot in Figure 2 is too many! The signs are clearly present, with the last two points fitted exactly (at least to the graphical accuracy and actually much closer than that) and fluctuations within the last two knot-intervals (cf. Figure 1, where only the final point is fitted exactly and one of the wobbles spans several data points).

The situation in Figure 3 is different. The fit, if computed exactly, would still pass through the last two data points, with even more violent fluctuations. However, the problem has become so ill-conditioned that all accuracy has been lost. Indeed, if the last interior knot were moved a tiny amount to the right, there would be no unique solution and an error message would have
been caused. Near-singularity is, sadly, not picked up by the routine, but can be spotted readily in a graph, as Figure 3. B-spline coefficients becoming large, with alternating signs, is another indication. However, it is better to avoid such situations, firstly by providing, whenever possible, data adequately covering the range of interest, and secondly by placing knots only where there is a reasonable amount of data.

The example here could, in fact, have utilised from the start the observation made in the second paragraph of this section, that three close knots can produce a rapid change in slope. The example has two such rapid changes and so requires two sets of three close knots (in fact, the two sets can be so close that one knot can serve in both sets, so only five knots prove sufficient in Figure 4). It should be noted, however, that the rapid turn occurs within the range spanned by the three knots. This is the reason that the six knots in Figure 2 are not satisfactory as they do not quite span the two turns.

Some more examples to illustrate the choice of knots are given in Cox and Hayes [1].

3.3.2. Automatic fitting with cubic splines

E02BEF also fits cubic splines to arbitrary data points with arbitrary weights but itself chooses the number and positions of the knots. The user has to supply only a threshold for the sum of squares of residuals. The routine first builds up a knot set by a series of trial fits in the $l^1$ norm. Then, with the knot set decided, the final spline is computed to minimize a certain smoothing measure subject to satisfaction of the chosen threshold. Thus it is easier to use than E02BAF (see previous section), requiring only some experimentation with this threshold. It should therefore be first choice unless the user has a preference for the ordinary least-squares fit or, for example, wishes to experiment with knot positions, trying to keep their number down (E02BEF aims only to be reasonably frugal with knots).

3.4. Spline Surfaces

3.4.1. Least-squares bicubic splines

E02DAF fits to arbitrary data points, with arbitrary weights, a bicubic spline with its two sets of interior knots specified by the user. For choosing these knots, the advice given for cubic splines, in Section 3.3.1 above, applies here too. (See also the next section, however.) If changes in the behaviour of the surface underlying the data are more marked in the direction of
one variable than of the other, more knots will be needed for the
former variable than the latter. Note also that, in the surface
case, the reduction in continuity caused by coincident knots will
extend across the whole spline surface: for example, if three
knots associated with the variable x are chosen to coincide at a
value L, the spline surface will have a discontinuous slope
across the whole extent of the line x=L.

With some sets of data and some choices of knots, the least-
squares bicubic spline will not be unique. This will not occur,
with a reasonable choice of knots, if the rectangle R is well
covered with data points: here R is defined as the smallest
rectangle in the (x,y) plane, with sides parallel to the axes,
which contains all the data points. Where the least-squares
solution is not unique, the minimal least-squares solution is
computed, namely that least-squares solution which has the
smallest value of the sum of squares of the B-spline coefficients
\( c \) (see the end of Section 2.3.2 above). This choice of least-
squares solution tends to minimize the risk of unwanted
fluctuations in the fit. The fit will not be reliable, however,
in regions where there are few or no data points.

3.4.2. Automatic fitting with bicubic splines

E02DDF also fits bicubic splines to arbitrary data points with
arbitrary weights but chooses the knot sets itself. The user has
to supply only a threshold for the sum of squares of residuals.
Just like the automatic curve E02BEF (Section 3.3.2), E02DDF then
builds up the knot sets and finally fits a spline minimizing a
smoothing measure subject to satisfaction of the threshold.
Again, this easier to use routine is normally to be preferred, at
least in the first instance.

E02DCF is a very similar routine to E02DDF but deals with data
points of equal weight which lie on a rectangular mesh in the
(x,y) plane. This kind of data allows a very much faster
computation and so is to be preferred when applicable.
Substantial departures from equal weighting can be ignored if the
user is not concerned with statistical questions, though the
quality of the fit will suffer if this is taken too far. In such
cases, the user should revert to E02DDF.

3.5. General Linear and Nonlinear Fitting Functions

3.5.1. General linear functions

For the general linear function (15), routines are available for
fitting in the \( l_1 \) and \( l_2 \) norms. The least-squares routines (which
are to be preferred unless there is good reason to use another
norm -- see Section 2.1.1) are in Chapter F04. The 1
routine is 1
E02GAF.

All the above routines are essentially linear algebra routines,
and in considering their use we need to view the fitting process
in a slightly different way from hitherto. Taking y to be the
dependent variable and x the vector of independent variables, we
have, as for equation (1) but with each x now a vector,

\[ \epsilon_r = y - f(x) \quad r = 1, 2, \ldots, m. \]

Substituting for \( f(x) \) the general linear form (15), we can write
this as

\[ c_1 \phi_1(x) + c_2 \phi_2(x) + \cdots + c_p \phi_p(x) = y - \epsilon_r, \quad r = 1, 2, \ldots, m. \]

Thus we have a system of linear equations in the coefficients \( c_j \).

Usually, in writing these equations, the \( \epsilon_r \) are omitted
and simply taken as implied. The system of equations is then
described as an overdetermined system (since we must have \( m \geq p \) if
there is to be the possibility of a unique solution to our
fitting problem), and the fitting process of computing the \( c_j \) to
minimize one or other of the norms (2), (3) and (4) can be
described, in relation to the system of equations, as solving the
overdetermined system in that particular norm. In matrix
notation, the system can be written as

\[ (\Phi)c = y, \]

where \((\Phi)\) is the \( m \) by \( p \) matrix whose element in row \( r \) and
column \( j \) is \( \phi_j(x_r) \), for \( r = 1, 2, \ldots, m; j = 1, 2, \ldots, p \). The vectors
\( c \) and \( y \) respectively contain the coefficients \( c_j \) and the data
\( y_r \).

The routines, however, use the standard notation of linear
algebra, the overdetermined system of equations being denoted by
The correspondence between this notation and that which we have used for the data-fitting problem (equation (20)) is therefore given by

\[ A = \Phi, \quad x = c, \quad b = y \]  

(22)

Note that the norms used by these routines are the unweighted norms (2) and (3). If the user wishes to apply weights to the data points, that is to use the norms (5) or (6), the equivalences (22) should be replaced by

\[ A = D\Phi, \quad x = c, \quad b = Dy \]

where \( D \) is a diagonal matrix with \( w_r \) as the \( r \)th diagonal element. Here \( w_r \), for \( r=1,2,...,m \), is the weight of the \( r \)th data point as defined in Section 2.1.2.

3.5.2. Nonlinear functions

Routines for fitting with a nonlinear function in the \( L_2 \) norm are provided in Chapter E04, and that chapter's Introduction should be consulted for the appropriate choice of routine. Again, however, the notation adopted is different from that we have used for data fitting. In the latter, we denote the fitting function by \( f(x;c) \), where \( x \) is the vector of independent variables and \( c \) is the vector of coefficients, whose values are to be determined. The squared \( L_2 \) norm, to be minimized with respect to the elements of \( c \), is then

\[
\sum_{r=1}^{m} w_r [y_r - f(x_r;c)]^2
\]

(23)

where \( y_r \) is the \( r \)th data value of the dependent variable, \( x_r \) is the vector containing the \( r \)th values of the independent variables, and \( w_r \) is the corresponding weight as defined in Section 2.1.2.

On the other hand, in the nonlinear least-squares routines of Chapter E04, the function to be minimized is denoted by
the minimization being carried out with respect to the elements of the vector $x$. The correspondence between the two notations is given by

$$x = c \text{ and } f(x) = w[y - f(x;c)], \quad i = r = 1, 2, \ldots, m.$$ 

Note especially that the vector $x$ of variables of the nonlinear least-squares routines is the vector $c$ of coefficients of the data-fitting problem, and in particular that, if the selected routine requires derivatives of the $f(x)$ to be provided, these are derivatives of $w[y - f(x;c)]$ with respect to the coefficients of the data-fitting problem.

3.6. Constraints

At present, there are only a limited number of routines which fit subject to constraints. Chapter E04 contains a routine, E04UCF, which can be used for fitting with a nonlinear function in the $l_2$ norm subject to equality or inequality constraints. This routine, unlike those in that chapter suited to the unconstrained case, is not designed specifically for minimizing functions which are sums of squares, and so the function (23) has to be treated as a general nonlinear function. The E04 Chapter Introduction should be consulted.

The remaining constraint routine relates to fitting with polynomials in the $l_1$ norm. E02AGF deals with polynomial curves and allows precise values of the fitting function and (if required) all its derivatives up to a given order to be prescribed at one or more values of the independent variable.

3.7. Evaluation, Differentiation and Integration

Routines are available to evaluate, differentiate and integrate polynomials in Chebyshev-series form and cubic or bicubic splines in B-spline form. These polynomials and splines may have been
produced by the various fitting routines or, in the case of polynomials, from prior calls of the differentiation and integration routines themselves.

E02AEF and E02AKF evaluate polynomial curves: the latter has a longer parameter list but does not require the user to normalise the values of the independent variable and can accept coefficients which are not stored in contiguous locations. E02BBF evaluates cubic spline curves, and E02DEF and E02DFF bicubic spline surfaces.

Differentiation and integration of polynomial curves are carried out by E02AHF and E02AJF respectively. The results are provided in Chebyshev-series form and so repeated differentiation and integration are catered for. Values of the derivative or integral can then be computed using the appropriate evaluation routine.

For splines the differentiation and integration routines provided are of a different nature from those for polynomials. E02BCF provides values of a cubic spline curve and its first three derivatives (the rest, of course, are zero) at a given value of \( x \) spline over its whole range. These routines can also be applied to surfaces of the form (14). For example, if, for each value of \( j \) in turn, the coefficients \( c_{ij} \), for \( i=1,2,\ldots,p \) are supplied to E02BCF with \( x=x_j \) and on each occasion we select from the output the value of the second derivative, \( d_j \), say, and if the whole set of \( d_j \) are then supplied to the same routine with \( x=y_0 \), the output will contain all the values at \( (x_0,y_0) \) of

\[
\frac{2}{dd f} \quad r+2 \quad \frac{dd f}{2} \quad \frac{dd f}{2} \quad \frac{dd f}{2}
\]

Equally, if after each of the first \( p \) calls of E02BCF we had selected the function value (E02BBF would also provide this) instead of the second derivative and we had supplied these values to E02BDF, the result obtained would have been the value of

\[
\int_A^B f(x,y)dy, \quad \int_A^B f(x,y)dy, \quad \int_A^B f(x,y)dy
\]
where A and B are the end-points of the y interval over which the spline was defined.

3.8. Index

Automatic fitting,  
with bicubic splines  E02DCF  
with cubic splines  E02DDF  
Data on rectangular mesh  E02DCF  
Differentiation,  
of cubic splines  E02BCF  
of polynomials  E02AHF  
Evaluation,  
of bicubic splines  E02DEF  
of cubic splines  E02EFF  
of cubic splines and derivatives  E02BCF  
of definite integral of cubic splines  E02BDF  
of polynomials  E02AEF  
Integration,  
of cubic splines (definite integral)  E02BDF  
of polynomials  E02AJF  
Least-squares curve fit,  
with cubic splines  E02BAF  
with polynomials,  
  arbitrary data points  E02ADF  
  with constraints  E02AGF  
Least-squares surface fit with bicubic splines  E02DAF  
   fit with general linear function,  
   1  
Sorting,  
  2-D data into panels  E02ZAF  

E02 -- Curve and Surface Fitting  Contents -- E02  
Chapter E02  

Curve and Surface Fitting  
E02ADF  Least-squares curve fit, by polynomials, arbitrary data points  
E02AEF  Evaluation of fitted polynomial in one variable from Chebyshev series form (simplified parameter list)  
E02AGF  Least-squares polynomial fit, values and derivatives may be constrained, arbitrary data points,
E02AHF  Derivative of fitted polynomial in Chebyshev series form
E02AJF  Integral of fitted polynomial in Chebyshev series form
E02AKF  Evaluation of fitted polynomial in one variable, from Chebyshev series form
E02BAF  Least-squares curve cubic spline fit (including interpolation)
E02BBF  Evaluation of fitted cubic spline, function only
E02BCF  Evaluation of fitted cubic spline, function and derivatives
E02BDF  Evaluation of fitted cubic spline, definite integral
E02BEF  Least-squares cubic spline curve fit, automatic knot placement
E02DAF  Least-squares surface fit, bicubic splines
E02DCF  Least-squares surface fit by bicubic splines with automatic knot placement, data on rectangular grid
E02DDF  Least-squares surface fit by bicubic splines with automatic knot placement, scattered data
E02DEF  Evaluation of a fitted bicubic spline at a vector of points
E02DFF  Evaluation of a fitted bicubic spline at a mesh of points
E02GAF  L1-approximation by general linear function
E02ZAF  Sort 2-D data into panels for fitting bicubic splines

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
E02ADF computes weighted least-squares polynomial approximations to an arbitrary set of data points.

2. Specification

SUBROUTINE E02ADF (M, KPLUS1, NROWS, X, Y, W, WORK1, WORK2, A, S, IFAIL)
INTEGER M, KPLUS1, NROWS, IFAIL
DOUBLE PRECISION X(M), Y(M), W(M), WORK1(3*M), WORK2(2*KPLUS1), A(NROWS,KPLUS1), S(KPLUS1)

3. Description

This routine determines least-squares polynomial approximations of degrees 0,1,...,k to the set of data points \((x_r, y_r)\) with weights \(w_r\), for \(r=1,2,...,m\).

The approximation of degree \(i\) has the property that it minimizes (sigma) the sum of squares of the weighted residuals \((\epsilon_r)\),

\[
(\epsilon_r) = w_r (y_r - f_r)
\]

where

\[
(\epsilon_r) = w_r (y_r - f_r)
\]

and \(f_r\) is the value of the polynomial of degree \(i\) at the \(r\)th data point.

Each polynomial is represented in Chebyshev-series form with normalised argument \(x\). This argument lies in the range -1 to +1 and is related to the original variable \(x\) by the linear transformation

\[
(2x - x_{\min} - x_{\max})
\]

\[
x = \frac{2x - x_{\min} - x_{\max}}{x_{\max} - x_{\min}}.
\]

Here \(x_{\max}\) and \(x_{\min}\) are respectively the largest and smallest values of \(x\). The polynomial approximation of degree \(i\) is represented as
where $T_j(x)$ is the Chebyshev polynomial of the first kind of degree $j$ with argument $x$.

For $i=0,1,...,k$, the routine produces the values of $a_{i+1,j+1}$, for $j=0,1,...,i$, together with the value of the root mean square residual $s = \sqrt{\frac{1}{i} \sum_i \left( \text{sigma} \right)^2}$. In the case $m=i+1$ the routine sets $s$ to zero.

The method employed is due to Forsythe [4] and is based upon the generation of a set of polynomials orthogonal with respect to summation over the normalized data set. The extensions due to Clenshaw [1] to represent these polynomials as well as the approximating polynomials in their Chebyshev-series forms are incorporated. The modifications suggested by Reinsch and Gentleman (see [5]) to the method originally employed by Clenshaw for evaluating the orthogonal polynomials from their Chebyshev-series representations are used to give greater numerical stability.

For further details of the algorithm and its use see Cox [2] and [3].

Subsequent evaluation of the Chebyshev-series representations of the polynomial approximations should be carried out using E02AEF.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number m of data points. Constraint: M >= MDIST >= 2, where MDIST is the number of distinct x values in the data.

2: KPLUS1 -- INTEGER Input
   On entry: k+1, where k is the maximum degree required. Constraint: 0 < KPLUS1 <= MDIST, where MDIST is the number of distinct x values in the data.

3: NROWS -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which E02ADF is called. Constraint: NROWS >= KPLUS1.

4: X(M) -- DOUBLE PRECISION array Input
   On entry: the values x of the independent variable, for r=1,2,...,m. Constraint: the values must be supplied in non-decreasing order with X(M) > X(1).

5: Y(M) -- DOUBLE PRECISION array Input
   On entry: the values y of the dependent variable, for r=1,2,...,m.

6: W(M) -- DOUBLE PRECISION array Input
   On entry: the set of weights, w, for r=1,2,...,m. For advice on the choice of weights, see Section 2.1.2 of the
Chapter Introduction. Constraint: \( W(r) > 0.0 \), for \( r=1,2,\ldots,m \).

7: \texttt{WORK1(3*M)} -- DOUBLE PRECISION array
   Workspace

8: \texttt{WORK2(2*KPLUS1)} -- DOUBLE PRECISION array
   Workspace

9: \texttt{A(NROWS,KPLUS1)} -- DOUBLE PRECISION array
   Output

   On exit: the coefficients of \( T(x) \) in the approximating \( j \)
   polynomial of degree \( i \). \( A(i+1,j+1) \) contains the coefficient \( a_{i+1,j+1} \), for \( i=0,1,\ldots,k; \ j=0,1,\ldots,i \).

10: \texttt{S(KPLUS1)} -- DOUBLE PRECISION array
    Output

    On exit: \( S(i+1) \) contains the root mean square residual \( s_i \),
    for \( i=0,1,\ldots,k \), as described in Section 3. For the
    interpretation of the values of the \( s_i \) and their use in
    selecting an appropriate degree, see Section 3.1 of the
    Chapter Introduction.

11: \texttt{IFAIL} -- INTEGER
    Input/Output

    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   The weights are not all strictly positive.

IFAIL= 2
   The values of \( X(r) \), for \( r=1,2,\ldots,M \) are not in non-decreasing order.

IFAIL= 3
   All \( X(r) \) have the same value: thus the normalisation of \( X \) is not possible.

IFAIL= 4
   On entry \( KPLUS1 < 1 \) (so the maximum degree required is negative)
or \( KPLUS1 > MDIST \), where \( MDIST \) is the number of distinct \( x \) values in the data (so there cannot be a unique solution for degree \( k=KPLUS1-1 \)).

IFAIL = 5

NROWS < KPLUS1.

7. Accuracy

No error analysis for the method has been published. Practical experience with the method, however, is generally extremely satisfactory.

8. Further Comments

The time taken by the routine is approximately proportional to \( m(k+1)(k+11) \).

The approximating polynomials may exhibit undesirable oscillations (particularly near the ends of the range) if the maximum degree \( k \) exceeds a critical value which depends on the number of data points \( m \) and their relative positions. As a rough guide, for equally-spaced data, this critical value is about \( 2\sqrt{m} \). For further details see Hayes [6] page 60.

9. Example

Determine weighted least-squares polynomial approximations of degrees 0, 1, 2 and 3 to a set of 11 prescribed data points. For the approximation of degree 3, tabulate the data and the corresponding values of the approximating polynomial, together with the residual errors, and also the values of the approximating polynomial at points half-way between each pair of adjacent data points.

The example program supplied is written in a general form that will enable polynomial approximations of degrees 0, 1, \ldots, \( k \) to be obtained to \( m \) data points, with arbitrary positive weights, and the approximation of degree \( k \) to be tabulated. E02AEF is used to evaluate the approximating polynomial. The program is self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
E02 -- Curve and Surface Fitting
E02AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AEF evaluates a polynomial from its Chebyshev-series representation.

2. Specification

```fortran
SUBROUTINE E02AEF (NPLUS1, A, XCAP, P, IFAIL)
INTEGER NPLUS1, IFAIL
DOUBLE PRECISION A(NPLUS1), XCAP, P
```

3. Description

This routine evaluates the polynomial

\[
1 - a_0 T_0(x) + a_1 T_1(x) + a_2 T_2(x) + \ldots + a_n T_n(x)
\]

for any value of \( x \) satisfying \(-1 \leq x \leq 1\). Here \( T_j(x) \) denotes the \( j \)th Chebyshev polynomial of the first kind of degree \( j \) with argument \( x \). The value of \( n \) is prescribed by the user.

In practice, the variable \( x \) will usually have been obtained from an original variable \( x \), where \( x \leq x \leq x \) and

\[
\min \quad \max
\]

\[
((x-x) - (x-x))
\]

\[
\min \quad \max
\]

\[
\frac{x - \min}{\max - \min}
\]

\[
(x - \min)
\]

\[
\max \quad \min
\]

Note that this form of the transformation should be used
computationally rather than the mathematical equivalent

\[
\frac{2x - x - x}{\min \max} = \frac{4}{\epsilon}
\]

since the former guarantees that the computed value of \(x\) differs from its true value by at most \(4\epsilon\), whereas the latter has no such guarantee.

The method employed is based upon the three-term recurrence relation due to Clenshaw [1], with modifications to give greater numerical stability due to Reinsch and Gentleman (see [4]).

For further details of the algorithm and its use see Cox [2] and [3].

4. References


5. Parameters

1: NPLUS1 -- INTEGER Input
   On entry: the number \(n+1\) of terms in the series (i.e., one greater than the degree of the polynomial). Constraint: \(NPLUS1 \geq 1\).

2: A(NPLUS1) -- DOUBLE PRECISION array Input
   On entry: \(A(i)\) must be set to the value of the \(i\)th coefficient in the series, for \(i=1,2,\ldots,n+1\).

3: XCAP -- DOUBLE PRECISION Input
On entry: $x$, the argument at which the polynomial is to be evaluated. It should lie in the range $-1$ to $+1$, but a value just outside this range is permitted (see Section 6) to allow for possible rounding errors committed in the transformation from $x$ to $x$ discussed in Section 3. Provided the recommended form of the transformation is used, a successful exit is thus assured whenever the value of $x$ lies in the range $x$ to $x$.
\begin{align*}
\text{min} & \quad \text{max}
\end{align*}

4: \quad P \quad \text{DOUBLE PRECISION} \\
On exit: the value of the polynomial.

5: \quad IFAIL \quad \text{INTEGER} \\
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
\quad \text{ABS}(XCAP) > 1.0 + 4(\epsilon), \text{where } (\epsilon) \text{is the machine precision. In this case the value of } P \text{ is set arbitrarily to zero.}

IFAIL= 2
\quad \text{On entry NPLUS1 < 1.}

7. Accuracy

The rounding errors committed are such that the computed value of the polynomial is exact for a slightly perturbed set of coefficients $a + (\delta)a$. The ratio of the sum of the absolute values of the $(\delta)a$ to the sum of the absolute values of the $a$ is less than a small multiple of $(n+1)$ times machine precision.

8. Further Comments
The time taken by the routine is approximately proportional to $n+1$.

It is expected that a common use of E02AEF will be the evaluation of the polynomial approximations produced by E02ADF and E02AFF(*).

9. Example

Evaluate at 11 equally-spaced points in the interval $-1 \leq x \leq 1$ the polynomial of degree 4 with Chebyshev coefficients, 2.0, 0.5, 0.25, 0.125, 0.0625.

The example program is written in a general form that will enable a polynomial of degree $n$ in its Chebyshev-series form to be evaluated at $m$ equally-spaced points in the interval $-1 \leq x \leq 1$. The program is self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E02 -- Curve and Surface Fitting

E02AGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AGF computes constrained weighted least-squares polynomial approximations in Chebyshev-series form to an arbitrary set of data points. The values of the approximations and any number of their derivatives can be specified at selected points.

2. Specification

```
SUBROUTINE E02AGF (M, KPLUS1, NROWS, XMIN, XMAX, X, Y, W, MF, XF, YF, LYF, IP, A, S, NP1, WRK, LWRK, IWRK, LIWRK, IFAIL)
```

INTEGER M, KPLUS1, NROWS, MF, LYF, IP(MF), NP1,

LWRK, IWRK, LIWRK, IFAIL)
3. Description

This routine determines least-squares polynomial approximations of degrees up to \( k \) to the set of data points \((x_r, y_r)\) with weights \( w_r\), for \( r=1,2,\ldots,m \). The value of \( k \), the maximum degree required, is prescribed by the user. At each of the values \( x_{F_r}\), for \( r=1,2,\ldots,M_F\), of the independent variable \( x \), the approximations and their derivatives up to order \( p \) are constrained to have one of the user-specified values \( y_{F_s}\), for \( s=1,2,\ldots,n \), where \( n=M_F+p \) -- \( r \)

The approximation of degree \( i \) has the property that, subject to the imposed contraints, it minimizes \( \sum_{r=1}^{m} w_r (y_r - f(x_r))^2 \) where

\[
(\text{epsilon}) = w_r (y_r - f(x_r))^2
\]

and \( f(x_r) \) is the value of the polynomial approximation of degree \( i \) at the \( r \)th data point.

Each polynomial is represented in Chebyshev-series form with normalised argument \( x \). This argument lies in the range \(-1\) to \(+1\) and is related to the original variable \( x \) by the linear transformation

\[
2x - (x + x_{\text{max}} + x_{\text{min}}) \quad \text{max} \quad \text{min}
\]

\[
x = \frac{2x - (x + x_{\text{max}} + x_{\text{min}})}{(x_{\text{max}} - x_{\text{min}})}
\]

\[
(\text{epsilon}) = w_r (y_r - f(x_r))^2
\]
where \( x \) and \( x' \), specified by the user, are respectively the lower and upper end-points of the interval of \( x \) over which the polynomials are to be defined.

The polynomial approximation of degree \( i \) can be written as

\[
\sum_{j=0}^{i} a_{ij} T_j(x) + \ldots + a_{ij} T_0(x) + \ldots + a_{ij} T_i(x)
\]

where \( T_j(x) \) is the Chebyshev polynomial of the first kind of degree \( j \) with argument \( x \). For \( i=n, n+1, \ldots, k \), the routine produces the values of the coefficients \( a_{ij} \) for \( j=0,1,\ldots,i \), together with the value of the root mean square residual, \( S_i \), defined as

\[
\frac{1}{\sqrt{m'}}, \quad m' = \text{number of data points with non-zero weight}.
\]

Values of the approximations may subsequently be computed using E02AEF or E02AKF.

First E02AGF determines a polynomial \( (\mu)(x) \), of degree \( n-1 \), which satisfies the given constraints, and a polynomial \( (\nu)(x) \), of degree \( n \), which has value (or derivative) zero wherever a constrained value (or derivative) is specified. It then fits \( y - (\mu)(x) \), for \( r=1,2,\ldots,m \) with polynomials of the required degree in \( x \) each with factor \( (\nu)(x) \). Finally the coefficients of \( (\mu)(x) \) are added to the coefficients of these fits to give the coefficients of the constrained polynomial approximations to the
data points $\{(x_r, y_r), \text{ for } r=1,2,\ldots,m\}$. The method employed is given in Hayes [3]: it is an extension of Forsythe's orthogonal polynomials method [2] as modified by Clenshaw [1].

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number $m$ of data points to be fitted.
   Constraint: $M \geq 1$.

2: KPLUS1 -- INTEGER Input
   On entry: $k+1$, where $k$ is the maximum degree required.
   Constraint: $n+1 \leq KPLUS1 \leq m''+n$, where $n$ is the total number of constraints and $m''$ is the number of data points with non-zero weights and distinct abscissae which do not coincide with any of the $x(r)$.

3: NROWS -- INTEGER Input
   On entry:
   the first dimension of the array A as declared in the (sub)program from which E02AGF is called.
   Constraint: $NROWS \geq KPLUS1$.

4: XMIN -- DOUBLE PRECISION Input

5: XMAX -- DOUBLE PRECISION Input
   On entry: the lower and upper end-points, respectively, of the interval $[x_\min, x_\max]$. Unless there are specific reasons to the contrary, it is recommended that XMIN and XMAX be set respectively to the lowest and highest value among the $x_r$ and $x(r)$. This avoids the danger of extrapolation provided there is a constraint point or data point with non-zero weight at each end-point. Constraint: XMAX > XMIN.
6: \text{X(M)} -- DOUBLE PRECISION array \textbf{Input}
On entry: the value \(x\) of the independent variable at the \(r\)th data point, for \(r=1,2,\ldots,m\). Constraint: the \(X(r)\) must be in non-decreasing order and satisfy \(XMIN \leq X(r) \leq XMAX\).

7: \text{Y(M)} -- DOUBLE PRECISION array \textbf{Input}
On entry: \(Y(r)\) must contain \(y\), the value of the dependent \(r\) variable at the \(r\)th data point, for \(r=1,2,\ldots,m\).

8: \text{W(M)} -- DOUBLE PRECISION array \textbf{Input}
On entry: the weights \(w\) to be applied to the data points \(x\), for \(r=1,2,\ldots,m\). For advice on the choice of weights see the Chapter Introduction. Negative weights are treated as positive. A zero weight causes the corresponding data point to be ignored. Zero weight should be given to any data point whose \(x\) and \(y\) values both coincide with those of a constraint (otherwise the denominators involved in the root-mean-square residuals \(s\) will be slightly in error).

9: \text{MF} -- INTEGER \textbf{Input}
On entry: the number of values of the independent variable at which a constraint is specified. Constraint: \(MF \geq 1\).

10: \text{XF(MF)} -- DOUBLE PRECISION array \textbf{Input}
On entry: the \(r\)th value of the independent variable at which a constraint is specified, for \(r = 1,2,\ldots,MF\). Constraint: these values need not be ordered but must be distinct and satisfy \(XMIN \leq XF(r) \leq XMAX\).

11: \text{YF(LYF)} -- DOUBLE PRECISION array \textbf{Input}
On entry: the values which the approximating polynomials and their derivatives are required to take at the points specified in \(XF\). For each value of \(XF(r)\), \(YF\) contains in successive elements the required value of the approximation, its first derivative, second derivative, \ldots, \(p\) th derivative, for \(r = 1,2,\ldots,MF\). Thus the value which the \(k\)th derivative of each approximation (\(k=0\) referring to the approximation itself) is required to take at the point \(XF(r)\) must be contained in \(YF(s)\), where
\[ s=r+k+p+p+\ldots+p, \]
\[ 1 \quad 2 \quad r-1 \]
for \(k=0,1,\ldots,p\) and \(r = 1,2,\ldots,MF\). The derivatives are \(r\) with respect to the user’s variable \(x\).
12: LYF -- INTEGER  
   On entry:
   the dimension of the array YF as declared in the
   (sub)program from which E02AGF is called.
   Constraint: LYF>=n, where n=MF+p+...+p.
   \hspace{1em}1 \ 2 \ 3 \ 4 \ 5

13: IP(MF) -- INTEGER array  
   On entry: IP(r) must contain p, the order of the highest-
   order derivative specified at XF(r), for r=1,2,...,MF.
   p=0 implies that the value of the approximation at XF(r) is
   specified, but not that of any derivative. Constraint: IP(r)
   >= 0, for r=1,2,...,MF.

14: A(NROWS,KPLUS1) -- DOUBLE PRECISION array  
   On exit: A(i+1,j+1) contains the coefficient a_{ij}
   in the approximating polynomial of degree i, for i=n,n+1,...,k;
   j=0,1,...,i.

15: S(KPLUS1) -- DOUBLE PRECISION array  
   On exit: S(i+1) contains s_i, for i=n,n+1,...,k, the root-
   mean-square residual corresponding to the approximating
   polynomial of degree i. In the case where the number of data
   points with non-zero weight is equal to k+1-n, s_i is
   indeterminate: the routine sets it to zero. For the
   interpretation of the values of s_i and their use in
   selecting an appropriate degree, see Section 3.1 of the
   Chapter Introduction.

16: NP1 -- INTEGER  
   On exit: n+1, where n is the total number of constraint
   conditions imposed: n=MF+p+...+p.
   \hspace{1em}1 \ 2 \ 3 \ 4 \ 5

17: WRK(LWRK) -- DOUBLE PRECISION array  
   On exit: WRK contains weighted residuals of the highest
   degree of fit determined (k). The residual at x is in
   element Z(n+1)+3(m+k+1)+r, for r=1,2,...,m. The rest of the
   array is used as workspace.

18: LWRK -- INTEGER  
   On entry:
   the dimension of the array WRK as declared in the
(sub)program from which E02AGF is called.
Constraint: \(LWRK \geq \max(4M + 3KPLUS1, 8n + 5 \times \text{IPMAX} + MF + 10) + 2n + 2\),
where \(\text{IPMAX} = \max(\text{IP}(r))\).

19: IWRK(LIWRK) -- INTEGER array Workspace

20: LIWRK -- INTEGER Input
On entry:
the dimension of the array IWRK as declared in the
(sub)program from which E02AGF is called.
Constraint: \(LIWRK \geq 2 \times MF + 2\).

21: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry \(M < 1\),
or \(KPLUS1 < n + 1\),
or \(NROWS < KPLUS1\),
or \(MF < 1\),
or \(LYF < n\),
or \(LWRK is too small (see Section 5),\)
or \(LIWRK < 2 \times MF + 2\).
(Here \(n\) is the total number of constraint conditions.)

IFAIL= 2
\(IP(r) < 0\) for some \(r = 1,2,\ldots,MF\).

IFAIL= 3
\(XMIN \geq XMAX\), or \(XF(r)\) is not in the interval \(XMIN\) to \(XMAX\)
for some \(r = 1,2,\ldots,MF\), or the \(XF(r)\) are not distinct.

IFAIL= 4
\(X(r)\) is not in the interval \(XMIN\) to \(XMAX\) for some
\(r=1,2,\ldots,M\).
IFAIL = 5
  X(r) < X(r-1) for some r = 2, 3, ..., M.

IFAIL = 6
  KPLUS1 > m''+n, where m'' is the number of data points with
  non-zero weight and distinct abscissae which do not coincide
  with any XF(r). Thus there is no unique solution.

IFAIL = 7
  The polynomials (mu)(x) and/or (nu)(x) cannot be determined.
  The problem supplied is too ill-conditioned. This may occur
  when the constraint points are very close together, or large
  in number, or when an attempt is made to constrain high-
  order derivatives.

7. Accuracy

No complete error analysis exists for either the interpolating
algorithm or the approximating algorithm. However, considerable
experience with the approximating algorithm shows that it is
generally extremely satisfactory. Also the moderate number of
constraints, of low order, which are typical of data fitting
applications, are unlikely to cause difficulty with the
interpolating routine.

8. Further Comments

The time taken by the routine to form the interpolating
  3
polynomial is approximately proportional to n , and that to form
  3
the approximating polynomials is very approximately proportional
to m(k+1)(k+1-n).

To carry out a least-squares polynomial fit without constraints,
use E02ADF. To carry out polynomial interpolation only, use
E01AEF(*).

9. Example

The example program reads data in the following order, using the
notation of the parameter list above:

MF

IP(i), XF(i), Y-value and derivative values (if any) at
XF(i), for i = 1, 2, ..., MF

M
The output is:

- the root-mean-square residual for each degree from $n$ to $k$;
- the Chebyshev coefficients for the fit of degree $k$;
- the data points, and the fitted values and residuals for the fit of degree $k$.

The program is written in a generalized form which will read any number of data sets.

The data set supplied specifies 5 data points in the interval $[0.0, 4.0]$ with unit weights, to which are to be fitted polynomials, $p$, of degrees up to 4, subject to the 3 constraints:

$$p(0.0)=1.0, \quad p'(0.0)=-2.0, \quad p(4.0)=9.0.$$ 

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
This routine forms the polynomial which is the derivative of a given polynomial. Both the original polynomial and its derivative are represented in Chebyshev-series form. Given the coefficients $a_i$, for $i=0,1,\ldots,n$, of a polynomial $p(x)$ of degree $n$, where

$$ p(x) = -a_0 + a_1 T_1(x) + \ldots + a_n T_n(x) $$

the routine returns the coefficients $a_i$, for $i=0,1,\ldots,n-1$, of the polynomial $q(x)$ of degree $n-1$, where

$$ q(x) = \frac{dp(x)}{dx} = -a_0 + a_1 T_1(x) + \ldots + a_{n-1} T_{n-1}(x). $$

Here $T_j(x)$ denotes the Chebyshev polynomial of the first kind of degree $j$ with argument $x$. It is assumed that the normalised variable $x$ in the interval $[-1,+1]$ was obtained from the user's original variable $x$ in the interval $[x_{\min},x_{\max}]$ by the linear transformation

$$ 2x - (x_{\max} + x_{\min}) \quad x_{\max} - x_{\min} $$

and that the user requires the derivative to be with respect to the variable $x$. If the derivative with respect to $x$ is required, set $x_{\max} = 1$ and $x_{\min} = -1$.

Values of the derivative can subsequently be computed, from the coefficients obtained, by using E02AKF.
The method employed is that of [1] modified to obtain the derivative with respect to \( x \). Initially setting \( a = a_0 = 0 \), the routine forms successively

\[
a_{i+1} = a_i + \frac{2}{i-1} a_i \left( \frac{x}{x_{\text{max}}} \right)^i, \quad i=n,n-1,...,1.
\]

\( \min \leq x \leq \max \)

4. References


5. Parameters

1: NP1 -- INTEGER Input
   On entry: \( n+1 \), where \( n \) is the degree of the given polynomial \( p(x) \). Thus NP1 is the number of coefficients in this polynomial. Constraint: NP1 \( \geq 1 \).

2: XMIN -- DOUBLE PRECISION Input

3: XMAX -- DOUBLE PRECISION Input
   On entry: the lower and upper end-points respectively of the interval \( [x_{\text{min}},x_{\text{max}}] \). The Chebyshev-series representation is in terms of the normalised variable \( x \), where

\[
x = \frac{2x - (x_{\text{max}} + x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}}.
\]

Constraint: XMAX > XMIN.

4: A(LA) -- DOUBLE PRECISION array Input
   On entry: the Chebyshev coefficients of the polynomial \( p(x) \). Specifically, element \( 1 + i \times \text{IA1} \) of A must contain the coefficient \( a_i \), for \( i=0,1,...,n \). Only these \( n+1 \) elements will be accessed.
Unchanged on exit, but see ADIF, below.

5: IA1 -- INTEGER 
   Input
   On entry: the index increment of A. Most frequently the
   Chebyshev coefficients are stored in adjacent elements of A,
   and IA1 must be set to 1. However, if, for example, they are
   stored in A(1),A(4),A(7),..., then the value of IA1 must be
   3. See also Section 8. Constraint: IA1 >= 1.

6: LA -- INTEGER 
   Input
   On entry:
   the dimension of the array A as declared in the (sub)program
   from which E02AHF is called.
   Constraint: LA>=1+(NP1-1)*IA1.

7: PATM1 -- DOUBLE PRECISION 
   Output
   On exit: the value of p(x ). If this value is passed to
   min
   the integration routine E02AJF with the coefficients of q(x)
   , then the original polynomial p(x) is recovered, including
   its constant coefficient.

8: ADIF(IADIF) -- DOUBLE PRECISION array 
   Output
   On exit: the Chebyshev coefficients of the derived
   polynomial q(x). (The differentiation is with respect to the
   variable x). Specifically, element 1+i*IADIF1 of ADIF
   contains the coefficient a , i=0,1,...,n-1. Additionally
   i
   element 1+n*IADIF1 is set to zero. A call of the routine may
   have the array name ADIF the same as A, provided that note
   is taken of the order in which elements are overwritten,
   when choosing the starting elements and increments IA1 and
   IADIF1: i.e., the coefficients a ,a ,...,a must be intact
   0 1 i-1
   after coefficient a is stored. In particular, it is
   i
   possible to overwrite the a completely by having IA1 =
   i
   IADIF1, and the actual arrays for A and ADIF identical.

9: IADIF1 -- INTEGER 
   Input
   On entry: the index increment of ADIF. Most frequently the
   Chebyshev coefficients are required in adjacent elements of
   ADIF, and IADIF1 must be set to 1. However, if, for example,
   they are to be stored in ADIF(1),ADIF(4),ADIF(7),..., then
   the value of IADIF1 must be 3. See Section 8. Constraint:
IADIF1 >= 1.

10:  LADIF -- INTEGER  Input

    On entry:
    the dimension of the array ADIF as declared in the
    (sub)program from which E02AHF is called.
    Constraint: LADIF>=1+(NP1-1)*IADIF1.

11:  IFAIL -- INTEGER  Input/Output

    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

    Errors detected by the routine:

    IFAIL= 1
    On entry NP1 < 1,
    or XMAX <= XMIN,
    or IA1 < 1,
    or LA<=(NP1-1)*IA1,
    or IADIF1 < 1,
    or LADIF<=(NP1-1)*IADIF1.

7. Accuracy

    There is always a loss of precision in numerical differentiation,
    in this case associated with the multiplication by 2i in the
    formula quoted in Section 3.

8. Further Comments

    The time taken by the routine is approximately proportional to
    n+1.

    The increments IA1, IADIF1 are included as parameters to give a
    degree of flexibility which, for example, allows a polynomial in
    two variables to be differentiated with respect to either
    variable without rearranging the coefficients.

9. Example
Suppose a polynomial has been computed in Chebyshev-series form to fit data over the interval \([-0.5, 2.5]\). The example program evaluates the 1st and 2nd derivatives of this polynomial at 4 equally spaced points over the interval. (For the purposes of this example, \(XMIN\), \(XMAX\) and the Chebyshev coefficients are simply supplied in DATA statements. Normally a program would first read in or generate data and compute the fitted polynomial.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E02 -- Curve and Surface Fitting
E02AJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details.

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02AJF determines the coefficients in the Chebyshev-series representation of the indefinite integral of a polynomial given in Chebyshev-series form.

2. Specification

```fortran
SUBROUTINE E02AJF (NP1, XMIN, XMAX, A, IA1, LA, QATM1,
  1 AINT, IAINT1, LAINT, IFAIL)
```

```fortran
INTEGER NP1, IA1, LA, IAINT1, LAINT, IFAIL
DOUBLE PRECISION XMIN, XMAX, A(LA), QATM1, AINT(LAINT)
```

3. Description

This routine forms the polynomial which is the indefinite integral of a given polynomial. Both the original polynomial and its integral are represented in Chebyshev-series form. If supplied with the coefficients \(a_i\), for \(i=0, 1, \ldots, n\), of a polynomial \(p(x)\) of degree \(n\), where

\[
p(x) = -a_0 + a_1 T(x) + \cdots + a_n T^n(x),
\]

\(\quad 2 \quad 0 \quad 1 \quad 1 \quad n \quad n\)
the routine returns the coefficients $a'$, for $i=0,1,...,n+1$, of
the polynomial $q(x)$ of degree $n+1$, where

$$
q(x) = -a' +a' T(x) + ... + a' T^{n+1}(x),
$$

and

$$
q(x)= |p(x)dx.
$$

Here $T_j(x)$ denotes the Chebyshev polynomial of the first kind of
degree $j$ with argument $x$. It is assumed that the normalised
variable $x$ in the interval $[-1,+1]$ was obtained from the user’s
original variable $x$ in the interval $[x_{\text{min}},x_{\text{max}}]$ by the linear
transformation

$$
x = \frac{2x - (x_{\text{max}} + x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}}.
$$

and that the user requires the integral to be with respect to the
variable $x$. If the integral with respect to $x$ is required, set
$x = 1$ and $x = -1$.

Values of the integral can subsequently be computed, from the
coefficients obtained, by using E02AKF.

The method employed is that of Chebyshev-series [1] modified for
integrating with respect to $x$. Initially taking $a = 0$, the
routine forms successively

$$
a_i = a_{i-1} + a_{i+1}, \quad i=1,2,...,n+1.
$$
i 2i 2

The constant coefficient $a'$ is chosen so that $q(x)$ is equal to a specified value, $Q_{ATM1}$, at the lower end-point of the interval on which it is defined, i.e., $x=-1$, which corresponds to $x=x_{\text{min}}$.

4. References


5. Parameters

1: NP1 -- INTEGER Input
On entry: $n+1$, where $n$ is the degree of the given polynomial $p(x)$. Thus NP1 is the number of coefficients in this polynomial. Constraint: NP1 $\geq 1$.

2: XMIN -- DOUBLE PRECISION Input

3: XMAX -- DOUBLE PRECISION Input
On entry: the lower and upper end-points respectively of the interval $[x_{\text{min}}, x_{\text{max}}]$. The Chebyshev-series representation is in terms of the normalised variable $x$, where

$$2x-(x_{\text{max}}+x_{\text{min}})$$

$$x=\frac{x_{\text{max}}-x_{\text{min}}}{x_{\text{max}}-x_{\text{min}}}.$$  

Constraint: XMAX $> XMIN$.

4: A(LA) -- DOUBLE PRECISION array Input
On entry: the Chebyshev coefficients of the polynomial $p(x)$ . Specifically, element $1+i*IA1$ of $A$ must contain the coefficient $a_i$ for $i=0,1,\ldots,n$. Only these $n+1$ elements will be accessed.

Unchanged on exit, but see AINT, below.

5: IA1 -- INTEGER Input
On entry: the index increment of \( A \). Most frequently the Chebyshev coefficients are stored in adjacent elements of \( A \), and \( IA1 \) must be set to 1. However, if for example, they are stored in \( A(1), A(4), A(7), \ldots \), then the value of \( IA1 \) must be 3. See also Section 8. Constraint: \( IA1 \geq 1 \).

6: \( LA \) -- INTEGER Input
On entry: the dimension of the array \( A \) as declared in the (sub)program from which E02AJF is called.
Constraint: \( LA \geq 1 + (NP1 - 1) \times IA1 \).

7: \( QATM1 \) -- DOUBLE PRECISION Input
On entry: the value that the integrated polynomial is required to have at the lower end-point of its interval of definition, i.e., at \( x = -1 \) which corresponds to \( x = \min \). Thus, \( QATM1 \) is a constant of integration and will normally be set to zero by the user.

8: \( AINT(LAINT) \) -- DOUBLE PRECISION array Output
On exit: the Chebyshev coefficients of the integral \( q(x) \). (The integration is with respect to the variable \( x \), and the constant coefficient is chosen so that \( q(x) \) equals \( QATM1 \))
Specifically, element \( 1 + i \times IAINT1 \) of \( AINT \) contains the coefficient \( a'_i \), for \( i = 0, 1, \ldots, n + 1 \). A call of the routine may have the array name \( AINT \) the same as \( A \), provided that note is taken of the order in which elements are overwritten when choosing starting elements and increments \( IA1 \) and \( IAINT1 \); i.e., the coefficients, \( a_0, a_1, \ldots, a_{i-2} \) must be intact after coefficient \( a'_i \) is stored. In particular it is possible to overwrite the \( a_0 \) entirely by having \( IA1 = IAINT1 = 1 \), and the actual array for \( A \) and \( AINT \) identical.

9: \( IAINT1 \) -- INTEGER Input
On entry: the index increment of \( AINT \). Most frequently the Chebyshev coefficients are required in adjacent elements of \( AINT \), and \( IAINT1 \) must be set to 1. However, if, for example, they are to be stored in \( AINT(1), AINT(4), AINT(7), \ldots \), then the value of \( IAINT1 \) must be 3. See also Section 8. Constraint: \( IAINT1 \geq 1 \).

10: \( LAINT \) -- INTEGER Input
On entry:
the dimension of the array AINT as declared in the
(sub)program from which E02AJF is called.
Constraint: LAINT>=1+NP1*IAINT1.

11: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:
IFAIL= 1
On entry NP1 < 1,
or XMAX <= XMIN,
or IA1 < 1,
or LA<=(NP1-1)*IA1,
or IAIN1 < 1,
or LAINT<=NP1*IAINT1.

7. Accuracy
In general there is a gain in precision in numerical integration,
in this case associated with the division by 2i in the formula
quoted in Section 3.

8. Further Comments
The time taken by the routine is approximately proportional to
n+1.
The increments IA1, IAIN1 are included as parameters to give a
degree of flexibility which, for example, allows a polynomial in
two variables to be integrated with respect to either variable
without rearranging the coefficients.

9. Example
Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval [-0.5,2.5]. The example program
evaluates the integral of the polynomial from 0.0 to 2.0. (For
the purpose of this example, XMIN, XMAX and the Chebyshev
coefficients are simply supplied in DATA statements. Normally a
program would read in or generate data and compute the fitted
polynomial).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E02 -- Curve and Surface Fitting
E02AKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose
E02AKF evaluates a polynomial from its Chebyshev-series
representation, allowing an arbitrary index increment for
accessing the array of coefficients.

2. Specification

SUBROUTINE E02AKF (NP1, XMIN, XMAX, A, IA1, LA, X, RESULT,
 IFAIL)
 INTEGER NP1, IA1, LA, IFAIL
 DOUBLE PRECISION XMIN, XMAX, A(LA), X, RESULT

3. Description
If supplied with the coefficients $a_i$, for $i=0,1,\ldots,n$, of a
polynomial $p(x)$ of degree $n$, where

$$
p(x) = -a_0 + a_1 T_1(x) + \cdots + a_n T_n(x),
$$

this routine returns the value of $p(x)$ at a user-specified value
of the variable $x$. Here $T_j(x)$ denotes the Chebyshev polynomial of
the first kind of degree $j$ with argument $x$. It is assumed that

the independent variable $x$ in the interval $[-1,+1]$ was obtained
from the user’s original variable $x$ in the interval $[x_{\text{min}}, x_{\text{max}}]$ by the linear transformation

$$x = \frac{2x_{\text{min}} - (x_{\text{max}} + x_{\text{min}})}{2}.$$ 

The coefficients $a_i$ may be supplied in the array $A$, with any
increment between the indices of array elements which contain
successive coefficients. This enables the routine to be used in
surface fitting and other applications, in which the array might
have two or more dimensions.

The method employed is based upon the three-term recurrence
relation due to Clenshaw [1], with modifications due to Reinsch
and Gentleman (see [4]). For further details of the algorithm and
its use see Cox [2] and Cox and Hayes [3].

4. References

[1] Clenshaw C W (1955) A Note on the Summation of Chebyshev-
series. Math. Tables Aids Comput. 9 118--120.


suite of algorithms for the non-specialist user. Report

J. 12 160--165.

5. Parameters

1: NP1 -- INTEGER
   Input
   On entry: $n+1$, where $n$ is the degree of the given
polynomial \( p(x) \). Constraint: \( N_P1 \geq 1 \).

2: \( XMIN \) -- DOUBLE PRECISION Input

3: \( XMAX \) -- DOUBLE PRECISION Input

On entry: the lower and upper end-points respectively of the interval \([x_{\min}, x_{\max}]\). The Chebyshev-series representation is in terms of the normalised variable \( x \), where

\[
\frac{2x - (x_{\max} + x_{\min})}{x_{\max} - x_{\min}} = \frac{x_{\min} - x}{x_{\max} - x_{\min}}.
\]

Constraint: \( XMIN < XMAX \).

4: \( A(LA) \) -- DOUBLE PRECISION array Input

On entry: the Chebyshev coefficients of the polynomial \( p(x) \). Specifically, element \( i + i \cdot IA1 \) must contain the coefficient \( a_i \), for \( i=0,1,...,n \). Only these \( n+1 \) elements will be accessed.

5: \( IA1 \) -- INTEGER Input

On entry: the index increment of \( A \). Most frequently, the Chebyshev coefficients are stored in adjacent elements of \( A \), and \( IA1 \) must be set to 1. However, if, for example, they are stored in \( A(1), A(4), A(7),... \), then the value of \( IA1 \) must be 3. Constraint: \( IA1 \geq 1 \).

6: \( LA \) -- INTEGER Input

On entry: the dimension of the array \( A \) as declared in the (sub)program from which \( E02AKF \) is called. Constraint: \( LA \geq (N_P1 - 1) \cdot IA1 + 1 \).

7: \( X \) -- DOUBLE PRECISION Input

On entry: the argument \( x \) at which the polynomial is to be evaluated. Constraint: \( XMIN \leq X \leq XMAX \).

8: \( RESULT \) -- DOUBLE PRECISION Output

On exit: the value of the polynomial \( p(x) \).
9:  IFAIL -- INTEGER  
     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry NP1 < 1,
          or IA1 < 1,
          or LA<=(NP1-1)*IA1,
          or XMIN >= XMAX.

IFAIL= 2
     X does not satisfy the restriction XMIN <= X <= XMAX.

7. Accuracy

The rounding errors are such that the computed value of the
polynomial is exact for a slightly perturbed set of coefficients
\( a_i + (\delta)a_i \). The ratio of the sum of the absolute values of the
\( i \)
(\delta)a_i to the sum of the absolute values of the \( a_i \) is less
\( i \)
than a small multiple of \((n+1)\)*machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to
\( n+1 \).

9. Example

Suppose a polynomial has been computed in Chebyshev-series form
to fit data over the interval \([-0.5,2.5]\). The example program
evaluates the polynomial at 4 equally spaced points over the
interval. (For the purposes of this example, XMIN, XMAX and the
Chebyshev coefficients are supplied in DATA statements. Normally
a program would first read in or generate data and compute the
fitted polynomial.)

The example program is not reproduced here. The source code for
CHAPTER 15. CHAPTER N

all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting

E02BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BAF computes a weighted least-squares approximation to an arbitrary set of data points by a cubic spline with knots prescribed by the user. Cubic spline interpolation can also be carried out.

2. Specification

SUBROUTINE E02BAF (M, NCAP7, X, Y, W, LAMDA, WORK1, WORK2, C, SS, IFAIL)

INTEGER M, NCAP7, IFAIL
DOUBLE PRECISION X(M), Y(M), W(M), LAMDA(NCAP7), WORK1(M), WORK2(4*NCAP7), C(NCAP7), SS

3. Description

This routine determines a least-squares cubic spline approximation \( s(x) \) to the set of data points \( (x^r, y^r) \) with weights \( w^r \), for \( r=1,2,\ldots,m \). The value of \( \text{NCAP7} = n+7 \), where \( n \) is the number of intervals of the spline (one greater than the number of interior knots), and the values of the knots \( (\lambda_5, \lambda_6, \ldots, \lambda_{n+3}) \), interior to the data interval, are prescribed by the user.

\( s(x) \) has the property that it minimizes \( (\theta) \), the sum of squares of the weighted residuals \( (\epsilon^r) \), for \( r=1,2,\ldots,m \), where

\[
(\epsilon^r) = w^r (y^r - s(x^r)).
\]
The routine produces this minimizing value of \( \theta \) and the coefficients \( c_1, c_2, \ldots, c_q \), where \( q = n+3 \), in the B-spline representation

\[
s(x) = \sum_{i=1}^{q} c_i N_i(x).
\]

Here \( N_i(x) \) denotes the normalised B-spline of degree 3 defined upon the knots \( \lambda_i, \lambda_{i+1}, \ldots, \lambda_{i+4} \).

In order to define the full set of B-splines required, eight additional knots \( \lambda_{n+4}, \lambda_{n+5}, \lambda_{n+6}, \lambda_{n+7} \) are inserted automatically by the routine. The first four of these are set equal to the smallest \( x \) and the last four to the largest \( x \).

The representation of \( s(x) \) in terms of B-splines is the most compact form possible in that only \( n+3 \) coefficients, in addition to the \( n+7 \) knots, fully define \( s(x) \).

The method employed involves forming and then computing the least-squares solution of a set of \( m \) linear equations in the coefficients \( c_i \) (\( i=1,2,\ldots,n+3 \)). The equations are formed using a recurrence relation for B-splines that is unconditionally stable (Cox [1], de Boor [5]), even for multiple (coincident) knots. The least-squares solution is also obtained in a stable manner by using orthogonal transformations, viz. a variant of Givens rotations (Gentleman [6] and [7]). This requires only one equation to be stored at a time. Full advantage is taken of the structure of the equations, there being at most four non-zero values of \( N_i(x) \) for any value of \( x \) and hence at most four.
coefficients in each equation.

For further details of the algorithm and its use see Cox [2], [3] and [4].

Subsequent evaluation of \( s(x) \) from its B-spline representation may be carried out using E02BBF. If derivatives of \( s(x) \) are also required, E02BCF may be used. E02BDF can be used to compute the definite integral of \( s(x) \).

4. References


5. Parameters

1: M -- INTEGER Input

On entry: the number \( m \) of data points. Constraint: \( M \geq MDIST \geq 4 \), where \( MDIST \) is the number of distinct \( x \) values in the data.

2: NCAP7 -- INTEGER Input
On entry: \( n+7 \), where \( n \) is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range \( x_1 \) to \( x_m \) over which the spline is defined. Constraint: \( 8 \leq NCAP7 \leq MDIST + 4 \), where \( MDIST \) is the number of distinct \( x \) values in the data.

3: \( X(M) \) -- DOUBLE PRECISION array
   On entry: the values \( x \) of the independent variable \( r \) (abscissa), for \( r=1,2,\ldots,m \). Constraint: \( x_1 \leq x \leq \cdots \leq x_m \).

4: \( Y(M) \) -- DOUBLE PRECISION array
   On entry: the values \( y \) of the dependent variable \( r \) (ordinate), for \( r=1,2,\ldots,m \).

5: \( W(M) \) -- DOUBLE PRECISION array
   On entry: the values \( w \) of the weights, for \( r=1,2,\ldots,m \).
   For advice on the choice of weights, see the Chapter Introduction. Constraint: \( W(r) > 0 \), for \( r=1,2,\ldots,m \).

6: \( LAMDA(NCAP7) \) -- DOUBLE PRECISION array
   On entry: \( LAMDA(i) \) must be set to the \( (i-4) \)th (interior) knot, \( \lambda_i \), for \( i=5,6,\ldots,n+3 \). Constraint: \( X(1) < LAMDA(5) \leq LAMDA(6) \leq \cdots \leq LAMDA(NCAP7-4) < X(M) \). On exit: the input values are unchanged, and \( LAMDA(i) \), for \( i = 1, 2, 3, 4, NCAP7-3, NCAP7-2, NCAP7-1, NCAP7 \) contains the additional (exterior) knots introduced by the routine. For advice on the choice of knots, see Section 3.3 of the Chapter Introduction.

7: \( WORK1(M) \) -- DOUBLE PRECISION array
   Workspace

8: \( WORK2(4*NCAP7) \) -- DOUBLE PRECISION array
   Workspace

9: \( C(NCAP7) \) -- DOUBLE PRECISION array
   On exit: the coefficient \( c_i \) of the B-spline \( N_i(x) \), for \( i=1,2,\ldots,n+3 \). The remaining elements of the array are not used.
10: SS -- DOUBLE PRECISION          Output
    On exit: the residual sum of squares, (theta).

11: IFAIL -- INTEGER               Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
          familiar with this parameter (described in the Essential
          Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
          Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
    The knots fail to satisfy the condition
    
    \[ X(1) < LAMDA(5) <= LAMDA(6) <= \ldots <= LAMDA(NCAP7-4) < X(M). \]
    
    Thus the knots are not in correct order or are not interior
    to the data interval.

IFAIL= 2
    The weights are not all strictly positive.

IFAIL= 3
    The values of \( X(r) \), for \( r = 1,2,\ldots,M \) are not in non-
    decreasing order.

IFAIL= 4
    \( NCAP7 < 8 \) (so the number of interior knots is negative) or
    \( NCAP7 > MDIST + 4 \), where MDIST is the number of distinct x
    values in the data (so there cannot be a unique solution).

IFAIL= 5
    The conditions specified by Schoenberg and Whitney [8] fail to
    hold for at least one subset of the distinct data
    abscissae. That is, there is no subset of NCAP7-4 strictly
    increasing values, \( X(R(1)),X(R(2)),\ldots,X(R(NCAP7-4)) \), among
    the abscissae such that
    
    \[ X(R(1)) < LAMDA(1) < X(R(5)), \]
    \[ X(R(2)) < LAMDA(2) < X(R(6)), \]
    \[ \ldots \]
    \[ X(R(NCAP7-8)) < LAMDA(NCAP7-8) < X(R(NCAP7-4)). \]

    This means that there is no unique solution: there are
    regions containing too many knots compared with the number
    of data points.
7. Accuracy

The rounding errors committed are such that the computed coefficients are exact for a slightly perturbed set of ordinates \( y + (\delta)y \). The ratio of the root-mean-square value for the \( r \) \( (\delta)y \) to the root-mean-square value of the \( r \) \( y \) can be expected to be less than a small multiple of \((\kappa) \cdot m \cdot \text{machine precision}\), where \((\kappa)\) is a condition number for the problem. Values of \((\kappa)\) for 20-30 practical data sets all proved to lie between 4.5 and 7.8 (see Cox [3]). (Note that for these data sets, replacing the coincident end knots at the end-points \( x \) and \( x_1 \) used in the routine by various choices of non-coincident exterior knots gave values of \((\kappa)\) between 16 and 180. Again see Cox [3] for further details.) In general we would not expect \((\kappa)\) to be large unless the choice of knots results in near-violation of the Schoenberg-Whitney conditions.

A cubic spline which adequately fits the data and is free from spurious oscillations is more likely to be obtained if the knots are chosen to be grouped more closely in regions where the function (underlying the data) or its derivatives change more rapidly than elsewhere.

8. Further Comments

The time taken by the routine is approximately \( C \cdot (2m + n + 7) \) seconds, where \( C \) is a machine-dependent constant.

Multiple knots are permitted as long as their multiplicity does not exceed 4, i.e., the complete set of knots must satisfy

\[
\lambda_i < \lambda_{i+4}, \quad i = 1, 2, \ldots, n+3, \quad (\text{cf. Section 6}).
\]

At a knot of multiplicity one (the usual case), \( s(x) \) and its first two derivatives are continuous. At a knot of multiplicity two, \( s(x) \) and its first derivative are continuous. At a knot of multiplicity three, \( s(x) \) is continuous, and at a knot of multiplicity four, \( s(x) \) is generally discontinuous.

The routine can be used efficiently for cubic spline interpolation, i.e., if \( m = n + 3 \). The abscissae must then of course
satisfy $x < x_1 < \ldots < x_m$. Recommended values for the knots in this case are $(\lambda_i) = x_i$, for $i=5,6,\ldots,n+3$.

9. Example

Determine a weighted least-squares cubic spline approximation with five intervals (four interior knots) to a set of 14 given data points. Tabulate the data and the corresponding values of the approximating spline, together with the residual errors, and also the values of the approximating spline at points half-way between each pair of adjacent data points.

The example program is written in a general form that will enable a cubic spline approximation with $n$ intervals ($n-1$ interior knots) to be obtained to $m$ data points, with arbitrary positive weights, and the approximation to be tabulated. Note that E02BBF is used to evaluate the approximating spline. The program is self-starting in that any number of data sets can be supplied.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E02 -- Curve and Surface Fitting
E02BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BBF evaluates a cubic spline from its B-spline representation.

2. Specification

```plaintext
SUBROUTINE E02BBF (NCAP7, LAMDA, C, X, S, IFAIL)
INTEGER NCAP7, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), X, S
```

3. Description
This routine evaluates the cubic spline \( s(x) \) at a prescribed argument \( x \) from its augmented knot set \( \{\lambda_i\} \), for \( i=1,2,...,n+7 \), (see E02BAF) and from the coefficients \( c_i \), for \( i=1,2,...,q \) in its B-spline representation

\[
s(x) = \sum_{i=1}^{q} c_i N_i(x)
\]

Here \( q=n+3 \), where \( n \) is the number of intervals of the spline, and \( N_i(x) \) denotes the normalised B-spline of degree 3 defined upon the knots \( \{\lambda_i\}, \{\lambda_{i+1}\},...,\{\lambda_{i+4}\} \). The prescribed argument \( x \) must satisfy \( \lambda_i \leq x \leq \lambda_{i+4} \). The method employed is that of evaluation by taking convex combinations due to de Boor [4]. For further details of the algorithm and its use see Cox [1] and [3]. It is expected that a common use of E02BBF will be the evaluation of the cubic spline approximations produced by E02BAF. A generalization of E02BBF which also forms the derivative of \( s(x) \) is E02BCF. E02BCF takes about 50% longer than E02BBF.

4. References


5. Parameters

1: NCAP7 -- INTEGER
   Input
   On entry: n+7, where n is the number of intervals (one greater than the number of interior knots, i.e., the knots strictly within the range (lambda) to (lambda) ) over 4 n+4 which the spline is defined. Constraint: NCAP7 >= 8.

2: LAMDA(NCAP7) -- DOUBLE PRECISION array
   Input
   On entry: LAMDA(j) must be set to the value of the jth member of the complete set of knots, (lambda) for j = 1,2,...,n+7. Constraint: the LAMDA(j) must be in non-decreasing order with LAMDA(NCAP7-3) > LAMDA(4).

3: C(NCAP7) -- DOUBLE PRECISION array
   Input
   On entry: the coefficient c of the B-spline N_i(x), for i = 1,2,...,n+3. The remaining elements of the array are not used.

4: X -- DOUBLE PRECISION
   Input
   On entry: the argument x at which the cubic spline is to be evaluated. Constraint: LAMDA(4) <= X <= LAMDA(NCAP7-3).

5: S -- DOUBLE PRECISION
   Output
   On exit: the value of the spline, s(x).

6: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The argument X does not satisfy LAMDA(4) <= X <= LAMDA(NCAP7-3).

In this case the value of S is set arbitrarily to zero.

IFAIL= 2
NCAP7 < 8, i.e., the number of interior knots is negative.

7. Accuracy

The computed value of s(x) has negligible error in most practical situations. Specifically, this value has an absolute error bounded in modulus by 18*c * machine precision, where c is
\[
\max_{j} \max_{j+1, j+2, j+3} \text{the largest in modulus of } c, c, c, \text{ and } c,
\]
and j is an integer such that \((\lambda_{j}) \leq x \leq (\lambda_{j+3})\). If \(c, c, c, c, c\) and \(c, c, c, c\) are all of the same sign, then the computed value of \(S(x)\) has a relative error not exceeding 20*machine precision in modulus. For further details see Cox [2].

8. Further Comments

The time taken by the routine is approximately \(C \times (1+0.1 \times \log(n+7))\) seconds, where C is a machine-dependent constant.

Note: the routine does not test all the conditions on the knots given in the description of LAMDA in Section 5, since to do this would result in a computation time approximately linear in \(n+7\) instead of \(\log(n+7)\). All the conditions are tested in E02BAF, however.

9. Example

Evaluate at 9 equally-spaced points in the interval 1.0 <= x <= 9.0 the cubic spline with (augmented) knots 1.0, 1.0, 1.0, 1.0, 3.0, 6.0, 8.0, 9.0, 9.0, 9.0, 9.0 and normalised cubic B-spline coefficients 1.0, 2.0, 4.0, 7.0, 6.0, 4.0, 3.0.
The example program is written in a general form that will enable
a cubic spline with \( n \) intervals, in its normalised cubic B-spline
form, to be evaluated at \( m \) equally-spaced points in the interval
\( \text{LAMDA}(4) \leq x \leq \text{LAMDA}(n+4) \). The program is self-starting in that
any number of data sets may be supplied.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting

E02BCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

E02BCF evaluates a cubic spline and its first three derivatives
from its B-spline representation.

2. Specification

```
SUBROUTINE E02BCF (NCAP7, LAMDA, C, X, LEFT, S, IFAIL)
INTEGER NCAP7, LEFT, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), X, S(4)
```

3. Description

This routine evaluates the cubic spline \( s(x) \) and its first three
derivatives at a prescribed argument \( x \). It is assumed that \( s(x) \)
is represented in terms of its B-spline coefficients \( c_i \), for \( i = 1, 2, \ldots, n+3 \) and (augmented) ordered knot set \( \text{lambda}_i \), for \( i = 1, 2, \ldots, n+7 \), (see E02BAF), i.e.,

\[
q
\]
\[
\begin{align*}
\text{s}(x) &= \sum_{i=1}^{q} c_i N_i(x) \\
\text{Here } q &= n + 3, \text{ } n \text{ is the number of intervals of the spline and } N_i(x) \\
\text{denotes the normalised B-spline of degree 3 (order 4) defined} \\
\text{upon the knots } (\lambda)_i, (\lambda)_{i+1}, \ldots, (\lambda)_{i+4}. \text{ The} \\
\text{prescribed argument } x \text{ must satisfy} \\
(\lambda)_i \leq x \leq (\lambda)_{i+4} \\
\text{At a simple knot } (\lambda)_i \text{ (i.e., one satisfying} \\
(\lambda)_i < (\lambda)_{i+1}, \text{ the third derivative of the} \\
\text{spline is in general discontinuous. At a multiple knot (i.e., two} \\
or more knots with the same value), lower derivatives, and even 
\text{the spline itself, may be discontinuous. Specifically, at a point} 
\text{x} = u \text{ where (exactly) } r \text{ knots coincide (such a point is termed a} 
\text{knot of multiplicity } r), \text{ the values of the derivatives of order} 
4-j, \text{ for } j = 1, 2, \ldots, r, \text{ are in general discontinuous. (Here} 
1 \leq r \leq 4; r > 4 \text{ is not meaningful.) The user must specify whether the} 
\text{value at such a point is required to be the left- or right-hand} 
\text{derivative.} \\
\text{The method employed is based upon:} \\
\text{(i) carrying out a binary search for the knot interval} 
\text{containing the argument } x \text{ (see Cox [3]),} \\
\text{(ii) evaluating the non-zero B-splines of orders 1, 2, 3 and} 
4 \text{ by recurrence (see Cox [2] and [3]),} \\
\text{(iii) computing all derivatives of the B-splines of order 4} 
\text{by applying a second recurrence to these computed B-spline} 
\text{values (see de Boor [1]),} \\
\text{(iv) multiplying the 4th-order B-spline values and their} 
\text{derivative by the appropriate B-spline coefficients, and} 
\text{summing, to yield the values of } s(x) \text{ and its derivatives.} \\
\text{E02BCF can be used to compute the values and derivatives of cubic} 
spline fits and interpolants produced by E02BAF.}
\end{align*}
\]
If only values and not derivatives are required, E02BBF may be used instead of E02BCF, which takes about 50% longer than E02BBF.

4. References


5. Parameters

1: NCAP7 -- INTEGER Input

On entry: \( n+7 \), where \( n \) is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range \((\lambda)\) to \((\lambda)\) over which the spline is defined).

Constraint: NCAP7 >= 8.

2: LAMDA(NCAP7) -- DOUBLE PRECISION array Input

On entry: LAMDA(j) must be set to the value of the jth member of the complete set of knots, \((\lambda)\), for \( j = 1, 2, \ldots, n+7 \). Constraint: the LAMDA(j) must be in non-decreasing order with LAMDA(NCAP7-3) > LAMDA(4).

3: C(NCAP7) -- DOUBLE PRECISION array Input

On entry: the coefficient \( c_i \) of the B-spline \( N_i(x) \), for \( i = 1, 2, \ldots, n+3 \). The remaining elements of the array are not used.

4: X -- DOUBLE PRECISION Input

On entry: the argument \( x \) at which the cubic spline and its derivatives are to be evaluated. Constraint: LAMDA(4) <= X <= LAMDA(NCAP7-3).
5: LEFT -- INTEGER
   Input
On entry: specifies whether left- or right-hand values of
the spline and its derivatives are to be computed (see
Section 3). Left- or right-hand values are formed according
to whether LEFT is equal or not equal to 1. If x does not
coincide with a knot, the value of LEFT is immaterial. If x
= LAMDA(4), right-hand values are computed, and if x = LAMDA
(NCAP7-3), left-hand values are formed, regardless of the
value of LEFT.

6: S(4) -- DOUBLE PRECISION array
   Output
On exit: S(j) contains the value of the (j-1)th derivative
of the spline at the argument x, for j = 1,2,3,4. Note that
S(1) contains the value of the spline.

7: IFAIL -- INTEGER
   Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   NCAP7 < 8, i.e., the number of intervals is not positive.

IFAIL = 2
   Either LAMDA(4) >= LAMDA(NCAP7-3), i.e., the range over
which s(x) is defined is null or negative in length, or X is
an invalid argument, i.e., X < LAMDA(4) or X >
LAMDA(NCAP7-3).

7. Accuracy

The computed value of s(x) has negligible error in most practical
situations. Specifically, this value has an absolute error
bounded in modulus by 18*c * machine precision, where c is
max
the largest in modulus of c ,c ,c and c , and j is an
j j+1 j+2 j+3
integer such that (lambda) <=x<=(lambda) . If c ,c
j+3 j+4
and c are all of the same sign, then the computed value of
j+3
s(x) has relative error bounded by 18*machine precision. For full
details see Cox [3].
No complete error analysis is available for the computation of the derivatives of \( s(x) \). However, for most practical purposes the absolute errors in the computed derivatives should be small.

8. Further Comments

The time taken by this routine is approximately linear in

\[ \log(n+7). \]

Note: the routine does not test all the conditions on the knots given in the description of LAMDA in Section 5, since to do this would result in a computation time approximately linear in \( n+7 \) instead of \( \log(n+7) \). All the conditions are tested in E02BAF, however.

9. Example

Compute, at the 7 arguments \( x = 0, 1, 2, 3, 4, 5, 6 \), the left- and right-hand values and first 3 derivatives of the cubic spline defined over the interval \( 0 \leq x \leq 6 \) having the 6 interior knots \( x = 1, 3, 3, 3, 4, 4 \), the 8 additional knots \( 0, 0, 0, 0, 6, 6, 6, 6 \), and the 10 B-spline coefficients \( 10, 12, 13, 15, 22, 26, 24, 18, 14, 12 \).

The input data items (using the notation of Section 5) comprise the following values in the order indicated:

\[ n \quad m \]
\[ \text{LAMDA}(j), \quad \text{for } j=1,2,\ldots,\text{NCAP7} \]
\[ C(j), \quad \text{for } j=1,2,\ldots,\text{NCAP7}-4 \]
\[ x(i), \quad \text{for } i=1,2,\ldots,m \]

The example program is written in a general form that will enable the values and derivatives of a cubic spline having an arbitrary number of knots to be evaluated at a set of arbitrary points. Any number of data sets may be supplied. The only changes required to the program relate to the dimensions of the arrays LAMDA and C.
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting E02BDF
E02BDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BDF computes the definite integral of a cubic spline from its B-spline representation.

2. Specification

SUBROUTINE E02BDF (NCAP7, LAMDA, C, DEFINT, IFAIL)
INTEGER NCAP7, IFAIL
DOUBLE PRECISION LAMDA(NCAP7), C(NCAP7), DEFINT

3. Description

This routine computes the definite integral of the cubic spline s(x) between the limits x=a and x=b, where a and b are respectively the lower and upper limits of the range over which s(x) is defined. It is assumed that s(x) is represented in terms of its B-spline coefficients c , for i=1,2,...,n+3 and
\[
\int_a^b s(x)\,dx = \sum_{i=1}^{n+3} c_i \int_{\lambda_{i-3}}^{\lambda_i} N_i(x) \,dx
\]

(augmented) ordered knot set (lambda) , for i=1,2,...,n+7, with
\[
\lambda_i = a, \text{ for } i = 1, 2, 3, 4 \text{ and } \lambda_i = b, \text{ for } i = n+4, n+5, n+6, n+7, \text{ (see E02BAF), i.e.,}
\]

\[
s(x) = \sum_{i=1}^{n+3} c_i N_i(x).
\]

\[
q \quad --
\]

\[
s(x) = \sum_{i=1}^{n+3} c_i N_i(x).
\]
Here $q=n+3$, $n$ is the number of intervals of the spline and $N_i(x)$ denotes the normalised B-spline of degree 3 (order 4) defined upon the knots $(\lambda_i), (\lambda_{i+1}), \ldots, (\lambda_{i+4})$.

The method employed uses the formula given in Section 3 of Cox [1].

E02BDF can be used to determine the definite integrals of cubic spline fits and interpolants produced by E02BAF.

4. References


5. Parameters

1: NCAP7 -- INTEGER Input

On entry: $n+7$, where $n$ is the number of intervals of the spline (which is one greater than the number of interior knots, i.e., the knots strictly within the range $a$ to $b$) over which the spline is defined. Constraint: NCAP7 $\geq 8$.

2: LAMDA(NCAP7) -- DOUBLE PRECISION array Input

On entry: LAMDA($j$) must be set to the value of the $j$th member of the complete set of knots, $(\lambda_j)$ for $j = 1,2,\ldots,n+7$. Constraint: the LAMDA($j$) must be in non-decreasing order with LAMDA(NCAP7-3) $>$ LAMDA(4) and satisfy

$LAMDA(1)=LAMDA(2)=LAMDA(3)=LAMDA(4)$

and

$LAMDA(NCAP7-3)=LAMDA(NCAP7-2)=LAMDA(NCAP7-1)=LAMDA(NCAP7)$.

3: C(NCAP7) -- DOUBLE PRECISION array Input

On entry: the coefficient $c_i$ of the B-spline $N_i(x)$, for $i = 1,2,\ldots,n+3$. The remaining elements of the array are not used.
4: DEFINT -- DOUBLE PRECISION

On exit: the value of the definite integral of s(x) between
the limits x=a and x=b, where a=(lambda) and b=(lambda) .

5: IFAIL -- INTEGER

On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
NCAP7 < 8, i.e., the number of intervals is not positive.

IFAIL= 2
At least one of the following restrictions on the knots is
violated:

\[ \lambda(NCAP7-3) > \lambda(4), \]
\[ \lambda(j) \geq \lambda(j-1), \]
for j = 2,3,...,NCAP7, with equality in the cases
j=2,3,4,NCAP7-2,NCAP7-1, and NCAP7.

7. Accuracy

The rounding errors are such that the computed value of the
integral is exact for a slightly perturbed set of B-spline
coefficients c differing in a relative sense from those supplied
by no more than 2.2*(n+3)*machine precision.

8. Further Comments

The time taken by the routine is approximately proportional to

n+7.

9. Example
Determine the definite integral over the interval 0≤x≤6 of a cubic spline having 6 interior knots at the positions (λ)=1, 3, 3, 3, 4, 4, the 8 additional knots 0, 0, 0, 0, 6, 6, 6, 6, and the 10 B-spline coefficients 10, 12, 13, 15, 22, 26, 24, 18, 14, 12.

The input data items (using the notation of Section 5) comprise the following values in the order indicated:

\begin{verbatim}
n
  LAMDA(j) for j = 1, 2, ..., NCAP7

  C(j), for j = 1, 2, ..., NCAP7-3
\end{verbatim}

The example program is written in a general form that will enable the definite integral of a cubic spline having an arbitrary number of knots to be computed. Any number of data sets may be supplied. The only changes required to the program relate to the dimensions of the arrays LAMDA and C.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

E02 -- Curve and Surface Fitting
E02BEF
E02BEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02BEF computes a cubic spline approximation to an arbitrary set of data points. The knots of the spline are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.

2. Specification

\begin{verbatim}
SUBROUTINE E02BEF (START, M, X, Y, W, S, NEST, N, LAMDA, C, FP, WRK, LWRK, IWRK, IFAIL)
\end{verbatim}
3. Description

This routine determines a smooth cubic spline approximation \( s(x) \) to the set of data points \((x_r, y_r), \) with weights \( w_r, \) for \( r = 1, 2, \ldots, m. \)

The spline is given in the B-spline representation

\[
s(x) = \sum_{i=1}^{n-4} c_i N_i(x)
\]

where \( N_i(x) \) denotes the normalised cubic B-spline defined upon

\[
N_i(x) = \frac{1}{w_i} 
\]

The total number \( n \) of these knots and their values \((\lambda_i), \ldots, (\lambda_{i+4})\) are chosen automatically by the routine. The knots \((\lambda_i), \ldots, (\lambda_{i+4})\) are the interior knots; they divide the approximation interval \([x_1, x_m]\) into \( n-7 \) sub-intervals.

The coefficients \( c_1, c_2, \ldots, c_{n-4} \) are then determined as the solution of the following constrained minimization problem:

\[
\text{minimize} \quad \sum_{i=5}^{n-4} \delta_i^2
\]

subject to the constraint

\[
\sum_{r=1}^{m} \epsilon_r \leq S
\]
r=1

where: \((\delta)\) stands for the discontinuity jump in the third order derivative of \(s(x)\) at the interior knot \((\lambda)\),

\(\epsilon\) denotes the weighted residual \(w(y - s(x))\),

and \(S\) is a non-negative number to be specified by the user.

The quantity \((\eta)\) can be seen as a measure of the (lack of) smoothness of \(s(x)\), while closeness of fit is measured through \((\theta)\). By means of the parameter \(S\), 'the smoothing factor', the user will then control the balance between these two (usually conflicting) properties. If \(S\) is too large, the spline will be too smooth and signal will be lost (underfit); if \(S\) is too small, the spline will pick up too much noise (overfit). In the extreme cases the routine will return an interpolating spline \((\theta=0)\) if \(S\) is set to zero, and the weighted least-squares cubic polynomial \((\eta=0)\) if \(S\) is set very large. Experimenting with \(S\) values between these two extremes should result in a good compromise. (See Section 8.2 for advice on choice of \(S\).)

The method employed is outlined in Section 8.3 and fully described in Dierckx [1], [2] and [3]. It involves an adaptive strategy for locating the knots of the cubic spline (depending on the function underlying the data and on the value of \(S\)), and an iterative method for solving the constrained minimization problem once the knots have been determined.

Values of the computed spline, or of its derivatives or definite integral, can subsequently be computed by calling E02BBF, E02BCF or E02BDF, as described in Section 8.4.

4. References


5. Parameters

1: START -- CHARACTER*1          Input
   On entry: START must be set to 'C' or 'W'.
   If START = 'C' (Cold start), the routine will build up the
   knot set starting with no interior knots. No values need be
   assigned to the parameters N, LAMDA, WRK or IWRK.
   If START = 'W' (Warm start), the routine will restart the
   knot-placing strategy using the knots found in a previous
   call of the routine. In this case, the parameters N, LAMDA,
   WRK, and IWRK must be unchanged from that previous call.
   This warm start can save much time in searching for a
   satisfactory value of S. Constraint: START = 'C' or 'W'.

2: M -- INTEGER          Input
   On entry: m, the number of data points. Constraint: M >= 4.

3: X(M) -- DOUBLE PRECISION array          Input
   On entry: the values x of the independent variable
   (abscissa) x, for r=1,2,...,m. Constraint: x <x <...<x
   1 2 m

4: Y(M) -- DOUBLE PRECISION array          Input
   On entry: the values y of the dependent variable
   (ordinate) y, for r=1,2,...,m.

5: W(M) -- DOUBLE PRECISION array          Input
   On entry: the values w of the weights, for r=1,2,...,m.
   For advice on the choice of weights, see the Chapter
   Introduction, Section 2.1.2. Constraint: W(r) > 0, for
   r=1,2,...,m.

6: S -- DOUBLE PRECISION          Input
   On entry: the smoothing factor, S.
   If S=0.0, the routine returns an interpolating spline.
   If S is smaller than machine precision, it is assumed equal
   to zero.
   For advice on the choice of S, see Section 3 and Section 8.2
Constraint: $S \geq 0.0$.

7: NEST -- INTEGER  
   Input
   On entry: an over-estimate for the number, $n$, of knots 
   required. Constraint: NEST $\geq 8$. In most practical 
   situations, NEST = M/2 is sufficient. NEST never needs to be 
   larger than M + 4, the number of knots needed for 
   interpolation ($S = 0.0$).

8: N -- INTEGER  
   Input/Output
   On entry: if the warm start option is used, the value of N 
   must be left unchanged from the previous call. On exit: the 
   total number, $n$, of knots of the computed spline.

9: LAMDA(NEST) -- DOUBLE PRECISION array  
   Input/Output
   On entry: if the warm start option is used, the values 
   LAMDA(1), LAMDA(2),...,LAMDA(N) must be left unchanged from 
   the previous call. On exit: the knots of the spline i.e., 
   the positions of the interior knots LAMDA(5), LAMDA(6),... 
   ,LAMDA(N-4) as well as the positions of the additional knots 
   LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = x  and 
   
   \[
   \text{LAMDA}(N-3) = \text{LAMDA}(N-2) = \text{LAMDA}(N-1) = \text{LAMDA}(N) = x \quad \text{needed} 
   \]
   \[
   \text{m} 
   \]
   for the B-spline representation.

10: C(NEST) -- DOUBLE PRECISION array  
    Output
    On exit: the coefficient $c_i$ of the B-spline $N_i(x)$ in the 
    spline approximation $s(x)$, for $i=1,2,...,n-4$.

11: FP -- DOUBLE PRECISION  
    Output
    On exit: the sum of the squared weighted residuals, $(\theta)$, 
    of the computed spline approximation. If FP = 0.0, this is 
    an interpolating spline. FP should equal $S$ within a relative 
    tolerance of 0.001 unless $n=8$ when the spline has no 
    interior knots and so is simply a cubic polynomial. For 
    knots to be inserted, $S$ must be set to a value below the 
    value of FP produced in this case.

12: WRK(LWRK) -- DOUBLE PRECISION array  
    Workspace
    On entry: if the warm start option is used, the values WRK 
    (1),...,WRK(n) must be left unchanged from the previous 
    call.

13: LWRK -- INTEGER  
    Input
    On entry: 
    the dimension of the array WRK as declared in the 
    (sub)program from which E02BEF is called.
Constraint: $\text{LWRK} \geq 4 \times M + 16 \times \text{NEST} + 41$.

14: IWRK(NEST) -- INTEGER array

On entry: if the warm start option is used, the values IWRK(1), ..., IWRK(n) must be left unchanged from the previous call.

This array is used as workspace.

15: IFAIL -- INTEGER

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1

On entry START /= 'C' or 'W',

or $M < 4$,

or $S < 0.0$,

or $S = 0.0$ and $\text{NEST} < M + 4$,

or $\text{NEST} < 8$,

or $\text{LWRK} < 4 \times M + 16 \times \text{NEST} + 41$.

IFAIL = 2

The weights are not all strictly positive.

IFAIL = 3

The values of $X(r)$, for $r=1,2,...,M$, are not in strictly increasing order.

IFAIL = 4

The number of knots required is greater than NEST. Try increasing NEST and, if necessary, supplying larger arrays for the parameters LAMDA, C, WRK and IWRK. However, if NEST is already large, say NEST > $M/2$, then this error exit may indicate that S is too small.
IFAIL= 5
The iterative process used to compute the coefficients of
the approximating spline has failed to converge. This error
exit may occur if S has been set very small. If the error
persists with increased S, consult NAG.

If IFAIL = 4 or 5, a spline approximation is returned, but it
fails to satisfy the fitting criterion (see (2) and (3) in
Section 3) - perhaps by only a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its
weighted sum of squared residuals FP is equal to the smoothing
factor S, up to a specified relative tolerance of 0.001 - except
that if n=8, FP may be significantly less than S: in this case
the computed spline is simply a weighted least-squares polynomial
approximation of degree 3, i.e., a spline with no interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02BEF depends on the complexity of
the shape of the data, the value of the smoothing factor S, and
the number of data points. If E02BEF is to be called for
different values of S, much time can be saved by setting START =

8.2. Choice of S

If the weights have been correctly chosen (see Section 2.1.2 of
the Chapter Introduction), the standard deviation of y would
be the same for all r, equal to (sigma), say. In this case,
choosing the smoothing factor S in the range (sigma) (m+-\sqrt{2m}),
as suggested by Reinsch [4], is likely to give a good start in
the search for a satisfactory value. Otherwise, experimenting
with different values of S will be required from the start,
taking account of the remarks in Section 3.

In that case, in view of computation time and memory
requirements, it is recommended to start with a very large value
for S and so determine the least-squares cubic polynomial; the
value returned for FP, call it FP , gives an upper bound for S.

Then progressively decrease the value of S to obtain closer fits
- say by a factor of 10 in the beginning, i.e., S=FP /10, S=FP
/100, and so on, and more carefully as the approximation shows more details.

The number of knots of the spline returned, and their location, generally depend on the value of $S$ and on the behaviour of the function underlying the data. However, if E02BEF is called with $	ext{START} = 'W'$, the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of $S$ and $	ext{START} = 'W'$, a fit can finally be accepted as satisfactory, it may be worthwhile to call E02BEF once more with the selected value for $S$ but now using $	ext{START} = 'C'$. Often, E02BEF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.3. Outline of Method Used

If $S=0$, the requisite number of knots is known in advance, i.e., $n=m+4$; the interior knots are located immediately as $(\lambda)_i = \frac{x_{i-2}}{x_{i-1}}$, for $i=5,6,...,n-4$. The corresponding least-squares spline (see E02BAF) is then an interpolating spline and therefore a solution of the problem.

If $S>0$, a suitable knot set is built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a spline is fitted to the data by least-squares (see E02BAF) and $(\theta)$, the weighted sum of squares of residuals, is computed. If $(\theta)>S$, new knots are added to the knot set to reduce $(\theta)$ at the next stage. The new knots are located in intervals where the fit is particularly poor, their number depending on the value of $S$ and on the progress made so far in reducing $(\theta)$. Sooner or later, we find that $(\theta)\leq S$ and at that point the knot set is accepted. The routine then goes on to compute the (unique) spline which has this knot set and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has $(\theta)=S$. The routine computes the spline by an iterative scheme which is ended when $(\theta)=S$ within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in E02BAF.

An exception occurs when the routine finds at the start that, even with no interior knots ($n=8$), the least-squares spline already has its weighted sum of squares of residuals $\leq S$. In this case, since this spline (which is simply a cubic polynomial) also has an optimal value for the smoothness measure $(\eta)$, namely
zero, it is returned at once as the (trivial) solution. It will usually mean that \( S \) has been chosen too large.

For further details of the algorithm and its use, see Dierckx [3]

8.4. Evaluation of Computed Spline

The value of the computed spline at a given value \( X \) may be obtained in the double precision variable \( S \) by the call:

\[
\text{CALL } \text{E02BBF}(N,LAMDA,C,X,S,IFAIL)
\]

where \( N \), \( LAMDA \) and \( C \) are the output parameters of \text{E02BEF}.

The values of the spline and its first three derivatives at a given value \( X \) may be obtained in the double precision array \( SDIF \) of dimension at least 4 by the call:

\[
\text{CALL } \text{E02BCF}(N,LAMDA,C,X,LEFT,SDIF,IFAIL)
\]

where if \( LEFT = 1 \), left-hand derivatives are computed and if \( LEFT \neq 1 \), right-hand derivatives are calculated. The value of \( LEFT \) is only relevant if \( X \) is an interior knot.

The value of the definite integral of the spline over the interval \( X(1) \) to \( X(M) \) can be obtained in the double precision variable \( SINT \) by the call:

\[
\text{CALL } \text{E02BDF}(N,LAMDA,C,SINT,IFAIL)
\]

9. Example

This example program reads in a set of data values, followed by a set of values of \( S \). For each value of \( S \) it calls \text{E02BEF} to compute a spline approximation, and prints the values of the knots and the B-spline coefficients \( c_i \).

The program includes code to evaluate the computed splines, by calls to \text{E02BBF}, at the points \( x \) and at points mid-way between them. These values are not printed out, however; instead the results are illustrated by plots of the computed splines, together with the data points (indicated by *) and the positions of the knots (indicated by vertical lines): the effect of decreasing \( S \) can be clearly seen. (The plots were obtained by calling NAG Graphical Supplement routine J06FAF(*).)
Please see figures in printed Reference Manual

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

SUBROUTINE E02DAF (M, PX, PY, X, Y, F, W, LAMDA, MU,
1  POINT, NPOINT, DL, C, NC, WS, NWS, EPS,
2  SIGMA, RANK, IFAIL)
INTEGER M, PX, PY, POINT(NPOINT), NPOINT, NC, NWS,
1  RANK, IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), W(M), LAMDA(PX), MU(PY),
1  DL(NC), C(NC), WS(NWS), EPS, SIGMA

This routine determines a bicubic spline fit \( s(x,y) \) to the set of data points \( (x_r, y_r, f_r) \) with weights \( w_r \), for \( r=1,2,...,m \). The two sets of internal knots of the spline, \{\( \lambda \}\} and \{\( \mu \}\}, associated with the variables \( x \) and \( y \) respectively, are prescribed by the user. These knots can be thought of as dividing the data region of the \( (x,y) \) plane into panels (see diagram in Section 5). A bicubic spline consists of a separate bicubic polynomial in each panel, the polynomials joining together with continuity up to the second derivative across the panel boundaries.

\( s(x,y) \) has the property that (\( \text{Sigma} \)), the sum of squares of its weighted residuals (\( \rho_r \)), for \( r=1,2,...,m \), where

\[
(\rho_r) = w_r (s(x_r,y_r)-f_r),
\]

(1)
is as small as possible for a bicubic spline with the given knot sets. The routine produces this minimized value of \((\Sigma)\) and the coefficients \(c_{ij}\) in the B-spline representation of \(s(x,y)\) - see Section 8. E02DEF and E02DFF are available to compute values of the fitted spline from the coefficients \(c_{ij}\).

The least-squares criterion is not always sufficient to determine the bicubic spline uniquely: there may be a whole family of splines which have the same minimum sum of squares. In these cases, the routine selects from this family the spline for which the sum of squares of the coefficients \(c_{ij}\) is smallest: in other words, the minimal least-squares solution. This choice, although arbitrary, reduces the risk of unwanted fluctuations in the spline fit. The method employed involves forming a system of \(m\) linear equations in the coefficients \(c_{ij}\) and then computing its least-squares solution, which will be the minimal least-squares solution when appropriate. The basis of the method is described in Hayes and Halliday [4]. The matrix of the equation is formed using a recurrence relation for B-splines which is numerically stable (see Cox [1] and de Boor [2] - the former contains the more elementary derivation but, unlike [2], does not cover the case of coincident knots). The least-squares solution is also obtained in a stable manner by using orthogonal transformations, viz. a variant of Givens rotation (see Gentleman [3]). This requires only one row of the matrix to be stored at a time. Advantage is taken of the stepped-band structure which the matrix possesses when the data points are suitably ordered, there being at most sixteen non-zero elements in any row because of the definition of B-splines. First the matrix is reduced to upper triangular form and then the diagonal elements of this triangle are examined in turn. When an element is encountered whose square, divided by the mean squared weight, is less than a threshold (epsilon), it is replaced by zero and the rest of the elements in its row are reduced to zero by rotations with the remaining rows. The rank of the system is taken to be the number of non-zero diagonal elements in the final triangle, and the non-zero rows of this triangle are used to compute the minimal least-squares solution. If all the diagonal elements are non-zero, the rank is equal to the number of coefficients \(c_{ij}\) and the solution obtained is the ordinary least-squares solution, which is unique in this case.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number of data points, m. Constraint: M > 1.

2: PX -- INTEGER Input

3: PY -- INTEGER Input
   On entry: the total number of knots (lambda) and (mu)
   associated with the variables x and y, respectively.
   Constraint: PX >= 8 and PY >= 8.

   (They are such that PX-8 and PY-8 are the corresponding
   numbers of interior knots.) The running time and storage
   required by the routine are both minimized if the axes are
   labelled so that PY is the smaller of PX and PY.

4: X(M) -- DOUBLE PRECISION array Input

5: Y(M) -- DOUBLE PRECISION array Input

6: F(M) -- DOUBLE PRECISION array Input
   On entry: the co-ordinates of the data point (x ,y ,f ), for
   r r r
   r=1,2,...,m. The order of the data points is immaterial, but
   see the array POINT, below.

7: W(M) -- DOUBLE PRECISION array Input
   On entry: the weight w of the rth data point. It is
   r important to note the definition of weight implied by the
   equation (1) in Section 3, since it is also common usage to
   define weight as the square of this weight. In this routine,
   each w should be chosen inversely proportional to the
   r
(absolute) accuracy of the corresponding \( f \), as expressed, for example, by the standard deviation or probable error of the \( f \). When the \( f \) are all of the same accuracy, all the \( w \) may be set equal to 1.0.

8: LAMDA(PX) -- DOUBLE PRECISION array Input/Output
On entry: LAMDA(i+4) must contain the \( i \)th interior knot (\( \lambda \)) associated with the variable \( x \), for 
\[
i = 1, 2, \ldots, PX-8.\]
The knots must be in non-decreasing order and lie strictly within the range covered by the data values of \( x \). A knot is a value of \( x \) at which the spline is allowed to be discontinuous in the third derivative with respect to \( x \), though continuous up to the second derivative. This degree of continuity can be reduced, if the user requires, by the use of coincident knots, provided that no more than four knots are chosen to coincide at any point. Two, or three, coincident knots allow loss of continuity in, respectively, the second and first derivative with respect to \( x \) at the value of \( x \) at which they coincide. Four coincident knots split the spline surface into two independent parts. For choice of knots see Section 8. On exit: the interior knots LAMDA(5) to LAMDA(PX-4) are unchanged, and the segments LAMDA(1:4) and LAMDA(PX-3:PX) contain additional (exterior) knots introduced by the routine in order to define the full set of \( B \)-splines required. The four knots in the first segment are all set equal to the lowest data value of \( x \) and the other four additional knots are all set equal to the highest value: there is experimental evidence that coincident end-knots are best for numerical accuracy. The complete array must be left undisturbed if E02DEF or E02DFF is to be used subsequently.

9: MU(PY) -- DOUBLE PRECISION array Input
On entry: MU(i+4) must contain the \( i \)th interior knot (\( \mu \)) associated with the variable \( y \), \( i = 1, 2, \ldots, PY-8. \) The same remarks apply to MU as to LAMDA above, with \( Y \) replacing \( X \), and \( y \) replacing \( x \).

10: POINT(NPOINT) -- INTEGER array Input
On entry: indexing information usually provided by E02ZAF which enables the data points to be accessed in the order which produces the advantageous matrix structure mentioned in Section 3. This order is such that, if the \( (x,y) \) plane is thought of as being divided into rectangular panels by the two sets of knots, all data in a panel occur before data in succeeding panels, where the panels are numbered from bottom
to top and then left to right with the usual arrangement of axes, as indicated in the diagram.

Please see figure in printed Reference Manual

A data point lying exactly on one or more panel sides is considered to be in the highest numbered panel adjacent to the point. E02ZAF should be called to obtain the array POINT, unless it is provided by other means.

11: NPOINT -- INTEGER Input
On entry:
the dimension of the array POINT as declared in the (sub)program from which E02DAF is called.
Constraint: NPOINT >= M + (PX-7)*(PY-7).

12: DL(NC) -- DOUBLE PRECISION array Output
On exit: DL gives the squares of the diagonal elements of the reduced triangular matrix, divided by the mean squared weight. It includes those elements, less than (epsilon), which are treated as zero (see Section 3).

13: C(NC) -- DOUBLE PRECISION array Output
On exit: C gives the coefficients of the fit. C((PY-4)*(i-1)+j) is the coefficient \( c_{ij} \) of Section 3 and Section 8 for \( i,j = 1,2,\ldots,\text{PX}-4 \) and \( i,j = 1,2,\ldots,\text{PY}-4 \). These coefficients are used by E02DEF or E02DFF to calculate values of the fitted function.

14: NC -- INTEGER Input
On entry: the value \((\text{PX}-4)*(\text{PY}-4)\).

15: WS(NWS) -- DOUBLE PRECISION array Workspace

16: NWS -- INTEGER Input
On entry:
the dimension of the array WS as declared in the (sub)program from which E02DAF is called.
Constraint: NWS>=(2*NC+1)*(3*PY-6)-2.

17: EPS -- DOUBLE PRECISION Input
On entry: a threshold (epsilon) for determining the effective rank of the system of linear equations. The rank is determined as the number of elements of the array DL (see below) which are non-zero. An element of DL is regarded as zero if it is less than (epsilon). Machine precision is a suitable value for (epsilon) in most practical applications which have only 2 or 3 decimals accurate in data. If some coefficients of the fit prove to be very large compared with
the data ordinates, this suggests that (epsilon) should be increased so as to decrease the rank. The array DL will give a guide to appropriate values of (epsilon) to achieve this, as well as to the choice of (epsilon) in other cases where some experimentation may be needed to determine a value which leads to a satisfactory fit.

18: SIGMA -- DOUBLE PRECISION Output
On exit: (Sigma), the weighted sum of squares of residuals. This is not computed from the individual residuals but from the right-hand sides of the orthogonally-transformed linear equations. For further details see Hayes and Halliday [4] page 97. The two methods of computation are theoretically equivalent, but the results may differ because of rounding error.

19: RANK -- INTEGER Output
On exit: the rank of the system as determined by the value of the threshold (epsilon). When RANK = NC, the least-squares solution is unique: in other cases the minimal least-squares solution is computed.

20: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one set of knots is not in non-decreasing order, or an interior knot is outside the range of the data values.

IFAIL= 2
More than four knots coincide at a single point, possibly because all data points have the same value of x (or y) or because an interior knot coincides with an extreme data value.

IFAIL= 3
Array POINT does not indicate the data points in panel order. Call E02ZAF to obtain a correct array.

IFAIL= 4
On entry M <= 1,
or \[ PX < 8, \]
or \[ PY < 8, \]
or \[ NC /= (PX-4)*(PY-4), \]
or \[ NWS \] is too small,
or \[ NPOINT \] is too small.

IFAIL= 5

All the weights \( w \) are zero or rank determined as zero.

7. Accuracy

The computation of the B-splines and reduction of the observation matrix to triangular form are both numerically stable.

8. Further Comments

The time taken by this routine is approximately proportional to \( 2^n \) the number of data points, \( m \), and to \( (3*(PY-4)+4) \).

The B-spline representation of the bicubic spline is

\[
s(x,y) = \sum_{ij} c_{ij} M_i(x) N_j(y)
\]

summed over \( i=1,2,\ldots,PX-4 \) and over \( j=1,2,\ldots,PY-4 \). Here \( M_i(x) \) and \( N_j(y) \) denote normalised cubic B-splines, the former defined on

the knots \( \lambda_i, \lambda_i, \ldots, \lambda_{i+4} \) and the latter on

the knots \( \mu_j, \mu_j, \ldots, \mu_{j+4} \). For further details, see


The choice of the interior knots, which help to determine the spline's shape, must largely be a matter of trial and error. It is usually best to start with a small number of knots and, examining the fit at each stage, add a few knots at a time at places where the fit is particularly poor. In intervals of \( x \) or \( y \) where the surface represented by the data changes rapidly, in
function value or derivatives, more knots will be needed than elsewhere. In some cases guidance can be obtained by analogy with the case of coincident knots: for example, just as three coincident knots can produce a discontinuity in slope, three close knots can produce rapid change in slope. Of course, such rapid changes in behaviour must be adequately represented by the data points, as indeed must the behaviour of the surface generally, if a satisfactory fit is to be achieved. When there is no rapid change in behaviour, equally-spaced knots will often suffice.

In all cases the fit should be examined graphically before it is accepted as satisfactory.

The fit obtained is not defined outside the rectangle

\[(\lambda) \leq x \leq (\lambda), \quad (\mu) \leq y \leq (\mu)\]

4 PX-3 4 PY-3

The reason for taking the extreme data values of x and y for these four knots is that, as is usual in data fitting, the fit cannot be expected to give satisfactory values outside the data region. If, nevertheless, the user requires values over a larger rectangle, this can be achieved by augmenting the data with two artificial data points \((a,c,0)\) and \((b,d,0)\) with zero weight, where \(a \leq x \leq b, \quad c \leq y \leq d\) defines the enlarged rectangle. In the case when the data are adequate to make the least-squares solution unique (\(\text{RANK} = NC\)), this enlargement will not affect the fit over the original rectangle, except for possibly enlarged rounding errors, and will simply continue the bicubic polynomials in the panels bordering the rectangle out to the new boundaries: in other cases the fit will be affected. Even using the original rectangle there may be regions within it, particularly at its corners, which lie outside the data region and where, therefore, the fit will be unreliable. For example, if there is no data point in panel 1 of the diagram in Section 5, the least-squares criterion leaves the spline indeterminate in this panel: the minimal spline determined by the subroutine in this case passes through the value zero at the point \(((\lambda), (\mu))\).

9. Example

This example program reads a value for \((\varepsilon)\), and a set of data points, weights and knot positions. If there are more y knots than x knots, it interchanges the x and y axes. It calls E02ZAF to sort the data points into panel order, E02DAF to fit a bicubic spline to them, and E02DEF to evaluate the spline at the data points.
Finally it prints:

- the weighted sum of squares of residuals computed from the linear equations;
- the rank determined by E02DAF;
- data points, fitted values and residuals in panel order;
- the weighted sum of squares of the residuals;
- the coefficients of the spline fit.

The program is written to handle any number of data sets.

Note: the data supplied in this example is not typical of a realistic problem: the number of data points would normally be much larger (in which case the array dimensions and the value of NWS in the program would have to be increased); and the value of (epsilon) would normally be much smaller on most machines (see Section 5; the relatively large value of 10−6 has been chosen in order to illustrate a minimal least-squares solution when RANK < NC; in this example NC = 24).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E02 -- Curve and Surface Fitting
E02DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02DCF computes a bicubic spline approximation to a set of data values, given on a rectangular grid in the x-y plane. The knots of the spline are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.

2. Specification

```fortran
SUBROUTINE E02DCF (START, MX, X, MY, Y, F, S, NXEST,
```
3. Description

This routine determines a smooth bicubic spline approximation

\[ s(x,y) = \sum_{i=1}^{n-4} \sum_{j=1}^{n-4} c_{ij} M_i(x) N_j(y), \]

where \( M_i(x) \) and \( N_j(y) \) denote normalised cubic B-splines, the

\[ \begin{align*}
& c_{ij} \quad \text{for } i=1,2,...,m \quad \text{and } j=1,2,...,m, \\
& x, y
\end{align*} \]

former defined on the knots \((\lambda)_{i} \) to \((\lambda)_{i+4}\) and the

\[ \begin{align*}
& c_{ij} \quad \text{for } i=1,2,...,m \quad \text{and } j=1,2,...,m, \\
& x, y
\end{align*} \]

latter on the knots \((\mu)_{j} \) to \((\mu)_{j+4}\). For further details, see

\[ \begin{align*}
& c_{ij} \quad \text{for } i=1,2,...,m \quad \text{and } j=1,2,...,m, \\
& x, y
\end{align*} \]

Hayes and Halliday [4] for bicubic splines and de Boor [1] for

normalised B-splines.

The total numbers \( n \) and \( n \) of these knots and their values

\[ \begin{align*}
& x, y
\end{align*} \]

\((\lambda)_{1},...,(\lambda)_{n},...,(\lambda)_{n}\) and \((\mu)_{1},...,(\mu)_{m},...,(\mu)_{m}\) are chosen

\[ \begin{align*}
& x, y
\end{align*} \]

automatically by the routine. The knots \((\lambda)_{1},...,\)

\[ \begin{align*}
& x, y
\end{align*} \]

\((\lambda)_{5}\) and \((\mu)_{5},...,(\mu)_{5}\) are the interior knots; they

\[ \begin{align*}
& x, y
\end{align*} \]

divide the approximation domain \([x_1,x_m] \times [y_1,y_m]\) into \( (n-7) \times (n-7) \) subpanels \([(\lambda)_{1},(\lambda)_{5}] \times [(\mu)_{1},(\mu)_{5}] \).
for $i=4,5,...,n-4$, $j=4,5,...,n-4$. Then, much as in the curve $x$ $y$ case (see E02BEF), the coefficients $c_{ij}$ are determined as the solution of the following constrained minimization problem:

$$\text{minimize} \quad (\eta),$$

subject to the constraint

$$\begin{align*}
\sum_{q=1}^{m} \sum_{r=1}^{m} x_{q} y_{r} (\theta) &= S, \\
&\leq (\epsilon),
\end{align*}$$

where $(\eta)$ is a measure of the (lack of) smoothness of $s(x,y)$. Its value depends on the discontinuity jumps in $s(x,y)$ across the boundaries of the subpanels. It is zero only when there are no discontinuities and is positive otherwise, increasing with the size of the jumps (see Dierckx [2] for details).

$(\epsilon)$ denotes the residual $f - s(x,y)$, and $S$ is a non-negative number to be specified by the user.

By means of the parameter $S$, 'the smoothing factor', the user will then control the balance between smoothness and closeness of fit, as measured by the sum of squares of residuals in (3). If $S$ is too large, the spline will be too smooth and signal will be lost (underfit); if $S$ is too small, the spline will pick up too much noise (overfit). In the extreme cases the routine will return an interpolating spline ($(\theta)=0$) if $S$ is set to zero, and the least-squares bicubic polynomial ($(\eta)=0$) if $S$ is set very large. Experimenting with $S$-values between these two extremes should result in a good compromise. (See Section 8.3 for advice on choice of $S$.)

The method employed is outlined in Section 8.5 and fully described in Dierckx [2] and [3]. It involves an adaptive strategy for locating the knots of the bicubic spline (depending on the function underlying the data and on the value of $S$), and an iterative method for solving the constrained minimization problem once the knots have been determined.
Values of the computed spline can subsequently be computed by calling E02DEF or E02DFF as described in Section 8.6.

4. References


5. Parameters

1: START -- CHARACTER*1 Input

On entry: START must be set to 'C' or 'W'.

If START = 'C' (Cold start), the routine will build up the knot set starting with no interior knots. No values need be assigned to the parameters NX, NY, LAMDA, MU, WRK or IWRK.

If START = 'W' (Warm start), the routine will restart the knot-placing strategy using the knots found in a previous call of the routine. In this case, the parameters NX, NY, LAMDA, MU, WRK and IWRK must be unchanged from that previous call. This warm start can save much time in searching for a satisfactory value of S. Constraint: START = 'C' or 'W'.

2: MX -- INTEGER Input

On entry: m , the number of grid points along the x axis.

Constraint: MX >= 4.

3: X(MX) -- DOUBLE PRECISION array Input

On entry: X(q) must be set to x , the x co-ordinate of the qth grid point along the x axis, for q=1,2,...,m .

Constraint: x <x <...<x .
1 2 m
x

4: MY -- INTEGER
Input
On entry: m, the number of grid points along the y axis.
Constraint: MY >= 4.

5: Y(MY) -- DOUBLE PRECISION array
Input
On entry: Y(r) must be set to y, the y co-ordinate of the
rth grid point along the y axis, for r=1,2,...,m.
Constraint: y < y <...< y.
1 2 m

6: F(MX*MY) -- DOUBLE PRECISION array
Input
On entry: F(m *(q-1)+r) must contain the data value f ,
y q,r for q=1,2,...,m and r=1,2,...,m.
x y

7: S -- DOUBLE PRECISION
Input
On entry: the smoothing factor, S.
If S=0.0, the routine returns an interpolating spline.
If S is smaller than machine precision, it is assumed equal to zero.
For advice on the choice of S, see Section 3 and Section 8.3
Constraint: S >= 0.0.

8: NXEST -- INTEGER
Input

9: NYEST -- INTEGER
Input
On entry: an upper bound for the number of knots n and n
x y required in the x- and y-directions respectively.
In most practical situations, NXEST =m/2 and NYEST m/2 is x y sufficient. NXEST and NYEST never need to be larger than m+4 and m+4 respectively, the numbers of knots needed for x y interpolation (S=0.0). See also Section 8.4. Constraint:
NXEST >= 8 and NYEST >= 8.

10: NX -- INTEGER
Input/Output
11: LAMDA(NXEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values LAMDA(1), LAMDA(2), ..., LAMDA(NX) must be left unchanged from the previous call. On exit: LAMDA contains the complete set of knots (lambda) associated with the x variable, i.e., the interior knots LAMDA(5), LAMDA(6), ..., LAMDA(NX-4) as well as the additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = X(1) and LAMDA(NX-3) = LAMDA(NX-2) = LAMDA(NX-1) = LAMDA(NX) = X(NX) needed for the B-spline representation.

12: NY -- INTEGER Input/Output
On entry: if the warm start option is used, the value of NY must be left unchanged from the previous call. On exit: the total number of knots, n, of the computed spline with respect to the y variable.

13: MU(NYEST) -- DOUBLE PRECISION array Input/Output
On entry: if the warm start option is used, the values MU(1), MU(2), ..., MU(NY) must be left unchanged from the previous call. On exit: MU contains the complete set of knots (mu) associated with the y variable, i.e., the interior knots MU(5), MU(6), ..., MU(NY-4) as well as the additional knots MU(1) = MU(2) = MU(3) = MU(4) = Y(1) and MU(NY-3) = MU(NY-2) = MU(NY-1) = MU(NY) = Y(NY) needed for the B-spline representation.

14: C((NXEST-4)*(NYEST-4)) -- DOUBLE PRECISION array Output
On exit: the coefficients of the spline approximation. C((n-4)*(i-1)+j) is the coefficient c defined in Section 3.

15: FP -- DOUBLE PRECISION Output
On exit: the sum of squared residuals, (theta), of the computed spline approximation. If FP = 0.0, this is an interpolating spline. FP should equal S within a relative tolerance of 0.001 unless NX = NY = 8, when the spline has no interior knots and so is simply a bicubic polynomial. For knots to be inserted, S must be set to a value below the value of FP produced in this case.

16: WRK(LWRK) -- DOUBLE PRECISION array Workspace
On entry: if the warm start option is used, the values WRK (1),...,WRK(4) must be left unchanged from the previous call.

This array is used as workspace.

17: LWRK -- INTEGER
   On entry: the dimension of the array WRK as declared in the (sub)program from which E02DCF is called.
   Constraint: 
   \[ LWRK \geq 4 \times (MX + MY) + 11 \times (NXEST + NYEST) + NXEST \times MY + \max(MY, NXEST) + 54. \]

18: IWRK(LIWRK) -- INTEGER array
   On entry: if the warm start option is used, the values IWRK (1), ..., IWRK(3) must be left unchanged from the previous call.
   This array is used as workspace.

19: LIWRK -- INTEGER
   On entry: the dimension of the array IWRK as declared in the (sub)program from which E02DCF is called.
   Constraint: Liwrk \geq 3 + MX + MY + NXEST + NYEST.

20: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry START /= 'C' or 'W',
   or \[ MX < 4, \]
   or \[ MY < 4, \]
or \( S < 0.0 \),
or \( S = 0.0 \) and \( NXEST < MX + 4 \),
or \( S = 0.0 \) and \( NYEST < MY + 4 \),
or \( NXEST < 8 \),
or \( NYEST < 8 \),
or \( LWRK < 4*(MX+MY)+11*(NXEST+NYEST)+NXEST*MY+\max(MY,NXEST)+54 \),
or \( LIWRK < 3 + MX + MY + NXEST + NYEST \).

\textbf{IFAIL= 2}
The values of \( X(q) \), for \( q = 1,2,\ldots, MX \), are not in strictly
increasing order.

\textbf{IFAIL= 3}
The values of \( Y(r) \), for \( r = 1,2,\ldots, MY \), are not in strictly
increasing order.

\textbf{IFAIL= 4}
The number of knots required is greater than allowed by
\( NXEST \) and \( NYEST \). Try increasing \( NXEST \) and/or \( NYEST \) and, if
necessary, supplying larger arrays for the parameters \( \text{LAMDA} \),
\( \text{MU} \), \( \text{C} \), \( \text{WRK} \) and \( IWRK \). However, if \( NXEST \) and \( NYEST \) are already
large, say \( NXEST > MX/2 \) and \( NYEST > MY/2 \), then this error
exit may indicate that \( S \) is too small.

\textbf{IFAIL= 5}
The iterative process used to compute the coefficients of
the approximating spline has failed to converge. This error
exit may occur if \( S \) has been set very small. If the error
persists with increased \( S \), consult NAG.

If \( \text{IFAIL} = 4 \) or \( 5 \), a spline approximation is returned, but it
fails to satisfy the fitting criterion (see (2) and (3) in
Section 3) -- perhaps by only a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its
sum of squared residuals \( FP \) is equal to the smoothing factor \( S \),
up to a specified relative tolerance of 0.001 -- except that if
\( n = 8 \) and \( n = 8 \), \( FP \) may be significantly less than \( S \): in this case
the computed spline is simply the least-squares bicubic
polynomial approximation of degree 3, i.e., a spline with no
interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02DCF depends on the complexity of the shape of the data, the value of the smoothing factor $S$, and the number of data points. If E02DCF is to be called for different values of $S$, much time can be saved by setting $\text{START} = 'W'$. 

8.2. Weighting of Data Points

E02DCF does not allow individual weighting of the data values. If these were determined to widely differing accuracies, it may be better to use E02DDF. The computation time would be very much longer, however.

8.3. Choice of $S$

If the standard deviation of $f$ is the same for all $q$ and $r$

$$f_{qr}$$

(the case for which this routine is designed - see Section 8.2.) and known to be equal, at least approximately, to $(\sigma)$, say, then following Reinsch [5] and choosing the smoothing factor $S$ in the range $(\sigma) (m^{1/2} - \sqrt{2m})$, where $m = m_{xy}$, is likely to give a good start in the search for a satisfactory value. If the standard deviations vary, the sum of their squares over all the data points could be used. Otherwise experimenting with different values of $S$ will be required from the start, taking account of the remarks in Section 3.

In that case, in view of computation time and memory requirements, it is recommended to start with a very large value for $S$ and so determine the least-squares bicubic polynomial; the value returned for $FP$, call it $FP_0$, gives an upper bound for $S$. Then progressively decrease the value of $S$ to obtain closer fits - say by a factor of 10 in the beginning, i.e., $S = FP_0 / 10$, $S = FP_0 / 100$, and so on, and more carefully as the approximation shows more details.

The number of knots of the spline returned, and their location, generally depend on the value of $S$ and on the behaviour of the function underlying the data. However, if E02DCF is called with $\text{START} = 'W'$, the knots returned may also depend on the smoothing
factors of the previous calls. Therefore if, after a number of trials with different values of $S$ and $\text{START} = 'W'$, a fit can finally be accepted as satisfactory, it may be worthwhile to call E02DCF once more with the selected value for $S$ but now using $\text{START} = 'C'$. Often, E02DCF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.4. Choice of $\text{NXEST}$ and $\text{NYEST}$

The number of knots may also depend on the upper bounds $\text{NXEST}$ and $\text{NYEST}$. Indeed, if at a certain stage in E02DCF the number of knots in one direction (say $n_x$) has reached the value of its upper bound ($\text{NXEST}$), then from that moment on all subsequent knots are added in the other ($y$) direction. Therefore the user has the option of limiting the number of knots the routine locates in any direction. For example, by setting $\text{NXEST} = 8$ (the lowest allowable value for $\text{NXEST}$), the user can indicate that he wants an approximation which is a simple cubic polynomial in the variable $x$.

8.5. Outline of Method Used

If $S=0$, the requisite number of knots is known in advance, i.e., $n_x=m+4$ and $n_y=m+4$; the interior knots are located immediately as $(\lambda_i) = x_{i-2}$ and $(\mu_j) = y_{j-2}$, for $i=5,6,...,n_x-4$ and $j=5,6,...,n_y-4$. The corresponding least-squares spline is then an interpolating spline and therefore a solution of the problem.

If $S>0$, suitable knot sets are built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a bicubic spline is fitted to the data by least-squares, and $(\theta)$, the sum of squares of residuals, is computed. If $(\theta)>S$, new knots are added to one knot set or the other so as to reduce $(\theta)$ at the next stage. The new knots are located in intervals where the fit is particularly poor, their number depending on the value of $S$ and on the progress made so far in reducing $(\theta)$. Sooner or later, we find that $(\theta)\leq S$ and at that point the knot sets are accepted. The routine then goes on to compute the (unique) spline which has these knot sets and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has $(\theta)=S$. The routine computes the spline by an iterative scheme which is ended when $(\theta)=S$ within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of
special form, done in a similarly stable and efficient manner as in E02BAF for least-squares curve fitting.

An exception occurs when the routine finds at the start that, even with no interior knots \((n=n_0=8)\), the least-squares spline already has its sum of residuals \(\leq S\). In this case, since this spline (which is simply a bicubic polynomial) also has an optimal value for the smoothness measure \((\eta)\), namely zero, it is returned at once as the (trivial) solution. It will usually mean that \(S\) has been chosen too large.

For further details of the algorithm and its use see Dierckx [2].

8.6. Evaluation of Computed Spline

The values of the computed spline at the points \((TX(r),TY(r))\), for \(r = 1,2,...,N\), may be obtained in the double precision array FF, of length at least \(N\), by the following code:

```fortran
IFAIL = 0
CALL E02DEF(N,NX,NY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where \(NX, NY, LAMDA, MU\) and \(C\) are the output parameters of E02DCF, \(WRK\) is a double precision workspace array of length at least \(NY-4\), and \(IWRK\) is an integer workspace array of length at least \(NY-4\).

To evaluate the computed spline on a \(KX\) by \(KY\) rectangular grid of points in the \(x-y\) plane, which is defined by the \(x\) co-ordinates stored in \(TX(q)\), for \(q=1,2,...,KX\), and the \(y\) co-ordinates stored in \(TY(r)\), for \(r=1,2,...,KY\), returning the results in the double precision array FG which is of length at least \(KX*KY\), the following call may be used:

```fortran
IFAIL = 0
CALL E02DFF(KX,KY,NX,NY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK, * IWRK,IWRK,IFAIL)
```

where \(NX, NY, LAMDA, MU\) and \(C\) are the output parameters of E02DCF, \(WRK\) is a double precision workspace array of length at least \(LWRK = \min(NWRK1,NWRK2)\), \(NWRK1 = KX*4+NX\), \(NWRK2 = KY*4+NY\), and \(IWRK\) is an integer workspace array of length at least \(LIWRK = KY + NY - 4\) if \(NWRK1 \geq NWRK2\), or \(KX + NX - 4\) otherwise. The result of the spline evaluated at grid point \((q,r)\) is returned in element \((KY*(q-1)+r)\) of the array FG.

9. Example

This example program reads in values of \(MX, MY, x,\) for \(q = 1,2,\).
ordinates $f_{qr}$ defined at the grid points $(x_r,y_q)$. It then calls E02DCF to compute a bicubic spline approximation for one specified value of $S$, and prints the values of the computed knots and B-spline coefficients. Finally it evaluates the spline at a small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**1. Purpose**

E02DDF computes a bicubic spline approximation to a set of scattered data. The knots of the spline are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.

**2. Specification**

```fortran
SUBROUTINE E02DDF (START, M, X, Y, F, W, S, NXEST, NYEST, 1 NX, LAMDA, NY, MU, C, FP, RANK, WRK, 2 LWRK, IWRK, LIWRK, IFAIL)
INTEGER M, NXEST, NYEST, NX, NY, RANK, LWRK, IWRK 1 (LIWRK), LIWRK, IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), W(M), S, LAMDA(NXEST), 1 (MU(NYEST)), C((NXEST-4)*(NYEST-4)), FP, WRK 2 (LWRK)
CHARACTER*1 START
```

**3. Description**

This routine determines a smooth bicubic spline approximation $s(x,y)$ to the set of data points $(x_r,y_r,f_{qr})$ with weights $w_r$, for $r=1,2,\ldots,m$. 

The approximation domain is considered to be the rectangle \([x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]\), where \(x_{\min}\) (\(y_{\min}\)) and \(x_{\max}\) (\(y_{\max}\)) denote the lowest and highest data values of \(x\) (\(y\)).

The spline is given in the B-spline representation

\[
s(x,y) = \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} M_{i}(x)N_{j}(y), \quad (1)
\]

where \(M_{i}(x)\) and \(N_{j}(y)\) denote normalised cubic B-splines, the former defined on the knots \(\lambda_{i}\) to \(\lambda_{i+4}\) and the latter on the knots \(\mu_{j}\) to \(\mu_{j+4}\). For further details, see Hayes and Halliday [4] for bicubic splines and de Boor [1] for normalised B-splines.

The total numbers \(n\) and \(n\) of these knots and their values \(\lambda_{1}, \ldots, \lambda_{n}\) and \(\mu_{1}, \ldots, \mu_{n}\) are chosen automatically by the routine. The knots \(\lambda_{1}, \ldots, 5\)

\(\lambda_{n}\) and \(\mu_{1}, \ldots, \mu_{n}\) are the interior knots; they divide the approximation domain \([x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]\) into \((n-7) \times (n-7)\) subpanels \([\lambda_{i}, \lambda_{i+1}] \times [\mu_{j}, \mu_{j+1}]\), for \(i=4,5,\ldots,n-4; j=4,5,\ldots,n-4\). Then, much as in the curve \(x\) \(y\) case (see E02BEF), the coefficients \(c_{ij}\) are determined as the solution of the following constrained minimization problem:

\[
\text{minimize} \quad (\eta_{i}), \quad (2)
\]

subject to the constraint

\[
m
\]
where: $\eta$ is a measure of the (lack of) smoothness of $s(x,y)$.
Its value depends on the discontinuity jumps in $s(x,y)$ across the boundaries of the subpanels. It is zero only when there are no discontinuities and is positive otherwise, increasing with the size of the jumps (see Dierckx [2] for details).

$\epsilon_r$ denotes the weighted residual $\sum w_r (f - s(x_r,y_r))$, and $S$ is a non-negative number to be specified by the user.

By means of the parameter $S$, 'the smoothing factor', the user will then control the balance between smoothness and closeness of fit, as measured by the sum of squares of residuals in (3). If $S$ is too large, the spline will be too smooth and signal will be lost (underfit); if $S$ is too small, the spline will pick up too much noise (overfit). In the extreme cases the method would return an interpolating spline ($\eta=0$) if $S$ were set to zero, and returns the least-squares bicubic polynomial ($\eta=0$) if $S$ is set very large. Experimenting with $S$-values between these two extremes should result in a good compromise. (See Section 8.2 for advice on choice of $S$.) Note however, that this routine, unlike E02BEF and E02DCF, does not allow $S$ to be set exactly to zero: to compute an interpolant to scattered data, E01SAF or E01SEF should be used.

The method employed is outlined in Section 8.5 and fully described in Dierckx [2] and [3]. It involves an adaptive strategy for locating the knots of the bicubic spline (depending on the function underlying the data and on the value of $S$), and an iterative method for solving the constrained minimization problem once the knots have been determined.

Values of the computed spline can subsequently be computed by calling E02DEF or E02DFF as described in Section 8.6.

4. References


5. Parameters

1: START -- CHARACTER*1
   Input
   On entry: START must be set to 'C' or 'W'.
   If START = 'C' (Cold start), the routine will build up the
   knot set starting with no interior knots. No values need be
   assigned to the parameters NX, NY, LAMDA, MU or WRK.

   If START = 'W' (Warm start), the routine will restart the
   knot-placing strategy using the knots found in a previous
   call of the routine. In this case, the parameters NX, NY,
   LAMDA, MU and WRK must be unchanged from that previous call.
   This warm start can save much time in searching for a
   satisfactory value of S. Constraint: START = 'C' or 'W'.

2: M -- INTEGER
   Input
   On entry: m, the number of data points.
   The number of data points with non-zero weight (see W below)
   must be at least 16.

3: X(M) -- DOUBLE PRECISION array
   Input

4: Y(M) -- DOUBLE PRECISION array
   Input

5: F(M) -- DOUBLE PRECISION array
   Input
   On entry: X(r), Y(r), F(r) must be set to the co-ordinates
   of (x ,y ,f ), the rth data point, for r=1,2,...,m. The
   r r r
   order of the data points is immaterial.

6: W(M) -- DOUBLE PRECISION array
   Input
   On entry: W(r) must be set to w , the rth value in the set
   r
   of weights, for r=1,2,...,m. Zero weights are permitted and
the corresponding points are ignored, except when determining $x$ , $x$ , $y$ and $y$ (see Section 8.4). For advice on the choice of weights, see Section 2.1.2 of the Chapter Introduction. Constraint: the number of data points with non-zero weight must be at least 16.

7: S -- DOUBLE PRECISION Input  
On entry: the smoothing factor, $S$.  
For advice on the choice of $S$, see Section 3 and Section 8.2. Constraint: $S > 0.0$.

8: NXEST -- INTEGER Input  
On entry: an upper bound for the number of knots $n$ and $n$ required in the $x$- and $y$-directions respectively.  
In most practical situations, $\text{NXEST} = \text{NYEST} = 4 + \sqrt{m}/2$ is sufficient. See also Section 8.3. Constraint: $\text{NXEST} \geq 8$ and $\text{NYEST} \geq 8$.

9: NYEST -- INTEGER Input  
On entry: if the warm start option is used, the value of $NY$ must be left unchanged from the previous call. On exit: the total number of knots, $n$, of the computed spline with respect to the $x$ variable.

10: NX -- INTEGER Input/Output  
On entry: if the warm start option is used, the value of $NX$ must be left unchanged from the previous call. On exit: the total number of knots, $n$, of the computed spline with respect to the $x$ variable.

11: LAMDA(NXEST) -- DOUBLE PRECISION array Input/Output  
On entry: if the warm start option is used, the values $\text{LAMDA}(1)$, $\text{LAMDA}(2)$,...,$\text{LAMDA}(\text{NX})$ must be left unchanged from the previous call. On exit: $\text{LAMDA}$ contains the complete set of knots (lambda) associated with the $x$ variable, i.e., the interior knots $\text{LAMDA}(5)$, $\text{LAMDA}(6)$,...,$\text{LAMDA}(\text{NX}-4)$ as well as the additional knots $\text{LAMDA}(1) = \text{LAMDA}(2) = \text{LAMDA}(3) = \text{LAMDA}(4) = x$ and $\text{LAMDA}(\text{NX}-3) = \text{LAMDA}(\text{NX}-2) = \text{LAMDA}(\text{NX}-1) = \min\{\text{LAMDA}(\text{NX})\}$ needed for the $B$-spline representation (where $x$ and $x$ are as described in Section 3).

12: NY -- INTEGER Input/Output  
On entry: if the warm start option is used, the value of $NY$ must be left unchanged from the previous call. On exit: the
total number of knots, \( n \), of the computed spline with respect to the \( y \) variable.

13: \text{MU(NYEST)} -- DOUBLE PRECISION array
\text{Input/Output}

On entry: if the warm start option is used, the values \text{MU(1)}, \text{MU(2)},...,\text{MU(NY)} must be left unchanged from the previous call. On exit: \text{MU} contains the complete set of knots (\( \mu_i \)) associated with the \( y \) variable, i.e., the interior knots \text{MU(5)}, \text{MU(6)},...,\text{MU(NY-4)} as well as the additional knots \text{MU(1)} = \text{MU(2)} = \text{MU(3)} = \text{MU(4)} = \text{y} \text{ and } \text{MU(NY-3)} = \text{MU(NY-2)} = \text{min}
\text{MU(NY-1)} = \text{MU(NY)} = \text{y} \text{ needed for the B-spline}
\text{max}
representation (where \text{y} \text{ and } \text{y} \text{ are as described in min max Section 3}).

14: \text{C((NXEST-4)*(NYEST-4))} -- DOUBLE PRECISION array
\text{Output}

On exit: the coefficients of the spline approximation. \text{C((n-4)*(i-1)+j)} is the coefficient \( c_{ij} \) defined in Section 3.

15: \text{FP} -- DOUBLE PRECISION
\text{Output}

On exit: the weighted sum of squared residuals, (\text{theta}), of the computed spline approximation. \text{FP} should equal \text{S} within a relative tolerance of 0.001 unless \text{NX} = \text{NY} = 8, when the spline has no interior knots and so is simply a bicubic polynomial. For knots to be inserted, \text{S} must be set to a value below the value of \text{FP} produced in this case.

16: \text{RANK} -- INTEGER
\text{Output}

On exit: \text{RANK} gives the rank of the system of equations used to compute the final spline (as determined by a suitable machine-dependent threshold). When \text{RANK} = (\text{NX-4})*(\text{NY-4}), the solution is unique; otherwise the system is rank-deficient and the minimum-norm solution is computed. The latter case may be caused by too small a value of \text{S}.

17: \text{WRK(LWRK)} -- DOUBLE PRECISION array
\text{Workspace}

On entry: if the warm start option is used, the value of \text{WRK(1)} must be left unchanged from the previous call. This array is used as workspace.

18: \text{LWRK} -- INTEGER
\text{Input}

On entry: the dimension of the array \text{WRK} as declared in the (sub)program from which \text{E02DDF} is called.
Constraint: \( LWRK \geq (7u^2v+25w^2)(w+1)+2(u+v+4M)+23w+56 \),

where

\( u=NXEST-4 \), \( v=NYEST-4 \), and \( w=\max(u,v) \).

For some problems, the routine may need to compute the minimal least-squares solution of a rank-deficient system of linear equations (see Section 3). The amount of workspace required to solve such problems will be larger than specified by the value given above, which must be increased by an amount, \( LWRK2 \) say. An upper bound for \( LWRK2 \) is given by \( 4u^2v^2w^2+2u^2v^2+4w^2 \), where \( u, v \) and \( w \) are as above. However, if there are enough data points, scattered uniformly over the approximation domain, and if the smoothing factor \( S \) is not too small, there is a good chance that this extra workspace is not needed. A lot of memory might therefore be saved by assuming \( LWRK2 = 0 \).

19: IWRK(LIWRK) -- INTEGER array       Workspace
   LIWRK -- INTEGER                    Input
   On entry:
   the dimension of the array IWRK as declared in the
   (sub)program from which E02DDF is called.
   Constraint: LIWRK\geq M+2(NXEST-7)(NYEST-7).

21: IFAIL -- INTEGER                  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   On entry START /= 'C' or 'W',
   or the number of data points with non-zero weight <
   16,
   or \( S \leq 0.0 \),
or $\text{NXEST} < 8,$

or $\text{NYEST} < 8,$

or $LWRK < (7u^2v+25w)(w+1)+2(u+v+4M)+23w+56,$
where $u = \text{NXEST} - 4, v = \text{NYEST} - 4$ and $w = \max(u,v),$

or $LIWRK < M+2(\text{NXEST}-7)(\text{NYEST}-7).$

IFAIL = 2
On entry either all the $X(r)$, for $r = 1, 2, \ldots, M$, are equal,
or all the $Y(r)$, for $r = 1, 2, \ldots, M$, are equal.

IFAIL = 3
The number of knots required is greater than allowed by
NXEST and NYEST. Try increasing NXEST and/or NYEST and, if
necessary, supplying larger arrays for the parameters LAMDA,
MU, C, WRK and IWRK. However, if NXEST and NYEST are already

large, say NXEST, NYEST > 4 + $\sqrt{M}/2$, then this error exit
may indicate that $S$ is too small.

IFAIL = 4
No more knots can be added because the number of B-spline
coefficients $(NX-4)(NY-4)$ already exceeds the number of
data points $M$. This error exit may occur if either of $S$ or $M$
is too small.

IFAIL = 5
No more knots can be added because the additional knot would
(quasi) coincide with an old one. This error exit may occur
if too large a weight has been given to an inaccurate data
point, or if $S$ is too small.

IFAIL = 6
The iterative process used to compute the coefficients of
the approximating spline has failed to converge. This error
exit may occur if $S$ has been set very small. If the error
persists with increased $S$, consult NAG.

IFAIL = 7
$LWRK$ is too small; the routine needs to compute the minimal
least-squares solution of a rank-deficient system of linear
equations, but there is not enough workspace. There is no
approximation returned but, having saved the information
contained in NX, LAMDA, NY, MU and WRK, and having adjusted
the value of LWRK and the dimension of array WRK
accordingly, the user can continue at the point the program
was left by calling E02DDF with START = 'W'. Note that the requested value for LWRK is only large enough for the current phase of the algorithm. If the routine is restarted with LWRK set to the minimum value requested, a larger request may be made at a later stage of the computation. See Section 5 for the upper bound on LWRK. On soft failure, the minimum requested value for LWRK is returned in IWRK(1) and the safe value for LWRK is returned in IWRK(2).

If IFAIL = 3, 4, 5 or 6, a spline approximation is returned, but it fails to satisfy the fitting criterion (see (2) and (3) in Section 3 -- perhaps only by a small amount, however.

7. Accuracy

On successful exit, the approximation returned is such that its weighted sum of squared residuals FP is equal to the smoothing factor S, up to a specified relative tolerance of 0.001 -- except that if n = 8 and n = 8, FP may be significantly less than S: in this case the computed spline is simply the least-squares bicubic polynomial approximation of degree 3, i.e., a spline with no interior knots.

8. Further Comments

8.1. Timing

The time taken for a call of E02DDF depends on the complexity of the shape of the data, the value of the smoothing factor S, and the number of data points. If E02DDF is to be called for different values of S, much time can be saved by setting START = It should be noted that choosing S very small considerably increases computation time.

8.2. Choice of S

If the weights have been correctly chosen (see Section 2.1.2 of the Chapter Introduction), the standard deviation of \( w f \) would be the same for all \( r \), equal to \( \sigma \), say. In this case, choosing the smoothing factor S in the range \( \sigma (m+\sqrt{2m}) \), as suggested by Reinsch [6], is likely to give a good start in the search for a satisfactory value. Otherwise, experimenting with different values of S will be required from the start.

In that case, in view of computation time and memory requirements, it is recommended to start with a very large value for S and so determine the least-squares bicubic polynomial; the
value returned for FP, call it $FP_0$, gives an upper bound for $S_0$.

Then progressively decrease the value of $S$ to obtain closer fits - say by a factor of 10 in the beginning, i.e., $S = FP_0 /10$, $S = FP_0 /100$, and so on, and more carefully as the approximation shows more details.

To choose $S$ very small is strongly discouraged. This considerably increases computation time and memory requirements. It may also cause rank-deficiency (as indicated by the parameter RANK) and endanger numerical stability.

The number of knots of the spline returned, and their location, generally depend on the value of $S$ and on the behaviour of the function underlying the data. However, if E02DDF is called with START = 'W', the knots returned may also depend on the smoothing factors of the previous calls. Therefore if, after a number of trials with different values of $S$ and START = 'W', a fit can finally be accepted as satisfactory, it may be worthwhile to call E02DDF once more with the selected value for $S$ but now using START = 'C'. Often, E02DDF then returns an approximation with the same quality of fit but with fewer knots, which is therefore better if data reduction is also important.

8.3. Choice of NXEST and NYEST

The number of knots may also depend on the upper bounds NXEST and NYEST. Indeed, if at a certain stage in E02DDF the number of knots in one direction (say $n_x$) has reached the value of its upper bound (NXEST), then from that moment on all subsequent knots are added in the other ($y$) direction. This may indicate that the value of NXEST is too small. On the other hand, it gives the user the option of limiting the number of knots the routine locates in any direction. For example, by setting NXEST = 8 (the lowest allowable value for NXEST), the user can indicate that he wants an approximation which is a simple cubic polynomial in the variable $x$.

8.4. Restriction of the approximation domain

The fit obtained is not defined outside the rectangle \[ [\lambda_{min}, \lambda_{max}] \times [\mu_{min}, \mu_{max}] \]. The reason for taking $x^{n-3} y^{n-3}$ the extreme data values of $x$ and $y$ for these four knots is that, as is usual in data fitting, the fit cannot be expected to give satisfactory values outside the data region. If, nevertheless,
the user requires values over a larger rectangle, this can be achieved by augmenting the data with two artificial data points \((a,c,0)\) and \((b,d,0)\) with zero weight, where \([a,b] \times [c,d]\) denotes the enlarged rectangle.

8.5. Outline of method used

First suitable knot sets are built up in stages (starting with no interior knots in the case of a cold start but with the knot set found in a previous call if a warm start is chosen). At each stage, a bicubic spline is fitted to the data by least-squares and \((\theta)\), the sum of squares of residuals, is computed. If \((\theta) > S\), a new knot is added to one knot set or the other so as to reduce \((\theta)\) at the next stage. The new knot is located in an interval where the fit is particularly poor. Sooner or later, we find that \((\theta) \leq S\) and at that point the knot sets are accepted. The routine then goes on to compute a spline which has these knot sets and which satisfies the full fitting criterion specified by (2) and (3). The theoretical solution has \((\theta) = S\). The routine computes the spline by an iterative scheme which is ended when \((\theta) = S\) within a relative tolerance of 0.001. The main part of each iteration consists of a linear least-squares computation of special form, done in a similarly stable and efficient manner as in E02DAF. As there also, the minimal least-squares solution is computed wherever the linear system is found to be rank-deficient.

An exception occurs when the routine finds at the start that, even with no interior knots \((N = 8)\), the least-squares spline already has its sum of squares of residuals \(\leq S\). In this case, since this spline (which is simply a bicubic polynomial) also has an optimal value for the smoothness measure \((\eta)\), namely zero, it is returned at once as the (trivial) solution. It will usually mean that \(S\) has been chosen too large.

For further details of the algorithm and its use see Dierckx [2].

8.6. Evaluation of computed spline

The values of the computed spline at the points \((x(r),y(r))\), for \(r = 1,2,\ldots,N\), may be obtained in the double precision array \(F\), of length at least \(N\), by the following code:

```fortran
IFAIL = 0
CALL E02DEF(N,NX,NY,TX,TY,LAMDA,MU,C,FF,WRK,IWRK,IFAIL)
```

where \(N\), \(N\), \(LAMDA\), \(MU\) and \(C\) are the output parameters of E02DDF, \(F\) is a double precision workspace array of length at least
NY-4, and IWRK is an integer workspace array of length at least NY-4.

To evaluate the computed spline on a KX by KY rectangular grid of points in the x-y plane, which is defined by the x co-ordinates stored in TX(q), for q=1,2,...,KX, and the y co-ordinates stored in TY(r), for r=1,2,...,KY, returning the results in the double precision array FG which is of length at least KX*KY, the following call may be used:

```fortran
IFAIL = 0
CALL E02DFF(KX,KY,NX,NY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK,
              * IWRK,LIWRK,IFAIL)
```

where NX, NY, LAMDA, MU and C are the output parameters of E02DDF, WRK is a double precision workspace array of length at least LWRK = min(NWRK1,NWRK2), NWRK1 = KX*4+NX, NWRK2 = KY*4+NY, and IWRK is an integer workspace array of length at least LIWRK = KY + NY - 4 if NWRK1 >= NWRK2, or KX + NX - 4 otherwise. The result of the spline evaluated at grid point (q,r) is returned in element (KY*(q-1)+r) of the array FG.

9. Example

This example program reads in a value of M, followed by a set of M data points (x(r),y(r),f(r)) and their weights w(r). It then calls E02DDF to compute a bicubic spline approximation for one specified value of S, and prints the values of the computed knots and B-spline coefficients. Finally it evaluates the spline at a small sample of points on a rectangular grid.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting

E02DEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
E02DEF calculates values of a bicubic spline from its B-spline representation.

2. Specification

```
SUBROUTINE E02DEF (M, PX, PY, X, Y, LAMDA, MU, C, FF, WRK, 1
   IWRK, IFAIL)
INTEGER M, PX, PY, IWRK(PY-4), IFAIL
DOUBLE PRECISION X(M), Y(M), LAMDA(PX), MU(PY), C((PX-4)*
   1
   (PY-4)), FF(M), WRK(PY-4)
```

3. Description

This routine calculates values of the bicubic spline \( s(x,y) \) at prescribed points \((x_r, y_r)\), for \( r = 1, 2, \ldots, m \), from its augmented knot sets \( \{\lambda_i\} \) and \( \{\mu_j\} \) and from the coefficients \( c_{ij} \), for \( i = 1, 2, \ldots, PX-4; j = 1, 2, \ldots, PY-4 \), in its B-spline representation

\[
  s(x,y) = \sum_{i=0}^{PX-4} \sum_{j=0}^{PY-4} c_{ij} M_i(x) N_j(y)
\]

Here \( M_i(x) \) and \( N_j(y) \) denote normalised cubic B-splines, the former defined on the knots \( \lambda_i \) to \( \lambda_{i+4} \) and the latter on the knots \( \mu_j \) to \( \mu_{j+4} \).

This routine may be used to calculate values of a bicubic spline given in the form produced by E01DAF, E02DAF, E02DCF and E02DDF. It is derived from the routine B2VRE in Anthony et al [1].

4. References


5. Parameters

1: M -- INTEGER
   Input
   On entry: \( m \), the number of points at which values of the
spline are required. Constraint: M \geq 1.

2: PX -- INTEGER
   Input

3: PY -- INTEGER
   Input
   On entry: PX and PY must specify the total number of knots associated with the variables x and y respectively. They are such that PX-8 and PY-8 are the corresponding numbers of interior knots. Constraint: PX \geq 8 and PY \geq 8.

4: X(M) -- DOUBLE PRECISION array
   Input

5: Y(M) -- DOUBLE PRECISION array
   Input
   On entry: X and Y must contain x_{r} and y_{r}, for r=1,2,...,m, respectively. These are the co-ordinates of the points at which values of the spline are required. The order of the points is immaterial. Constraint: X and Y must satisfy

   LAMDA(4) \leq X(r) \leq LAMDA(PX-3)

   and

   MU(4) \leq Y(r) \leq MU(PY-3), for r=1,2,...,m.

   The spline representation is not valid outside these intervals.

6: LAMDA(PX) -- DOUBLE PRECISION array
   Input

7: MU(PY) -- DOUBLE PRECISION array
   Input
   On entry: LAMDA and MU must contain the complete sets of knots \{(\lambda)\} and \{(\mu)\} associated with the x and y variables respectively. Constraint: the knots in each set must be in non-decreasing order, with LAMDA(PX-3) > LAMDA(4) and MU(PY-3) > MU(4).

8: C((PX-4)*(PY-4)) -- DOUBLE PRECISION array
   Input
   On entry: C((PY-4)*(i-1)+j) must contain the coefficient c_{ij} described in Section 3, for i=1,2,...,PX-4; j=1,2,...,PY-4.

9: FF(M) -- DOUBLE PRECISION array
   Output
   On exit: FF(r) contains the value of the spline at the point (x_{r},y_{r}), for r=1,2,...,m.

10: WRK(PY-4) -- DOUBLE PRECISION array
    Workspace
11: IWRK(PY-4) -- INTEGER array
   Workspace

12: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 1,
   or PY < 8,
   or PX < 8.

IFAIL= 2
   On entry the knots in array LAMDA, or those in array MU, are
   not in non-decreasing order, or LAMDA(PX-3) <= LAMDA(4), or
   MU(PY-3) <= MU(4).

IFAIL= 3
   On entry at least one of the prescribed points (x ,y ) lies
   r r
   outside the rectangle defined by LAMDA(4), LAMDA(PX-3) and
   MU(4), MU(PY-3).

7. Accuracy

The method used to evaluate the B-splines is numerically stable,
in the sense that each computed value of s(x ,y ) can be regarded
r r
as the value that would have been obtained in exact arithmetic
from slightly perturbed B-spline coefficients. See Cox [2] for
details.

8. Further Comments

Computation time is approximately proportional to the number of
points, m, at which the evaluation is required.

9. Example
This program reads in knot sets LAMDA(1),..., LAMDA(PX) and MU(1),..., MU(PY), and a set of bicubic spline coefficients c ,
  ij
Following these are a value for m and the co-ordinates (x ,y ),
  r r
for r=1,2,...,m, at which the spline is to be evaluated.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E02 -- Curve and Surface Fitting
---
E02DFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02DFF calculates values of a bicubic spline from its B-spline representation. The spline is evaluated at all points on a rectangular grid.

2. Specification

```fortran
SUBROUTINE E02DFF (MX, MY, PX, PY, X, Y, LAMDA, MU, C, FF, WRK, LWRK, IWRK, LIWRK, IFAIL)
  INTEGER MX, MY, PX, PY, LWRK, IWRK(LIWRK), LIWRK, IFAIL
  DOUBLE PRECISION X(MX), Y(MY), LAMDA(PX), MU(PY), C((PX-4)*(PY-4)), FF(MX*MY), WRK(LWRK)
```

3. Description

This routine calculates values of the bicubic spline s(x,y) on a rectangular grid of points in the x-y plane, from its augmented knot sets \{\{lambda\}\} and \{\{mu\}\} and from the coefficients c ,
  ij
for i=1,2,...,PX-4; j=1,2,...,PY-4, in its B-spline representation

\[
s(x,y) = \sum_{i=1}^{PX-4} \sum_{j=1}^{PY-4} c_{ij} M_i(x) N_j(y).
\]
Here $M(x)$ and $N(y)$ denote normalised cubic B-splines, the
former defined on the knots $(\lambda_i)$ to $(\lambda_i)$ and the
latter on the knots $(\mu_j)$ to $(\mu_j)$.

The points in the grid are defined by co-ordinates $x_q$, for
$q=1,2,\ldots,m$, along the $x$ axis, and co-ordinates $y_r$, for
$r=1,2,\ldots,m$ along the $y$ axis.

This routine may be used to calculate values of a bicubic spline
given in the form produced by E01DAF, E02DAF, E02DCF and E02DDF.
It is derived from the routine B2VRE in Anthony et al [1].

4. References

Approximation Subroutine Library. National Physical
Laboratory.


5. Parameters

1: MX -- INTEGER Input

2: MY -- INTEGER Input

On entry: MX and MY must specify $m$ and $m$ respectively,
the number of points along the $x$ and $y$ axis that define the
rectangular grid. Constraint: $MX \geq 1$ and $MY \geq 1$.

3: PX -- INTEGER Input

4: PY -- INTEGER Input

On entry: PX and PY must specify the total number of knots
associated with the variables $x$ and $y$ respectively. They are
such that $PX-8$ and $PY-8$ are the corresponding numbers of
interior knots. Constraint: $PX \geq 8$ and $PY \geq 8$.

5: X(MX) -- DOUBLE PRECISION array Input

6: Y(MY) -- DOUBLE PRECISION array Input

On entry: X and Y must contain $x_q$, for $q=1,2,\ldots,m$, and $y_r$, for
for \( r=1,2,\ldots,m \), respectively. These are the \( x \) and \( y \) ordinates that define the rectangular grid of points at which values of the spline are required. Constraint: \( X \) and \( Y \) must satisfy

\[
LAMDA(4) \leq X(q) < X(q+1) \leq LAMDA(PX-3), \quad \text{for } q=1,2,\ldots,m-1
\]

\[
\text{and}
\]

\[
MU(4) \leq Y(r) < Y(r+1) \leq MU(PY-3), \quad \text{for } r=1,2,\ldots,m-1.
\]

The spline representation is not valid outside these intervals.

7: LAMDA(PX) -- DOUBLE PRECISION array

On entry: LAMDA and MU must contain the complete sets of knots \( \{ (\lambda) \} \) and \( \{ (\mu) \} \) associated with the \( x \) and \( y \) variables respectively. Constraint: the knots in each set must be in non-decreasing order, with \( LAMDA(PX-3) > LAMDA(4) \) and \( MU(PY-3) > MU(4) \).

8: MU(PY) -- DOUBLE PRECISION array

On entry: \( C((i-1)+j) \) must contain the coefficient described in Section 3, for \( i=1,2,\ldots,PX-4; \)

\[
j=1,2,\ldots,PY-4.
\]

9: C((PX-4)*(PY-4)) -- DOUBLE PRECISION array

On entry: \( C((i-1)+j) \) must contain the coefficient described in Section 3, for \( i=1,2,\ldots,PX-4; \)

\[
j=1,2,\ldots,PY-4.
\]

10: FF(MX*MY) -- DOUBLE PRECISION array

On exit: \( FF(MY*(q-1)+r) \) contains the value of the spline at the point \( (x,y) \), for \( q=1,2,\ldots,m; \)

\[
r=1,2,\ldots,m.
\]

11: WRK(LWRK) -- DOUBLE PRECISION array

Workspace

12: LWRK -- INTEGER

On entry: the dimension of the array WRK as declared in the (sub)program from which E02DFF is called. Constraint: \( LWRK \geq \min(NWRK1,NWRK2) \), where \( NWRK1=4*MX+PX \), \( NWRK2=4*MY+PY \).

13: IWRK(LIWRK) -- INTEGER array

Workspace

14: LIWRK -- INTEGER

On entry:
the dimension of the array IWRK as declared in the
(sub)program from which E02DFF is called.
Constraint: LIWRK >= MY + PY - 4 if NWRK1 > NWRK2, or MX + PX - 4 otherwise, where NWRK1 and NWRK2 are as defined in
the description of argument LWRK.

15: IFAIL -- INTEGER                Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
    On entry MX < 1,
    or MY < 1,
    or PY < 8,
    or PX < 8.

IFAIL= 2
    On entry LWRK is too small,
    or LIWRK is too small.

IFAIL= 3
    On entry the knots in array LAMDA, or those in array MU, are
    not in non-decreasing order, or LAMDA(PX-3) <= LAMDA(4), or
    MU(PY-3) <= MU(4).

IFAIL= 4
    On entry the restriction LAMDA(4) <= X(1) <... < X(MX) <=
    LAMDA(PX-3), or the restriction MU(4) <= Y(1) <... < Y(MY)
    <= MU(PY-3), is violated.

7. Accuracy

The method used to evaluate the B-splines is numerically stable,
in the sense that each computed value of s(x ,y ) can be regarded
  r  r
as the value that would have been obtained in exact arithmetic from slightly perturbed B-spline coefficients. See Cox [2] for details.

8. Further Comments

Computation time is approximately proportional to \( m^2 + 4(m + m) \).

9. Example

This program reads in knot sets \( \text{LAMDA}(1), \ldots, \text{LAMDA}(\text{PX}) \) and \( \text{MU}(1), \ldots, \text{MU}(\text{PY}) \), and a set of bicubic spline coefficients \( c_{ij} \).

Following these are values for \( m \) and the \( x \) co-ordinates \( x_q \), for \( q=1,2,\ldots,m \), and values for \( m \) and the \( y \) co-ordinates \( y_r \), for \( r=1,2,\ldots,m \), defining the grid of points on which the spline is to be evaluated.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
3. Description

Given a matrix $A$ with $m$ rows and $n$ columns ($m \geq n$) and a vector $b$ with $m$ elements, the routine calculates an $l_1$ solution to the over-determined system of equations

$$Ax = b.$$ 

That is to say, it calculates a vector $x$, with $n$ elements, which minimizes the $l_1$-norm (the sum of the absolute values) of the residuals

$$r(x) = \sum_{i=1}^{m} |r_i|,$$

where the residuals $r_i$ are given by

$$r_i = r_{i1} - a_{ij}x_j, \quad i=1,2,\ldots,m, \quad j=1$$

Here $a_{ij}$ is the element in row $i$ and column $j$ of $A$, $b_i$ is the $i$-th element of $b$, and $x_j$ the $j$-th element of $x$. The matrix $A$ need not be of full rank.

Typically in applications to data fitting, data consisting of $m$ points with co-ordinates $(t_i, y_i)$ are to be approximated in the $l_1$-norm by a linear combination of known functions $(\phi_j(t))$,

$$(\alpha_1 \phi_1(t)) + (\alpha_2 \phi_2(t)) + \ldots + (\alpha_n \phi_n(t)).$$

This is equivalent to fitting an $l_1$ solution to the over-determined system of equations

$$n$$
Thus if, for each value of \( i \) and \( j \), the element \( a_{ij} \) of the matrix \( A \) in the previous paragraph is set equal to the value of \( \phi_j(t_i) \) and \( b \) is set equal to \( y_i \), the solution vector \( x \) will contain the required values of the \( \alpha_j \). Note that the independent variable \( t \) above can, instead, be a vector of several independent variables (this includes the case where each \( \phi_i \) is a function of a different variable, or set of variables).

The algorithm is a modification of the simplex method of linear programming applied to the primal formulation of the \( l_1 \) problem (see Barrodale and Roberts [1] and [2]). The modification allows several neighbouring simplex vertices to be passed through in a single iteration, providing a substantial improvement in efficiency.

4. References


5. Parameters

1: \( M \) -- INTEGER
   On entry: the number of equations, \( m \) (the number of rows of the matrix \( A \)). Constraint: \( M \geq n \geq 1 \).

2: \( A(LA,NPLUS2) \) -- DOUBLE PRECISION array
   Input/Output
   On entry: \( A(i,j) \) must contain \( a_{ij} \), the element in the \( i \)th row and \( j \)th column of the matrix \( A \), for \( i=1,2,...,m \) and \( j=1,2,...,n \). The remaining elements need not be set. On exit: \( A \) contains the last simplex tableau generated by the simplex method.
3: LA -- INTEGER  
   On entry: 
   the first dimension of the array A as declared in the 
   (sub)program from which E02GAF is called. 
   Constraint: LA >= M + 2.

4: B(M) -- DOUBLE PRECISION array  
   Input/Output 
   On entry: b, the ith element of the vector b, for 
   \[ i = 1, 2, \ldots, m. \] 
   On exit: the ith residual r corresponding to 
   \[ r \] 
   the solution vector x, for \( i = 1, 2, \ldots, m. \)

5: NPLUS2 -- INTEGER  
   Input 
   On entry: \( n + 2 \), where \( n \) is the number of unknowns (the 
   number of columns of the matrix A). Constraint: 3 <= NPLUS2 
   <= M + 2.

6: TOLER -- DOUBLE PRECISION  
   Input 
   On entry: a non-negative value. In general TOLER specifies 
   a threshold below which numbers are regarded as zero. The 
   recommended threshold value is \((\epsilon)\) where \((\epsilon)\) 
   is the machine precision. The recommended value can be 
   computed within the routine by setting TOLER to zero. If 
   premature termination occurs a larger value for TOLER may 
   result in a valid solution. Suggested value: 0.0.

7: X(NPLUS2) -- DOUBLE PRECISION array  
   Output 
   On exit: \( X(j) \) contains the jth element of the solution 
   vector \( x \), for \( j = 1, 2, \ldots, n. \) The elements \( X(n+1) \) and \( X(n+2) \) 
   are unused.

8: RESID -- DOUBLE PRECISION  
   Output 
   On exit: the sum of the absolute values of the residuals 
   for the solution vector \( x \).

9: IRANK -- INTEGER  
   Output 
   On exit: the computed rank of the matrix A.

10: ITER -- INTEGER  
    Output 
    On exit: the number of iterations taken by the simplex 
    method.

11: IWORK(M) -- INTEGER array  
    Workspace

12: IFAIL -- INTEGER  
    Input/Output 
    On entry: IFAIL must be set to 0, -1 or 1. For users not 
    familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
An optimal solution has been obtained but this may not be unique.

IFAIL= 2
The calculations have terminated prematurely due to rounding errors. Experiment with larger values of TOLER or try scaling the columns of the matrix (see Section 8).

IFAIL= 3
On entry NPLUS2 < 3,

or NPLUS2 > M + 2,

or LA < M + 2.

7. Accuracy

Experience suggests that the computational accuracy of the solution x is comparable with the accuracy that could be obtained by applying Gaussian elimination with partial pivoting to the n equations satisfied by this algorithm (i.e., those equations with zero residuals). The accuracy therefore varies with the conditioning of the problem, but has been found generally very satisfactory in practice.

8. Further Comments

The effects of m and n on the time and on the number of iterations in the Simplex Method vary from problem to problem, but typically the number of iterations is a small multiple of n and the total time taken by the routine is approximately

\[ \frac{2}{m} n \]

proportional to \( mn \).

It is recommended that, before the routine is entered, the columns of the matrix A are scaled so that the largest element in each column is of the order of unity. This should improve the conditioning of the matrix, and also enable the parameter TOLER to perform its correct function. The solution x obtained will then, of course, relate to the scaled form of the matrix. Thus if
the scaling is such that, for each $j=1,2,...,n$, the elements of the $j$th column are multiplied by the constant $k$, the element $x_j$ of the solution vector $x$ must be multiplied by $k$ if it is desired to recover the solution corresponding to the original matrix $A$.

9. Example

Suppose we wish to approximate a set of data by a curve of the form

$$y = Ke^{t} + Le + M$$

where $K$, $L$ and $M$ are unknown. Given values $y_i$ at 5 points $t_i$ we may form the over-determined set of equations for $K$, $L$ and $M$

$$x_i - x_i = e K + e L + M = y_i, \quad i=1,2,\ldots,5.$$  

E02GAF is used to solve these in the $l_1$ sense.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E02 -- Curve and Surface Fitting

E02ZAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E02ZAF sorts two-dimensional data into rectangular panels.

2. Specification

SUBROUTINE E02ZAF (PX, PY, LAMDA, MU, M, X, Y, POINT, NPOINT, ADRES, NADRES, IFAIL)
3. Description

A set of \( m \) data points with rectangular Cartesian co-ordinates \( x, y \) are sorted into panels defined by lines parallel to the y and x axes. The intercepts of these lines on the x and y axes are given in \( \text{LAMDA}(i) \), for \( i = 5, 6, \ldots, \text{PX}-4 \) and \( \text{MU}(j) \), for \( j = 5, 6, \ldots, \text{PY}-4 \), respectively. The subroutine orders the data so that all points in a panel occur before data in succeeding panels, where the panels are numbered from bottom to top and then left to right, with the usual arrangement of axes, as shown in the diagram. Within a panel the points maintain their original order.

Please see figure in printed Reference Manual

A data point lying exactly on one or more panel sides is taken to be in the highest-numbered panel adjacent to the point. The subroutine does not physically rearrange the data, but provides the array \( \text{POINT} \) which contains a linked list for each panel, pointing to the data in that panel. The total number of panels is \((\text{PX}-7) \times (\text{PY}-7)\).

4. References

None.

5. Parameters

1: \( \text{PX} \) -- INTEGER

2: \( \text{PY} \) -- INTEGER

On entry: \( \text{PX} \) and \( \text{PY} \) must specify eight more than the number of intercepts on the x axis and y axis, respectively.

Constraint: \( \text{PX} \geq 8 \) and \( \text{PY} \geq 8 \).

3: \( \text{LAMDA}(\text{PX}) \) -- DOUBLE PRECISION array

On entry: \( \text{LAMDA}(5) \) to \( \text{LAMDA}(\text{PX}-4) \) must contain, in non-decreasing order, the intercepts on the x axis of the sides of the panels parallel to the y axis.

4: \( \text{MU}(\text{PY}) \) -- DOUBLE PRECISION array

On entry: \( \text{MU}(5) \) to \( \text{MU}(\text{PY}-4) \) must contain, in non-decreasing order, the intercepts on the y axis of the sides of the panels parallel to the x axis.
5: M -- INTEGER
    Input
    On entry: the number m of data points.

6: X(M) -- DOUBLE PRECISION array
    Input

7: Y(M) -- DOUBLE PRECISION array
    Input
    On entry: the co-ordinates of the rth data point \((x_r, y_r)\),
    for \(r = 1, 2, \ldots, m\).

8: POINT(NPOINT) -- INTEGER array
    Output
    On exit: for \(i = 1, 2, \ldots, NADRES\), \(POINT(m+i) = I_1\) is the
    index of the first point in panel \(i\), \(POINT(I_1) = I_2\) is the
    index of the second point in panel \(i\) and so on.

    \(POINT(IN) = 0\) indicates that \(X(IN), Y(IN)\) was the last point
    in the panel.

    The co-ordinates of points in panel \(i\) can be accessed in
    turn by means of the following instructions:
    \[
    \begin{align*}
    & IN = M + I \\
    & IN = POINT(IN) \\
    & IF (IN.EQ. 0) GOTO 20 \\
    & XI = X(IN) \\
    & YI = Y(IN) \\
    & . \\
    & . \\
    & . \\
    & GOTO 10 \\
    & GOTO 10
    \end{align*}
    \]

9: NPOINT -- INTEGER
    Input
    On entry:
    the dimension of the array POINT as declared in the
    (sub)program from which E02ZAF is called.
    Constraint: \(NPOINT \geq M + (PX-7) \times (PY-7)\).

10: ADRES(NADRES) -- INTEGER array
    Workspace

11: NADRES -- INTEGER
    Input
    On entry: the value \((PX-7) \times (PY-7)\), the number of panels
    into which the \((x,y)\) plane is divided.

12: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1  
The intercepts in the array LAMDA, or in the array MU, are not in non-decreasing order.

IFAIL= 2  
On entry PX < 8, or PY < 8, or M <= 0, or NADRES /= (PX-7)*(PY-7), or NPOINT < M + (PX-7)*(PY-7).

7. Accuracy

Not applicable.

8. Further Comments

The time taken by this routine is approximately proportional to m*log(NADRES).

This subroutine was written to sort two dimensional data in the manner required by routines E02DAF and E02DBF(*). The first 9 parameters of E02ZAF are the same as the parameters in E02DAF and E02DBF(*) which have the same name.

9. Example

This example program reads in data points and the intercepts of the panel sides on the x and y axes; it calls E02ZAF to set up the index array POINT; and finally it prints the data points in panel order.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagFittingPackage (NAGE02)

Exports:
- e02adf
- e02aef
- e02agf
- e02ahf
- e02ajf
- e02akf
- e02baf
- e02bbf
- e02bcf
- e02bdf
- e02bef
- e02daf
- e02def
- e02ddf
- e02def
- e02dff
- e02gaf
- e02zaf

---

)abbrev package NAGE02 NagFittingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:59 1994
++ Description:
++ This package uses the NAG Library to find a
++ function which approximates a set of data points. Typically the
++ data contain random errors, as of experimental measurement, which
++ need to be smoothed out. To seek an approximation to the data, it
++ is first necessary to specify for the approximating function a
++ mathematical form (a polynomial, for example) which contains a
++ number of unspecified coefficients: the appropriate fitting
++ routine then derives for the coefficients the values which
++ provide the best fit of that particular form. The package deals
++ mainly with curve and surface fitting (i.e., fitting with
++ functions of one and of two variables) when a polynomial or a
++ cubic spline is used as the fitting function, since these cover
++ the most common needs. However, fitting with other functions
++ and/or more variables can be undertaken by means of general
++ linear or nonlinear routines (some of which are contained in
++ other packages) depending on whether the coefficients in the
++ function occur linearly or nonlinearly. Cases where a graph
++ rather than a set of data points is given can be treated simply
++ by first reading a suitable set of points from the graph.
++ The package also contains routines for evaluating,
++ differentiating and integrating polynomial and spline curves and
++ surfaces, once the numerical values of their coefficients have
++ been determined.

NagFittingPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
e02adf : (Integer,Integer,Integer,Matrix DoubleFloat,_,
Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ e02adf(m,kplus1,nrows,x,y,w,ifail)
++ computes weighted least-squares polynomial approximations
++ to an arbitrary set of data points.
++ See \downlink{Manual Page}{manpageXXe02adf}.
e02aef : (Integer,Matrix DoubleFloat,DoubleFloat,Integer) -> Result
++ e02aef(nplus1,a,xcap,ifail)
++ evaluates a polynomial from its Chebyshev-series
++ representation.
++ See \downlink{Manual Page}{manpageXXe02aef}.
e02agf : (Integer,Integer,Integer,DoubleFloat,_,
DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_,
Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,Integer,Integer) -> Result
++ e02agf(m,kplus1,nrows,xmin,xmax,x,y,w,xf,yf,lyf,ip,lwrk,liwrk,ifail)
++ computes constrained weighted least-squares polynomial
++ approximations in Chebyshev-series form to an arbitrary set of
++ data points. The values of the approximations and any number of
++ their derivatives can be specified at selected points.
++ See \downlink{Manual Page}{manpageXXe02agf}.
e02ahf : (Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,_,
Integer,Integer,Integer,Integer,Integer) -> Result
++ e02ahf(np1,xmin,xmax,a,ia1,la,ladif,ifail)
++ determines the coefficients in the Chebyshev-series
++ representation of a derivative of a polynomial given in
++ Chebyshev-series form.
++ See \downlink{Manual Page}{manpageXXe02ahf}.
e02ajf : (Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,_,
Integer,Integer,DoubleFloat,Integer,Integer,Integer) -> Result
++ e02ajf(np1,xmin,xmax,a,ia1,la,qatm1,iaint1,laint,ifail)
++ determines the coefficients in the Chebyshev-series
++ representation of the indefinite integral of a polynomial given
++ in Chebyshev-series form.
++ See \downlink{Manual Page}{manpageXXe02ajf}.
e02akf : (Integer,DoubleFloat,DoubleFloat,Matrix DoubleFloat,_,
Integer,Integer,DoubleFloat,Integer) -> Result
++ e02akf(np1,xmin,xmax,a,ia1,la,x,ifail)
++ evaluates a polynomial from its Chebyshev-series
**representation, allowing an arbitrary index increment for accessing the array of coefficients.
++ See \downlink{Manual Page}{manpageXXe02akf}.

\texttt{e02baf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_ Matrix DoubleFloat,Matrix DoubleFloat,_, Integer) -> Result}  
++ \texttt{e02baf(m,ncap7,x,y,w,lamda,ifail)}  
++ computes a weighted least-squares approximation to an arbitrary set of data points by a cubic splines prescribed by the user. Cubic spline can also be carried out.  
++ See \downlink{Manual Page}{manpageXXe02baf}.

\texttt{e02bbf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,_ Integer) -> Result}  
++ \texttt{e02bbf(ncap7,lamda,c,x,ifail)}  
++ evaluates a cubic spline representation.  
++ See \downlink{Manual Page}{manpageXXe02bbf}.

\texttt{e02bcf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,_ Integer,Integer) -> Result}  
++ \texttt{e02bcf(ncap7,lamda,c,x,left,ifail)}  
++ evaluates a cubic spline and its first three derivatives from its B-spline representation.  
++ See \downlink{Manual Page}{manpageXXe02bcf}.

\texttt{e02bdf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result}  
++ \texttt{e02bdf(ncap7,lamda,c,ifail)}  
++ computes the definite integral from its B-spline representation.  
++ See \downlink{Manual Page}{manpageXXe02bdf}.

\texttt{e02bef : (String,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_ Matrix DoubleFloat,DoubleFloat,Integer,Integer,Integer,_, Matrix DoubleFloat,Integer,Matrix DoubleFloat,_, Matrix DoubleFloat,Integer) -> Result}  
++ \texttt{e02bef(start,m,x,y,w,s,nest,lwrk,n,lamda,ifail,wrk,iwrk)}  
++ computes a cubic spline approximation to an arbitrary set of data points. The knot are located automatically, but a single parameter must be specified to control the trade-off between closeness of fit and smoothness of fit.  
++ See \downlink{Manual Page}{manpageXXe02bef}.

\texttt{e02daf : (Integer,Integer,Integer,Matrix DoubleFloat,_ Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_, Matrix DoubleFloat,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,DoubleFloat,Matrix DoubleFloat,Integer) -> Result}  
++ \texttt{e02daf(m,px,py,x,y,f,w,mu,point,npoint,nc,nws,eps,lamda,ifail)}  
++ forms a minimal, weighted least-squares bicubic spline surface fit with prescribed knots to a given set of data points.  
++ See \downlink{Manual Page}{manpageXXe02daf}.

\texttt{e02dcf : (String,Integer,Matrix DoubleFloat,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_, Matrix DoubleFloat,Integer) -> Result}  
++ \texttt{e02dcf(start,m,x,y,w,s,nest,lwrk,n,lamda,ifail,wrk,iwrk)}  
++ computes a cubic spline approximation to an arbitrary set of data points.
Matrix Integer,Integer) -> Result
++ e02dcf(start,mx,x,my,f,s,nxest,nyest,lwrk,liwrk,nx,_
++ lamda,ny,wrk,ifail)
++ computes a bicubic spline approximation to a set of data
++ values, given on a rectangular grid in the x-y plane. The knots
++ of the spline are located automatically, but a single parameter
++ must be specified to control the trade-off between closeness of
++ fit and smoothness of fit.
++ See \downlink{Manual Page}{manpageXXe02dcf}.
e02ddf : (String,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
++ Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
++ Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ e02ddf(start,m,x,y,f,w,s,nxest,nyest,lwrk,liwrk,nx,_
++ lamda,ny,wrk,ifail)
++ computes a bicubic spline approximation to a set of
++ scattered data are located
++ automatically, but a single parameter must be specified to
++ control the trade-off between closeness of fit and smoothness of
++ fit.
++ See \downlink{Manual Page}{manpageXXe02ddf}.
e02def : (Integer,Integer,Integer,Matrix DoubleFloat,_
++ Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
++ Integer) -> Result
++ e02def(m,px,py,x,y,lamda,mu,c,ifail)
++ calculates values of a bicubic spline
++ representation.
++ See \downlink{Manual Page}{manpageXXe02def}.
e02dff : (Integer,Integer,Integer,Integer,_
++ Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_
++ Matrix DoubleFloat,Integer,Integer,Integer) -> Result
++ e02dff(mx,my,px,py,x,y,lamda,mu,c,lwrk,liwrk,ifail)
++ calculates values of a bicubic spline
++ representation. The spline is evaluated at all points on a
++ rectangular grid.
++ See \downlink{Manual Page}{manpageXXe02dff}.
e02gaf : (Integer,Integer,Integer,DoubleFloat,_
++ Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ e02gaf(m,la,nplus2,toler,a,b,ifail)
++ calculates an l solution to an over-determined system of
++ linear equations.
++ See \downlink{Manual Page}{manpageXXe02gaf}.
e02zaf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
++ Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
++ Integer,Integer) -> Result
++ e02zaf(px,py,lamda,mu,m,x,y,npoint,nadres,ifail)
++ sorts two-dimensional data into rectangular panels.
++ See \downlink{Manual Page}{manpageXXe02zaf}.
Implementation ==>
add
import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Matrix Integer)
import AnyFunctions1(String)

e02adf(mArg: Integer, kplus1Arg: Integer, nrowsArg: Integer, _
  xArg: Matrix DoubleFloat, yArg: Matrix DoubleFloat, _
  wArg: Matrix DoubleFloat, _
  ifailArg: Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02adf",_
    "x":S,"a":S,"s":S,_
    "work1":S,"work2":S]$Lisp,_
    ["a":S,"m":S,"work1":S,"work2":S]$Lisp,_
    ["double":S,["x":S,"m":S]$Lisp,["y":S,"m":S]$Lisp,_
    ,["a":S,"m":S]$Lisp,["x":S,"nrows":S,"kplus1":S]$Lisp,_
    ["s":S,"kplus1":S]$Lisp,["work1":S,_
    ["m":S,"a":S,"s":S,"ifail":S]$Lisp,_
    ["*":S,3$Lisp,"m":S]$Lisp]$Lisp,_
    ["work2":S,["*":S,2$Lisp,"kplus1":S]$Lisp]$Lisp]$Lisp,_
    ,["integer":S,"m":S,"kplus1":S,"nrows":S,_
    "ifail":S]$Lisp]$_
    )$_
    ]$_
    pretend List (Record(key:Symbol,entry:Any))$Result

e02aef(nplus1Arg: Integer, aArg: Matrix DoubleFloat, xcapArg: DoubleFloat, _
  ifailArg: Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02aef",_
    ["p":S]$Lisp,_
    ["double":S,["a":S,"nplus1":S]$Lisp,"xcap":S,_
    ["p":S]$Lisp,_
    ,["integer":S,"nplus1":S,"ifail":S]$Lisp,_
    )$_
    ]$_
    _
e02agf(mArg:Integer,kplus1Arg:Integer,nrowsArg:Integer,
xminArg:DoubleFloat,xmaxArg:DoubleFloat,xArg:Matrix DoubleFloat,
yArg:Matrix DoubleFloat,xfArg:Matrix DoubleFloat,yfArg:Integer,
lyfArg:Integer,ipArg:Matrix Integer,lwrkArg:Integer,liwrkArg:Integer,
ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp,_
"e02agf",_
["m"::S,"kplus1"::S,"nrows"::S,"xmin"::S,"xmax"::S_,
)$Lisp_,
)$Lisp_
[("double":"S","xmin":"S","xmax":"S","x":"S","m":"S"$Lisp_,
["y":"S","m":"S"$Lisp_,"xf":"S","mf":"S"$Lisp_,
["yf":"S","lyf":"S","wrk":"S","lwrk":"S"$Lisp_,
["a":"S","nrows":"S","kplus1":"S"$Lisp_,
["s":"S","kplus1":"S","wrk":"S","iwrk":"S"_]
)$Lisp_,
["integer":"S","m":"S","kplus1":"S","nrows":"S_
,"mf":"S","lyf":"S","iwrk":"S","wrk":"S","iwrk":"S_"
"np1":"S","ifail":"S","iwrk":"S","iwrk":"S"$Lisp]$Lisp_]
)$Lisp_}
[("mArg::Any,kplus1Arg::Any,nrowsArg::Any,xminArg::Any,
 xmaxArg::Any,xfArg::Any,lyfArg::Any,lwrkArg::Any,lyfArg::Any,
 ipArg::Any,ifailArg::Any ]_)
@List Any]@$Lisp)$Lisp_)
pretend List (Record(key:Symbol,entry:Any))$Result

e02ahf(np1Arg:Integer,xminArg:DoubleFloat,xmaxArg:DoubleFloat,
arg:Matrix DoubleFloat,ia1Arg:Integer,laArg:Integer,
badif1Arg:Integer,badifArg:Integer,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp_,
"e02ahf",_
["np1"::S,"xmin"::S,"xmax"::S,"ia1"::S,"la"::S_,
["patm1"::S,"adif"::S]$Lisp_,
["double":"S","xmin":"S","xmax":"S","a":"S","la":"S"$Lisp_,
["badif":"S","badif"::S]$Lisp_,
["integer":"S","np1":"S","ia1"::S,"la":"S","badif1"::S_
,"badif"::S,"ifail"::S]$Lisp_]
)$Lisp_
["patm1"::S,"adif"::S,"ifail"::S]$Lisp_]
[(np1Arg::Any,xminArg::Any,xmaxArg::Any,ia1Arg::Any,laArg::Any,_}
iadif1Arg::Any, ladifArg::Any, ifailArg::Any, aArg::Any) 
pretend List (Record(key:Symbol, entry:Any))$Result

e02ajf(np1Arg: Integer, xminArg: DoubleFloat, xmaxArg: DoubleFloat, _
aArg: Matrix DoubleFloat, ia1Arg: Integer, laArg: Integer, _
qatm1Arg: DoubleFloat, ia int1Arg: Integer, laintArg: Integer, _
ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"e02ajf", _
["np1": S, "xmin": S, "xmax": S, "ia1": S, "la": S_, ", qatm1": S, "ia int1": S, "laint": S, "ifail": S, "a": S, "aint": S]$Lisp_, _
["aint": S]$Lisp_, _
["double": S, "xmin": S, "xmax": S, ["a": S, "la": S]$Lisp_, _
, qatm1": S, ["aint": S, "laint": S]$Lisp]$Lisp_ _, _
["integer": S, "np1": S, "ia1": S, "la": S, "ia int1": S_, _
, "laint": S, "ifail": S]$Lisp_ ]$Lisp_, _
["aint": S, "ifail": S]$Lisp_, _
[(np1Arg::Any, xminArg::Any, xmaxArg::Any, aArg::Any, ifailArg::Any, aArg::Any) 
pretend List (Record(key:Symbol, entry:Any))$Result

e02akf(np1Arg: Integer, xminArg: DoubleFloat, xmaxArg: DoubleFloat, _
aArg: Matrix DoubleFloat, ia1Arg: Integer, laArg: Integer, _
xArg: DoubleFloat, ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"e02akf", _
["np1": S, "xmin": S, "xmax": S, "ia1": S, "la": S_, _
, x": S, "result": S, "ifail": S, "a": S]$Lisp_, _
["result": S]$Lisp_, _
["double": S, "xmin": S, "xmax": S, ["a": S, "la": S]$Lisp_, _
, x": S, "result": S]$Lisp_, _
, ["integer": S, "np1": S, "ia1": S, "la": S, "ifail": S_ _
]$Lisp_, _
["result": S, "ifail": S]$Lisp_, _
[(np1Arg::Any, xminArg::Any, xmaxArg::Any, aArg::Any, ifailArg::Any, aArg::Any) _
xArg::Any, ifailArg::Any, aArg::Any) 
pretend List (Record(key:Symbol, entry:Any))$Result

e02baf(mArg: Integer, ncap7Arg: Integer, xArg: Matrix DoubleFloat, _
yArg: Matrix DoubleFloat, wArg: Matrix DoubleFloat, _
lamdaArg: Matrix DoubleFloat, ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp, _
"e02baf", _
["m": S, "ncap7": S, "ss": S, "ifail": S, "x": S, "y": S, "w": S_, _
"c": S, "lamda": S_}
\begin{verbatim}
"work1":::S,"work2":::S]$Lisp,_
["c":::S,"ss":::S,"work1":::S,"work2":::S]$Lisp,_
["double":::S,["x":::S,"m":::S]$Lisp,["y":::S,"m":::S]$Lisp_,
["w":::S,"m":::S]$Lisp,["c":::S,"ncap7":::S]$Lisp,"ss":::S,_
["lamda":::S,"ncap7":::S]$Lisp,["work1":::S,"m":::S]$Lisp_,
["work2":::S,["*":::S,4$Lisp,"ncap7":::S]$Lisp]$Lisp]$Lisp_,
["integer":::S,"m":::S,"ncap7":::S,"ifail":::S_]
]$Lisp_,
["c":::S,"ss":::S,"lamda":::S,"ifail":::S]$Lisp_,
[([mArg::Any,ncap7Arg::Any,ifailArg::Any,xArg::Any,yArg::Any,_
  wArg::Any,lamdaArg::Any])_]
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))]
pretend List (Record(key:Symbol,entry:Any))]
\end{verbatim}

\begin{verbatim}
e02bbf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
cArg:Matrix DoubleFloat,xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02bbf",_
     ["s":::S]$Lisp_]
    ["double":::S,["lamda":::S,"ncap7":::S]$Lisp_,
     ["c":::S,"ncap7":::S]$Lisp,"x":::S,"s":::S]$Lisp_,
     ["integer":::S,"ncap7":::S,"ifail":::S]$Lisp_]
  ]$Lisp_,
  ["s":::S,"ifail":::S]$Lisp_,
  [([ncap7Arg::Any,xArg::Any,ifailArg::Any,lamdaArg::Any,cArg::Any ])_
   @List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))]
\end{verbatim}

\begin{verbatim}
e02bcf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat,_
cArg:Matrix DoubleFloat,xArg:DoubleFloat,leftArg:Integer,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02bcf",_
    ["ncap7":::S,"x":::S,"left":::S,"ifail":::S,"lamda":::S,_
     "c":::S,"s":::S]$Lisp_,
     ["s":::S]$Lisp_]
    ["double":::S,["lamda":::S,"ncap7":::S]$Lisp_,
     ["integer":::S,"ncap7":::S,"left":::S,"ifail":::S_]
  ]$Lisp_,
  ["s":::S,"ifail":::S]$Lisp_,
  [([ncap7Arg::Any,xArg::Any,leftArg::Any,ifailArg::Any,_
     lamdaArg::Any,cArg::Any ])_
   @List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))]
\end{verbatim}

\begin{verbatim}
e02bdf(ncap7Arg:Integer,lamdaArg:Matrix DoubleFloat, _
cArg:Matrix DoubleFloat, xArg:DoubleFloat, leftArg:Integer, ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "e02bdf",_
    ["ncap7":::S,"x":::S,"left":::S,"ifail":::S,"lamda":::S,_
     "c":::S,"s":::S]$Lisp_,
     ["s":::S]$Lisp_]
    ["double":::S,["lamda":::S,"ncap7":::S]$Lisp_,
     ["integer":::S,"ncap7":::S,"left":::S,"ifail":::S_]
  ]$Lisp_,
  ["s":::S,"ifail":::S]$Lisp_,
  [([ncap7Arg::Any,xArg::Any,leftArg::Any,ifailArg::Any,_
     lamdaArg::Any,cArg::Any ])_
   @List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))]
\end{verbatim}
cArg: Matrix DoubleFloat, 
ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp, 
"e02bdf", 
["ncap7":S,"defint":S,"ifail":S,"lamda":S,"c":S]$Lisp, 
["defint":S]$Lisp, 
["double":S,["lamda":S,"ncap7":S]$Lisp, 
,"c":S,"ncap7":S]$Lisp,"defint":S]$Lisp, 
,["integer":S,"ncap7":S,"ifail":S]$Lisp 
]$Lisp, 
["defint":S,"ifail":S]$Lisp, 
[
]@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

e02bef(startArg: String, mArg: Integer, xArg: Matrix DoubleFloat, 

,yArg: Matrix DoubleFloat, wArg: Matrix DoubleFloat, sArg: DoubleFloat, 

,nestArg: Integer, lwrkArg: Integer, nArg: Integer, 

,lamdaArg: Matrix DoubleFloat, ifailArg: Integer, 

,wrkArg: Matrix Integer): Result ==
[(invokeNagman(NIL$Lisp, 
"e02bef", 
["start":S,"m":S,"s":S,"nest":S,"lwrk":S 
,"wrk":S,"iwrk":S]$Lisp, 
["c":S,"fp":S]$Lisp, 
["double":S,["x":S,"m":S]$Lisp,["y":S,"m":S]$Lisp, 
[,]"w":S,"m":S]$Lisp,"c":S,"nest":S]$Lisp, 
]$Lisp, 
,["integer":S,"m":S,"nest":S,"lwrk":S,"n":S_ 
,"ifail":S,["iwrk":S,"nest":S]$Lisp]$Lisp, 
,["character":S,"start":S]$Lisp_ 
]$Lisp, 
[("startArg":Any,mArg:Integer,sArg:Integer,nestArg:Integer, 

,lamdaArg:Matrix DoubleFloat, ifailArg:Integer ]]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result


e02daf(mArg: Integer, pxArg: Integer, pyArg: Integer, 

,xArg: Matrix DoubleFloat, yArg: Matrix DoubleFloat, 

,fArg: Matrix DoubleFloat, 

,wArg: Matrix DoubleFloat, muArg: Matrix DoubleFloat, 

,pointArg: Matrix Integer, 

,npointArg: Integer, nwsArg: Integer, 

,epsArg: DoubleFloat, lamdaArg: Matrix DoubleDouble, 

,ifailArg: Integer): Result ==
"e02def", _
["m":S,"px"":S,"py"":S,"npoint":S,"nc":S_ 
["double":S,"m":S]$Lisp,"y":S,"m":S]$Lisp_ 
,"m":S,$Lisp,"w":S,"m":S]$Lisp_ 
,"c":S,"nc":S]$Lisp_ 
]$Lisp_ 
["double":S,"m":S,"my":S,"nx":S,"nxest":S"_ 
]"c":S,"fp":S]$Lisp_ 
]"c":S,"nxest":S,"nyest":S,4$Lisp]$Lisp_ 
["*=S,"nyest":S,4$Lisp]$Lisp]$Lisp_ 
]$Lisp_ 
,"ifail":S]$Lisp_ 
["character":S,"start":S]$Lisp_ 
]$Lisp_ 
CHAPTER 15. CHAPTER N

"iwrk":S,"ifail":S$Lisp, _
[(startArg::Any,mxArg::Any,myArg::Any,sArg::Any,nxestArg::Any, _
 nyestArg::Any,lwrkArg::Any,liwrkArg::Any,nxArg::Any,nyArg::Any, _
 ifailArg::Any,xArg::Any,yArg::Any,fArg::Any,lambdaArg::Any, _
 muArg::Any,wrkArg::Any,iwrkArg::Any )]_ 
@List Any$Lisp$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

e02ddf(startArg::String,mArg:Integer,xArg:Matrix DoubleFloat, _
yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat, _
wArg:Matrix DoubleFloat, _
sArg:DoubleFloat,nxestArg:Integer,nyestArg:Integer, _
lwrkArg:Integer,liwrkArg:Integer,nxArg:Integer, _
lambdaArg:Matrix DoubleFloat,nyArg:Integer,muArg:Matrix DoubleFloat, _
wrkArg:Matrix DoubleFloat,ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp, _
 "e02ddf",_ 
 ["start":S,"m":S,"s":S,"nxest":S,"nyest":S_,
  "iwrk":S,"lambda":S,"mu":S,"wrk":S]$Lisp, _
 ["double":S,["x":S,"m":S]$Lisp,["y":S,"m":S]$Lisp_ ,
 ["fp":S,"m":S]$Lisp,["w":S,"m":S]$Lisp,"s":S, _
 ["c":S,["*":S,"nxest":S,4$Lisp]$Lisp,$Lisp_ ,
 ["":S,"nyest":S,4$Lisp]$Lisp]$Lisp,$Lisp_ ,
 ["fp":S,["lambda":S,"nxest":S]$Lisp,["mu":S,"nyest":S]$Lisp_,
 ["wrk":S,"lwrk":S]$Lisp_ ]$Lisp_ ,
 ["integer":S,"m":S,"nxest":S,"nyest":S_,
  "nx":S,"ny":S,"ifail":S]$Lisp_ ,
 ["character":S,"start":S]$Lisp_ ]$Lisp_ ,
  "mu":S,"wrk":S,"ifail":S]$Lisp_ ,
 [(startArg::Any,mArg::Any,sArg::Any,nxestArg::Any,nyestArg::Any_,
  lambdaArg::Any,lwrkArg::Any,liwrkArg::Any,nxArg::Any,nyArg::Any, _
  ifailArg::Any,xArg::Any,yArg::Any,fArg::Any,lambdaArg::Any, _
  muArg::Any,wrkArg::Any ]_ 
@List Any$Lisp$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

e02def(mArg:Integer,pxArg:Integer,pyArg:Integer, _
 xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat, _
lambdaArg:Matrix DoubleFloat, _
muArg:Matrix DoubleFloat,cArg:Matrix DoubleFloat, _
ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp, _
 "e02def",_
  [(invokeNagman(NIL$Lisp,
    "e02dff",
    ["ff"::S,"wrk"::S,"ifail"::S]$Lisp,
    ["double"::S,\"x"::S,"m"::S]$Lisp,[\"y"::S,"m"::S]$Lisp,
    [\"lamda"::S,\"px"::S]$Lisp,[\"mu"::S,\"py"::S]$Lisp,
    [\"c"::S,\"*-\"::S,"px"::S,4$Lisp]$Lisp,$\"py"::S,4$Lisp]$Lisp]
    $Lisp,$Lisp,
    [\"integer"::S,\"m"::S,\"px"::S,\"py"::S,$\"ifail"::S,$\"iwrk"::S]$Lisp,
    [\"double"::S,\"x"::S,\"lwrk"::S]$Lisp,$\"lwrk"::S,\"py"::S]$Lisp]
    $Lisp,$Lisp,
    [\"integer"::S,\"m"::S,\"px"::S,\"py"::S]$Lisp,$\"lwrk"::S,\"ifail"::S]$Lisp,$\"lwrk"::S]$Lisp]
    $Lisp,$Lisp,
    [\"double"::S,\"x"::S,\"y"::S,\"lamda"::S,\"mu"::S,\"c"::S]$Lisp,$\"ff"::S,$\"wrk"::S,$\"ifail"::S]$Lisp]
    $Lisp,$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

e02gaf(mArg:Integer,laArg:Integer,nplus2Arg:Integer,tolerArg:DoubleFloat,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,
    "e02gaf", [\"m"::S,"la"::S,"nplus2"::S,\"toler"::S,$\"a"::S,\"b"::S,$\"ifail"::S]$Lisp]
    $Lisp,$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result
package NAGF04 NagLinearEquationSolvingPackage

| NagLinearEquationSolvingPackage.input |

)set break resume
)sys rm -f NagLinearEquationSolvingPackage.output
)spool NagLinearEquationSolvingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 165
showArrayValues true
--R
--R
--R (1) true
--R
--E 1

--S 2 of 165
showScalarValues true
--R
--R
--R (2) true
--R
--E 2

--S 3 of 165
ia:=3
--R
--R
--R (3) 3
--R
--E 3

--S 4 of 165
b:Matrix Complex SF:= [[1 ,0 ,0 ]]
--R
--R
--R (4) [1. 0. 0.]
--R
--E 4

--S 5 of 165
ib:=3
--R
--R
--R (5) 3
n := 3

m := 1

ic := 3

a := Matrix Complex SF :=
[[1, 1 + 2*%i, 2 + 10*%i],
  [1 + %i, 3*%i, -5 + 14*%i],
  [1 + %i, 5*%i, -8 + 20*%i]]

result := f04adf(ia, ib, n, m, ic, a, -1)

)clear all

showArrayValues true
showScalarValues true

ia:=8

b:Matrix SF:= 

[-359 ,281 ,85 ]

n:=3

a:Matrix SF:=

[[33,16,72],
 [-24,-10,-57],
 [-8,-4,-17],
 [0,0,0],
 [0,0,0],
 [0,0,0],
 [0,0,0],
 [0,0,0]]


\[
\begin{bmatrix}
  33 & 16 & 72 \\
 -24 & -10 & 57 \\
 -8 & -4 & 17 \\
 0 & 0 & 0 \\
 0 & 0 & 0 \\
 0 & 0 & 0
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
\begin{bmatrix}
  23 & 32 & 33 & 31
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)
n:=4

\[
\begin{bmatrix}
5 & 7 & 6 & 5 \\
7 & 10 & 8 & 7 \\
6 & 8 & 10 & 9 \\
5 & 7 & 9 & 10 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

result:=f04asf(ia,b,n,a,-1)

)clear all
showArrayValues true
--R
--R
--R (1) true
--R
--E 25

showScalarValues true
--R
--R
--R (2) true
--R
--E 26

a:Matrix SF:=
[[33,16,72],
 [-24,-10,-57],
 [-8,-4,-17],
 [0,0,0],
 [0,0,0],
 [0,0,0],
 [0,0,0],
 [0,0,0]]

+ 33. 16. 72. +
--R
--R
--R | |
--R |
--R |
--R | 0. 0. 0. |
--R (3) |
--R |
--R |
--R |
--R |
--R |
--R |
--R + 0. 0. 0. +
--R
--E 27

ia:=8
--R
--R
b: Matrix SF := [[-359, 281, 85]]

n := 3

iaa := 8

result := f04atf(a, ia, b, n, 8, -1)
n := 6
Type: PositiveInteger

nz := 15
Type: PositiveInteger

nzmax := 50
Type: PositiveInteger

licn := 3*nzmax
Type: PositiveInteger

t1 := [0.0 for i in 1..(licn-nz)]
Type: List(Float)
t2 := [5.0, 2.0, -1.0, 2.0, 3.0, -2.0, 1.0, 0.0, 1.0, 0.0, -1.0, 0.0, 2.0, -3.0, -1.0, 0.0, 6.0]  
(8)  
Type: List(Float)

---
t3 := concat(t2, t1)  
(9)  
Type: List(Float)

---
a := [t3] :: Matrix SF  
(10)  
Type: Matrix(DoubleFloat)

---
lirn := (3*nzmax/2)::Integer  
---
t1 := [0.0 for i in 1..(lim-nz)]

t2 := [1,2,2,2,3,4,4,4,5,5,5,6,6,6]

t3 := concat(t2,t1)

irn := [t3] :: Matrix Integer
\[ t1 := [0.0 \text{ for } i \text{ in } 1..(licn-nz)] \]

\[ t2 := [1,2,3,4,3,1,4,5,1,4,5,6,1,2,6] \]

\[ t3 := \text{concat}(t2, t1) \]

\[ \text{icn} := [t3] :: \text{Matrix Integer} \]
pivot := 0.1

lblock := true

grow := true

abort := [true,true,false,true]

ifail := 1
--S 57 of 165
-- res1 := f01brf(n, nz, licn, lirn, pivot, lblock, grow, abort, a, irn, icn, ifail)
--E 57

--S 58 of 165
-- a := (res1) \cdot a
--E 58

--S 59 of 165
-- icn := (res1) \cdot icn
--E 59

--S 60 of 165
-- ikeep := (res1) \cdot ikeep
--E 60

--S 61 of 165
rhs := [[15, 12, 18, 3, -6, 0]]
--R
--R
--R (25) [[15, 12, 18, 3, -6, 0]]
--R
--R Type: List(List(Integer))
--E 61

--S 62 of 165
-- idisp := (res1) \cdot idisp
--E 62

--S 63 of 165
mtype := 1
--R
--R
--R (26) 1
--R
--R Type: PositiveInteger
--E 63

--S 64 of 165
result := f04axf(n, a, licn, icn, ikeep, mtype, [[idisp(1,1), idisp(1,2)]], rhs)
--E 64

clear all

--S 65 of 165
showArrayValues true
--R
--R
--R (1) true
--R
--R Type: Boolean
--E 65
showScalarValues true
(2) true
Type: Boolean

job:=0
(3) 0
Type: NonNegativeInteger

n:=5
(4) 5
Type: PositiveInteger

d:Matrix SF:= [[4, 10, 29, 25, 5]]
(5) [4.0, 10.0, 29.0, 25.0, 5.0]
Type: Matrix(DoubleFloat)

e:Matrix SF:= [[0, -2, -6, 15, 8]]
(6) [0.0, -2.0, -6.0, 15.0, 8.0]
Type: Matrix(DoubleFloat)

b:Matrix SF:= [[6, 9, 2, 14, 7]]
(7) [6.0, 9.0, 2.0, 14.0, 7.0]
Type: Matrix(DoubleFloat)
-- result:=f04faf(job,n,d,e,b,-1)
--E 72
)
clear all

--S 73 of 165
showArrayValues true
--R
--R  (1) true
--R  Type: Boolean
--E 73

--S 74 of 165
showScalarValues true
--R
--R  (2) true
--R  Type: Boolean
--E 74

--S 75 of 165
m:=6
--R
--R  (3) 6
--R  Type: PositiveInteger
--E 75

--S 76 of 165
n:=4
--R
--R  (4) 4
--R  Type: PositiveInteger
--E 76

--S 77 of 165
mra:=8
--R
--R  (5) 8
--R  Type: PositiveInteger
--E 77

--S 78 of 165
tol:= 5.0e-4
--R
--R  (6) 0.0005
lwork:=32

\[
\begin{bmatrix}
0.05 & 0.05 & 0.25 & -0.25 \\
0.25 & 0.25 & 0.05 & -0.05 \\
0.35 & 0.35 & 1.75 & -1.75 \\
1.75 & 1.75 & 0.35 & -0.35 \\
0.30 & -0.30 & 0.30 & 0.30 \\
0.40 & -0.40 & 0.40 & 0.40 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
0.0000000000000003 \times 10^{-2} & 0.0000000000000003 \times 10^{-2} & 0.25 & -0.25 \\
0.25 & 0.25 & 0.0000000000000003 \times 10^{-2} & -0.9999999999999996 \times 10^{-2} \\
0.34999999999999998 & 0.34999999999999998 & 1.75 & -1.75 \\
1.75 & 1.75 & 0.34999999999999998 & -0.34999999999999998 \\
0.29999999999999999 & -0.30000000000000004 & 0.29999999999999999 & 0.29999999999999999 \\
0.40000000000000002 & -0.39999999999999997 & 0.40000000000000002 & 0.40000000000000002 \\
0.0 & 0.0 & 0.0 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 3 & 4 & 5 & 6
\end{bmatrix}
\]

\[
\begin{bmatrix}
\end{bmatrix}
\]
clear all

showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean

n := 4;

Type: PositiveInteger

m := 4;

Type: PositiveInteger

nn := n*m;

Type: PositiveInteger

licn := 90;

Type: PositiveInteger

lirn := 50;
a := new(1,licn,0.0)$Matrix DoubleFloat;

irn := new(1,lirn,0)$Matrix Integer;

icn := new(1,licn,0)$Matrix Integer;

k := 0;

for i in 1..nn repeat
  k := k + 1
  a(1,k) := 1
  irn(1,k) := i
  icn(1,k) := i

for i in 1..n repeat
  for j in 1..(m-1) repeat
    k := k + 1
    jj := (i-1)*n + j
    a(1,k) := -0.25
    irn(1,k) := jj
icn(1,k) := jj + 1
--R
--R
--E 95

--S 96 of 165
for i in (n+1)..nn repeat
  k := k + 1
  a(1,k) := -0.25
  irn(1,k) := i-n
  icn(1,k) := i
--R
--R
--E 96

--S 97 of 165
nz := k;
--R
--R
--R
--E 97

--S 98 of 165
abort := [true,true,true];
--R
--R
--R
--E 98

--S 99 of 165
droptl := 0.1;
--R
--R
--R
--E 99

--S 100 of 165
densw := 0.8;
--R
--R
--R
--E 100

--S 101 of 165
ifail := 111;
--R
--R
--R
--E 101
```
-- S 102 of 165
-- res1 := f01maf(nn, nz, licn, lirn, abort, a, irn, icn, droptl, densw, ifail)
-- E 102

-- S 103 of 165
-- a := res1."avals"
-- E 103

-- S 104 of 165
-- irn := res1."irn"
-- E 104

-- S 105 of 165
-- icn := res1."icn"
-- E 105

-- S 106 of 165
b := [[0.5, 0.25, 0.25, 0.5, 0.25, 0.0, 0.0, 0.25, 0.25, 0.0, 0.0, 0.25, 0.5, 0.25, 0.25, 0.5]]
-- R
-- R (20)
-- R [[0.5, 0.25, 0.25, 0.5, 0.25, 0.0, 0.0, 0.25, 0.25, 0.0, 0.0, 0.25, 0.5, 0.25, 0.25, 0.5]]
-- R Type: List(List(Float))
-- E 106

-- S 107 of 165
acc := [[0.0001, 0.0]]
-- R
-- R (21) [[0.0001, 0.0]]
-- R Type: List(List(Float))
-- E 107

-- S 108 of 165
noits := [[50, 0]]
-- R
-- R (22) [[50, 0]]
-- R Type: List(List(NonNegativeInteger))
-- E 108

-- S 109 of 165
-- wkeep := res1."wkeep"
-- E 109

-- S 110 of 165
-- ikeep := res1."ikeep"
-- E 110

-- S 111 of 165
```
-- inform := res1."inform"
--E 111

--S 112 of 165
-- result:=
-- f04maf(nn,nz,a,licn,irn,lirn,icn,wkeep,ikeep,inform,b,acc,noits,ifail)
--E 112

)clear all

--S 113 of 165
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 113

--S 114 of 165
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 114

--S 115 of 165
n := 10
--R
--R
--R (3) 10
--R Type: PositiveInteger
--E 115

--S 116 of 165
b := new(1,n,0.0)$Matrix DoubleFloat;
--R
--R
--R Type: Matrix(DoubleFloat)
--E 116

--S 117 of 165
b(1,1) := 6
--R
--R
--R (5) 6.
--R Type: DoubleFloat
--E 117

--S 118 of 165
for i in 2..n-1 repeat
  b(1,i) := 4
  --R
  --R
--E 118

--S 119 of 165
b(1,n) := 6
--R
--R
--R (7) 6.
--R
--E 119

--S 120 of 165
precon := true
--R
--R
--R (8) true
--R
--E 120

--S 121 of 165
shft := 0
--R
--R
--R (9) 0
--R
--E 121

-- The following two parameters are not used
-- by the ASPs. They may be useful if a user
-- supplies FORTRAN code directly.

--S 122 of 165
lrwork:=1
--R
--R
--R (10) 1
--R
--E 122

--S 123 of 165
liwork:=1
--R
--R
--R (11) 1
--R
--E 123
rtol := 0.00001

(12) 0.00001

Type: Float

itnlim := 100

(13) 100

Type: PositiveInteger

msglvl := 1

(14) 1

Type: PositiveInteger

ifail := 1

(15) 1

Type: PositiveInteger

z:=new(10,10,0.0)$Matrix MFLOAT;

Type: Matrix(MachineFloat)

z(1,1):=2;z(1,2):=1;z(1,10):=3

(17) 3.0

Type: MachineFloat

z(10,1):=3;z(10,9):=1;z(10,10):=2
for i in 2..(n-1) repeat
  z(i,i-1) := 1
  z(i,i) := 2
  z(i,i+1) := 1

m := Matrix(MFLOAT):= [[2,1,0,0,0,0,0,0,0,0],[1,2,1,0,0,0,0,0,0,0],
[0,1,2,1,0,0,0,0,0,0],[0,0,1,2,1,0,0,0,0,0],[0,0,0,1,2,1,0,0,0,0],
[0,0,0,0,1,2,1,0,0,0],[0,0,0,0,0,1,2,1,0,0],[0,0,0,0,0,0,1,2,1,0],
[0,0,0,0,0,0,0,1,2,1],[0,0,0,0,0,0,0,0,1,2]]
```plaintext
PACKAGE NAGF04 NAGLINEAREQUATIONSSOLVINGPACKAGE

--R | |
--R |0.0 1.0 2.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0|
--R |
--R |0.0 0.0 1.0 2.0 1.0 0.0 0.0 0.0 0.0 0.0|
--R |
--R |0.0 0.0 0.0 1.0 2.0 1.0 0.0 0.0 0.0 0.0|
--R |
--R |0.0 0.0 0.0 0.0 1.0 2.0 1.0 0.0 0.0 0.0|
--R |
--R |0.0 0.0 0.0 0.0 0.0 1.0 2.0 1.0 0.0 0.0|
--R |
--R |0.0 0.0 0.0 0.0 0.0 0.0 1.0 2.0 1.0 0.0|
--R |
--R |0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 2.0 +
--R Type: Matrix(MachineFloat)
--E 133

--S 134 of 165
-- result:=f04mbf(n,b,precon,shft,itnlim,msglvl,lrwork,liwork,_
    rtol,ifail,z::ASP28(APROD),(m::ASP34(MSOLVE)))
-- E 134

)clear all

--S 135 of 165
showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 135

--S 136 of 165
showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 136

--S 137 of 165
n:=6
--R
--R (3) 6
--R Type: PositiveInteger
--E 137
```
### Chapter 15.

--- S 138 of 165

\[
\begin{bmatrix}
1.0 & 2.0 & 1.0 & 3.0 & 1.0 & 5.0 & 4.0 & 1.5 & 0.5 & 1.0 & 1.5 & 5.0 & 1.0 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

--- E 138

--- S 139 of 165

\[
14
\]

Type: PositiveInteger

--- E 139

--- S 140 of 165

\[
\begin{bmatrix}
1.0 & 1.0 & 4.0 & 16.0 & 1.0 & 16.0 \\
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

--- E 140

--- S 141 of 165

\[
\begin{bmatrix}
1 & 2 & 2 & 1 & 5 & 3 \\
\end{bmatrix}
\]

Type: Matrix(Integer)

--- E 141

--- S 142 of 165

\[
2
\]

Type: PositiveInteger

--- E 142

--- S 143 of 165

\[
\begin{bmatrix}
6 & -10 \\ 15 & -21 \\ 11 & -3 \\ 0 & 24 \\ 51 & -39 \\ 46 & 67 \\
\end{bmatrix}
\]
package nagf04 naglinearequationsolvingpackage

--r
--r +6. -10. +
--r | |
--r |15. -21. |
--r | |
--r |11. -3. |
--r (9) | |
--r |0. 24. |
--r | |
--r |51. -39. |
--r | |
--r +46. 67. +
--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r

--r (1) true

result:=f04mcf(n,al,lal,d,nrow,ir,b,nrb,iselct,nrx,-1)

)clear all

--s 148 of 165
showarrayvalues true
--r

--r

--r (1) true
showScalarValues true

n := 12

m := 13

h:SF := 0.1

b :Matrix SF := -h^2 * [[0],[0],[0],[1],[1],[0],[0],[1],[1],[0],[0],[0],[-h^-3]]

b =
+ 0.
+ 0.
| 0. |
| 0. |
| 0. |
| 0. |
| 0. |
| - 1.0000000000000004E-2 |
| - 1.0000000000000004E-2 |
| 0. |
--R | 
--R (6) | 0. | 
--R | 
--R | - 1.0000000000000004E-2 | 
--R | 
--R | - 1.0000000000000004E-2 | 
--R | 
--R | 0. | 
--R | 
--R | 0. | 
--R | 
--R | 0. | 
--R | 
--R | 0. | 
--R | 
--R | 0. | 
--R | 
--R | + 10.000000000000002 + 
--R Type: Matrix(DoubleFloat)

--E 153

--S 154 of 165

a : Matrix MachineFloat:=
    [[1,0,0,-1,0,0,0,0,0,0,0,0],
     [0,1,0,-1,0,0,0,0,0,0,0,0],
     [0,0,1,-1,0,0,0,0,0,0,0,0],
     [-1,0,-1,4,-1,0,0,0,0,0,0,0],
     [0,-1,0,-1,4,-1,0,0,0,0,0,0],
     [0,0,0,0,0,0,1,-1,0,0,0,0],
     [0,0,0,0,0,1,-1,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,0],
     [0,0,0,0,0,0,0,0,0,0,0,0],
     [1,1,1,0,0,1,1,0,1,1,1]]

--E 154

Type: Matrix(MachineFloat)
damp := 0.0

atol := 1.0e-5

btol := 1.0e-4

conlim := 1/atol

itnlim := 100

msglvl := 1
liwork := 1  
(--R)
(--R (14) 1)
(--R Type: PositiveInteger)
(--E 161)

lrwork := 1  
(--R)
(--R (15) 1)
(--R Type: PositiveInteger)
(--E 162)

ifail := 1  
(--R)
(--R (16) 1)
(--R Type: PositiveInteger)
(--E 163)

result:=f04qaf(m,n,damp,atol, btol,conlim, itnlim, msglvl, lrwork, 

liwork,b,ifail,a::ASP30(APROD))
(--E 164)

)show NagLinearEquationSolvingPackage
(--R)
(--R NagLinearEquationSolvingPackage is a package constructor
(--R Abbreviation for NagLinearEquationSolvingPackage is NAGF04
(--R This constructor is exposed in this frame.
(--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGF04
(--R)
(--R-------------------------- Operations --------------------------)
(--R f04adf : (Integer,Matrix(Complex(DoubleFloat)),Integer,Integer,Integer,Integer,Matrix(Complex(DoubleFloat)),Integer) -> Result
(--R f04arf : (Integer,Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer) -> Result
(--R f04asf : (Integer,Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer) -> Result
(--R f04atf : (Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer, Integer,Integer) -> Result
(--R f04axf : (Integer,Matrix(DoubleFloat),Integer,Matrix(Integer),Matrix(Complex(DoubleFloat)),Integer) -> Result
(--R f04alf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
(--R f04jgf : (Integer,Integer,Integer,DoubleFloat,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
(--R f04maf : (Integer,Integer,Matrix(DoubleFloat),Integer,Matrix(Integer),Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat), Matrix(DoubleFloat),Matrix(Integer),Integer) -> Result
(--R f04mbf : (Integer,Matrix(DoubleFloat),Boolean,DoubleFloat,Integer, Integer,Integer,Integer,Integer,DoubleFloat,Integer,Union(fn: FileName,fp: Asp28(APROD)),Union(fn: FileName,fp: Asp34(MSOLVE))) -> Result
(--R f04mcf : (Integer,Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Matrix(Integer),Integer,Matrix(DoubleFloat),Integer,Integer,Integer,Integer) -> Result
(--R f04qaf : (Integer,Integer,DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp30(APROD))) -> Result
(--R)
(--E 165)
This package uses the NAG Library to solve the matrix equation
\[ AX = B, \]
where \( B \) may be a single vector or a matrix of multiple right-hand sides.

The matrix \( A \) may be real, complex, symmetric, Hermitian positive-definite, or sparse. It may also be rectangular, in which case a least-squares solution is obtained.

1. Scope of the Chapter

This chapter, together with two routines in Chapter F07, is concerned with the solution of the matrix equation \( AX = B \), where \( B \) may be a single vector or a matrix of multiple right-hand sides. The matrix \( A \) may be real, complex, symmetric, Hermitian positive-definite, or sparse. It may also be rectangular, in which case a least-squares solution is obtained.

2. Background to the Problems

A set of linear equations may be written in the form

\[ Ax = b \]

where the known matrix \( A \), with real or complex coefficients, is of size \( m \) by \( n \), (\( m \) rows and \( n \) columns), the known right-hand vector \( b \) has \( m \) components (\( m \) rows and one column), and the required solution vector \( x \) has \( n \) components (\( n \) rows and one column). There may sometimes be \( p \) vectors \( b_i \), \( i=1,2,...,p \) on the right-hand side and the equations may then be written as

\[ AX = B \]

the required matrix \( X \) having as its \( p \) columns the solutions of \( Ax = b_i \), \( i=1,2,...,p \). Some routines deal with the latter case, but for clarity only the case \( p=1 \) is discussed here.
The most common problem, the determination of the unique solution of \( Ax = b \), occurs when \( m = n \) and \( A \) is non-singular, that is \( \text{rank}(A) = n \) problem, discussed in Section 2.2 below, is the determination of the least-squares solution of \( Ax^\approx = b \), i.e., the determination of a vector \( x \) which minimizes the Euclidean length (two norm) of the residual vector \( r = b - Ax \). The usual case has \( m > n \) and \( \text{rank}(A) = n \), in which case \( x \) is unique.

### 2.1. Unique Solution of \( Ax = b \)

Most of the routines in this chapter, as well as two routines in Chapter F07, solve this particular problem. The solution is obtained by performing either an LU factorization, or a Cholesky factorization, as discussed in Section 2 of the F01 Chapter Introduction.

Two of the routines in this chapter use a process called iterative refinement to improve the initial solution in order to obtain a solution that is correct to working accuracy. It should be emphasised that if \( A \) and \( b \) are not known exactly then not all the figures in this solution may be meaningful. To be more precise, if \( x \) is the exact solution of the equations

\[
    Ax = b
\]

and \( x \) is the solution of the perturbed equations

\[
    (A+E)x = b + e,
\]

then, provided that \( (\kappa)(A) \leq 1 \),

\[
    \frac{||x-x||}{||x||} \leq \frac{(\kappa)(A)}{1 - (\kappa)(A)} \cdot \frac{||E||}{||A||} \cdot \frac{||e||}{||b||}
\]

where \( (\kappa)(A) = ||A|| ||A||^{-1} \) is the condition number of \( A \) with respect to inversion. Thus, if \( A \) is ill-conditioned (
(\kappa(A) \text{ is large}), x \text{ may differ significantly from } x. \text{ Often}
\|E\| \ll \kappa(A) \text{ in which case the above bound effectively}
\|A\|
\text{ reduces to}
\|x-x\|
\|x\| \ll \kappa(A) \left( \|E\| \|e\| \right)
\|A\| \|b\|

2.2. The Least-squares Solution of Ax=b

The least-squares problem is to find a vector x to minimize
\[ T \]
\[ r \]
where \( r=b-Ax \).

When m\geq n and \text{rank}(A)=n then the solution vector x is unique. For
the cases where x is not unique the routines in this chapter
obtain the minimal length solution, that is the vector x for
\[ T \]
which \( x^*x \) is a minimum.

2.3. Calculating the Inverse of a Matrix

The routines in this chapter can also be used to calculate the
inverse of a square matrix A by solving the equation
\[ AX=I, \]
where I is the identity matrix.

3. Recommendations on Choice and Use of Routines

3.1. General Purpose Routines

Many of the routines in this chapter perform the complete
solution of the required equations, but some of the routines, as
well as the routines in Chapter F07, assume that a prior
factorization has been performed, using the appropriate
factorization routine from Chapter F01 or Chapter F07. These, so-
called, general purpose routines can be useful when explicit
information on the factorization is required, as well as the
solution of the equations, or when the solution is required for
multiple right-hand sides, or for a sequence of right-hand sides.

Note that some of the routines that perform a complete solution also allow multiple right-hand sides.

3.2. Iterative Refinement

The routines that perform iterative refinement are more costly than those that do not perform iterative refinement, both in terms of time and storage, and should only be used if the problem really warrants the additional accuracy provided by these routines. The storage requirements are approximately doubled, while the additional time is not usually prohibitive since the initial factorization is used at each iteration.

3.3. Sparse Matrix Routines

The routines for sparse matrices should usually be used only when the number of non-zero elements is very small, less than 10% of the total number of elements of A. Additionally, when the matrix is symmetric positive-definite the sparse routines should generally be used only when A does not have a (variable) band structure.

There are four routines for solving sparse linear equations, two for solving general real systems (F04AXF and F04QAF), one for solving symmetric positive-definite systems (F04MAF) and one for solving symmetric systems that may, or may not, be positive-definite (F04MBF). F04AXF and F04MAF utilise factorizations of the matrix A obtained by routines in Chapter F01, while the other two routines use iterative techniques and require a user-supplied function to compute matrix-vector products Ac and A c for any given vector c. The routines requiring factorizations will usually be faster and the factorization can be utilised to solve for several right-hand sides, but the original matrix has to be explicitly supplied and is overwritten by the factorization, and the storage requirements will usually be substantially more than those of the iterative routines.

Routines F04MBF and F04QAF both allow the user to supply a preconditioner.

F04MBF can be used to solve systems of the form (A-(lambda)I)x=b, which can be useful in applications such as Rayleigh quotient iteration.

F04QAF also solves sparse least-squares problems and allows the solution of damped (regularized) least-squares problems.
3.4. Decision Trees

If at any stage the answer to a question is 'Don't know' this should be read as 'No'.

For those routines that need to be preceded by a factorization routine, the appropriate routine name is given in brackets after the name of the routine for solving the equations. Note also that you may be directed to a routine in Chapter F07.

3.4.1. Routines for unique solution of $Ax=b$

Please see figure in printed Reference Manual

3.4.2. Routines for Least-squares problems

Please see figure in printed Reference Manual

F04 -- Simultaneous Linear Equations

Chapter F04

Eigenvalues and Eigenvectors

F04ADF Approximate solution of complex simultaneous linear equations with multiple right-hand sides

F04ARF Approximate solution of real simultaneous linear equations, one right-hand side

F04ASF Accurate solution of real symmetric positive-definite simultaneous linear equations, one right-hand side

F04ATF Accurate solution of real simultaneous linear equations, one right-hand side

F04AXF Approximate solution of real sparse simultaneous linear equations (coefficient matrix already factorized by F01BRF or F01BSF)

F04FAF Approximate solution of real symmetric positive-definite tridiagonal simultaneous linear equations, one right-hand side
F04JGF  Least-squares (if rank = n) or minimal least-squares (if rank < n) solution of m real equations in n unknowns, rank <= n, m>=n

F04MAF  Real sparse symmetric positive-definite simultaneous linear equations (coefficient matrix already factorized)

F04MBF  Real sparse symmetric simultaneous linear equations

F04MCF  Approximate solution of real symmetric positive-definite variable-bandwidth simultaneous linear equations (coefficient matrix already factorized)

F04QAF  Sparse linear least-squares problem, m real equations in n unknowns

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F04 -- Simultaneous Linear Equations  F04ADF
F04ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04ADF calculates the approximate solution of a set of complex linear equations with multiple right-hand sides, using an LU factorization with partial pivoting.

2. Specification

SUBROUTINE F04ADF (A, IA, B, IB, N, M, C, IC, WKSPCE, 1
INTEGER IA, IB, N, M, IC, IFAIL
DOUBLE PRECISION WKSPCE(*)
COMPLEX(KIND(1.0D0)) A(IA,*), B(IB,*), C(IC,*)

3. Description

Given a set of complex linear equations AX=B, the routine first computes an LU factorization of A with partial pivoting, PA=LU, where P is a permutation matrix, L is lower triangular and U is unit upper triangular. The columns x of the solution X are found by forward and backward substitution in Ly=Pb and Ux=y, where b is a column of the right-hand side matrix B.
4. References


5. Parameters

1: \( A(I_A,*) \) -- COMPLEX(KIND(1.0D)) array Input/Output
   \hspace{1cm} Note: the second dimension of the array \( A \) must be at least \( \max(1,N) \).
   \hspace{1cm} On entry: the \( n \) by \( n \) matrix \( A \). On exit: \( A \) is overwritten by
   \hspace{1cm} the lower triangular matrix \( L \) and the off-diagonal elements
   \hspace{1cm} of the upper triangular matrix \( U \). The unit diagonal elements
   \hspace{1cm} of \( U \) are not stored.

2: \( I_A \) -- INTEGER Input
   \hspace{1cm} On entry:
   \hspace{1cm} the first dimension of the array \( A \) as declared in the
   \hspace{1cm} (sub)program from which F04ADF is called.
   \hspace{1cm} Constraint: \( I_A \geq \max(1,N) \).

3: \( B(I_B,*) \) -- COMPLEX(KIND(1.0D)) array Input
   \hspace{1cm} Note: the second dimension of the array \( B \) must be at least
   \hspace{1cm} \( \max(1,M) \).
   \hspace{1cm} On entry: the \( n \) by \( m \) right-hand side matrix \( B \). See also
   \hspace{1cm} Section 8.

4: \( I_B \) -- INTEGER Input
   \hspace{1cm} On entry:
   \hspace{1cm} the first dimension of the array \( B \) as declared in the
   \hspace{1cm} (sub)program from which F04ADF is called.
   \hspace{1cm} Constraint: \( I_B \geq \max(1,N) \).

5: \( N \) -- INTEGER Input
   \hspace{1cm} On entry: \( n \), the order of the matrix \( A \). Constraint: \( N \geq 0 \).

6: \( M \) -- INTEGER Input
   \hspace{1cm} On entry: \( m \), the number of right-hand sides. Constraint: \( M \geq 0 \).

7: \( C(I_C,*) \) -- COMPLEX(KIND(1.0D)) array Output
   \hspace{1cm} Note: the second dimension of the array \( C \) must be at least
   \hspace{1cm} \( \max(1,M) \).
   \hspace{1cm} On exit: the \( n \) by \( m \) solution matrix \( X \). See also Section 8.

8: \( I_C \) -- INTEGER Input
   \hspace{1cm} On entry:
   \hspace{1cm} the first dimension of the array \( C \) as declared in the
   \hspace{1cm} (sub)program from which F04ADF is called.
   \hspace{1cm} Constraint: \( I_C \geq \max(1,N) \).
9:  WKSPCE(*) -- DOUBLE PRECISION Workspace
    Note: the dimension of the array WKSPCE must be at least
    max(1, N).

10: IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
        familiar with this parameter (described in the Essential
        Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
        Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
    The matrix A is singular, possibly due to rounding errors.

IFAIL = 2
    On entry N < 0,
    or    M < 0,
    or    IA < max(1, N),
    or    IB < max(1, N),
    or    IC < max(1, N).

7. Accuracy

The accuracy of the computed solution depends on the conditioning
of the original matrix. For a detailed error analysis see

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your
implementation, the routine may be called with the same actual
array supplied for parameters B and C, in which case the solution
vectors will overwrite the right-hand sides. However this is not
standard Fortran 77, and may not work on all systems.
9. Example

To solve the set of linear equations \( AX = B \) where

\[
A = \begin{pmatrix}
1 & 1+2i & 2+10i \\
(1+bi) & 3i & -5+14i \\
(1+i) & 5i & -8+20i
\end{pmatrix}
\]

and

\[
B = \begin{pmatrix}
1 \\
0 \\
0
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
4. References


5. Parameters

1: A(IA,*) -- DOUBLE PRECISION array Input/Output
   Note: the second dimension of the array A must be at least max(1,N).
   On entry: the n by n matrix A. On exit: A is overwritten by
   the lower triangular matrix L and the off-diagonal elements
   of the upper triangular matrix U. The unit diagonal elements
   of U are not stored.

2: IA -- INTEGER Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which F04ARF is called.
   Constraint: IA >= max(1,N).

3: B(*) -- DOUBLE PRECISION array Input
   Note: the dimension of the array B must be at least
   max(1,N).
   On entry: the right-hand side vector b.

4: N -- INTEGER Input
   On entry: n, the order of the matrix A. Constraint: N >= 0.

5: C(*) -- DOUBLE PRECISION array Output
   Note: the dimension of the array C must be at least
   max(1,N).
   On exit: the solution vector x.

6: WKSPCE(*) -- DOUBLE PRECISION array Workspace
   Note: the dimension of the array WKSPCE must be at least
   max(1,N).

7: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
The matrix A is singular, possibly due to rounding errors.

IFAIL = 2
On entry N < 0,
or IA < max(1,N).

7. Accuracy

The accuracy of the computed solution depends on the conditioning of the original matrix. For a detailed error analysis see Wilkinson and Reinsch [1] page 107.

8. Further Comments

The time taken by the routine is approximately proportional to n

Unless otherwise stated in the Users' Note for your implementation, the routine may be called with the same actual array supplied for parameters B and C, in which case the solution vector will overwrite the right-hand side. However this is not standard Fortran 77, and may not work on all systems.

9. Example

To solve the set of linear equations Ax=b where

\[
\begin{pmatrix}
33 & 16 & 72 \\
-24 & -10 & -57 \\
-8 & -4 & -17
\end{pmatrix}
\]

\[
A=\begin{pmatrix}
-359 \\
281 \\
85
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

F04ASF calculates the accurate solution of a set of real symmetric positive-definite linear equations with a single right-hand side, \( Ax = b \), using a Cholesky factorization and iterative refinement.

2. Specification

```fortran
SUBROUTINE F04ASF (A, IA, B, N, C, WK1, WK2, IFAIL)
INTEGER IA, N, IFAIL
DOUBLE PRECISION A(IA,*), B(*), C(*), WK1(*), WK2(*)
```

3. Description

Given a set of real linear equations \( Ax = b \), where \( A \) is a symmetric positive-definite matrix, the routine first computes a Cholesky factorization of \( A \) as \( A = LL^T \) where \( L \) is lower triangular. An approximation to \( x \) is found by forward and backward substitution. The residual vector \( r = b - Ax \) is then calculated using additional precision and a correction \( d \) to \( x \) is found by solving \( LL^T d = r \). \( x \) is then replaced by \( x + d \), and this iterative refinement of the solution is repeated until machine accuracy is obtained.

4. References


5. Parameters

1: \( A(IA,*) \) -- DOUBLE PRECISION array Input/Output

Note: the second dimension of the array \( A \) must be at least \( \max(1,N) \).

On entry: the upper triangle of the \( n \) by \( n \) positive-definite symmetric matrix \( A \). The elements of the array below the diagonal need not be set. On exit: the elements of the array below the diagonal are overwritten; the upper triangle of \( A \) is unchanged.
2: IA -- INTEGER
On entry: 
the first dimension of the array A as declared in the 
(sub)program from which F04ASF is called. 
Constraint: IA >= max(1,N).

3: B(*) -- DOUBLE PRECISION array
Note: the dimension of the array B must be at least 
max(1,N).
On entry: the right-hand side vector b.

4: N -- INTEGER
On entry: n, the order of the matrix A. Constraint: N >= 0.

5: C(*) -- DOUBLE PRECISION array
Note: the dimension of the array C must be at least 
max(1,N).
On exit: the solution vector x.

6: WK1(*) -- DOUBLE PRECISION array
Note: the dimension of the array WK1 must be at least 
max(1,N).

7: WK2(*) -- DOUBLE PRECISION array
Note: the dimension of the array WK2 must be at least 
max(1,N).

8: IFAIL -- INTEGER
On entry: IFAIL must be set to 0, -1 or 1. For users not 
familiar with this parameter (described in the Essential 
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see 
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are 
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
The matrix A is not positive-definite, possibly due to 
rounding errors.

IFAIL = 2
Iterative refinement fails to improve the solution, i.e., 
the matrix A is too ill-conditioned.
IFAIL= 3
   On entry N < 0,
   or IA < max(1, N).

7. Accuracy

The computed solutions should be correct to full machine accuracy. For a detailed error analysis see Wilkinson and Reinsch [1] page 39.

8. Further Comments

3

The time taken by the routine is approximately proportional to n

The routine must not be called with the same name for parameters B and C.

9. Example

To solve the set of linear equations Ax=b where

\[
\begin{pmatrix}
5 & 7 & 6 & 5 \\
7 & 10 & 8 & 7 \\
6 & 8 & 10 & 9 \\
5 & 7 & 9 & 10
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
23 \\
32 \\
31
\end{pmatrix}
\]

b=(33).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F04 -- Simultaneous Linear Equations  F04ATF
   F04ATF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

F04ATF calculates the accurate solution of a set of real linear equations with a single right-hand side, using an LU factorization with partial pivoting, and iterative refinement.

2. Specification

```plaintext
SUBROUTINE F04ATF (A, IA, B, N, C, AA, IAA, WKS1, WKS2,
  INTEGER IA, N, IAA, IFAIL
  DOUBLE PRECISION A(IA,*), B(*), C(*), AA(IAA,*), WKS1(*),
  WKS2(*)
```

3. Description

Given a set of real linear equations, \( Ax=b \), the routine first computes an LU factorization of \( A \) with partial pivoting, \( PA=LU \), where \( P \) is a permutation matrix, \( L \) is lower triangular and \( U \) is unit upper triangular. An approximation to \( x \) is found by forward and backward substitution in \( Ly=Pb \) and \( Ux=y \). The residual vector \( r=b-Ax \) is then calculated using additional precision, and a correction \( d \) to \( x \) is found by solving \( LUd=r \). \( x \) is replaced by \( x+d \), and this iterative refinement of the solution is repeated until full machine accuracy is obtained.

4. References


5. Parameters

1: \( A(IA,*) \) -- DOUBLE PRECISION array

   Note: the second dimension of the array \( A \) must be at least \( \max(1,N) \).

   On entry: the \( n \) by \( n \) matrix \( A \).

2: \( IA \) -- INTEGER

   On entry:
   the first dimension of the array \( A \) as declared in the (sub)program from which F04ATF is called.

   Constraint: \( IA \geq \max(1,N) \).

3: \( B(*) \) -- DOUBLE PRECISION array

   Note: the dimension of the array \( B \) must be at least \( \max(1,N) \).

   On entry: the right-hand side vector \( b \).

4: \( N \) -- INTEGER

   Input
On entry: n, the order of the matrix A. Constraint: N >= 0.

5: C(*) -- DOUBLE PRECISION array Output
Note: the dimension of the array C must be at least max(1,N).
On exit: the solution vector x.

6: AA(IAA,*) -- DOUBLE PRECISION array Output
Note: the second dimension of the array AA must be at least max(1,N).
On exit: the triangular factors L and U, except that the unit diagonal elements of U are not stored.

7: IAA -- INTEGER Input
On entry:
the first dimension of the array AA as declared in the (sub)program from which F04ATF is called.
Constraint: IAA >= max(1,N).

8: WKS1(*) -- DOUBLE PRECISION array Workspace
Note: the dimension of the array WKS1 must be at least max(1,N).

9: WKS2(*) -- DOUBLE PRECISION array Workspace
Note: the dimension of the array WKS2 must be at least max(1,N).

10: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The matrix A is singular, possibly due to rounding errors.

IFAIL= 2
Iterative refinement fails to improve the solution, i.e., the matrix A is too ill-conditioned.
IFAIL= 3
    On entry N < 0,
    or IA < max(1,N),
    or IAA < max(1,N).

7. Accuracy

The computed solutions should be correct to full machine accuracy. For a detailed error analysis see Wilkinson and Reinsch [1] page 107.

8. Further Comments

The time taken by the routine is approximately proportional to n

The routine must not be called with the same name for parameters B and C.

9. Example

To solve the set of linear equations Ax=b where

\[
\begin{pmatrix}
33 & 16 & 72 \\
-24 & -10 & -57 \\
-8 & -4 & -17
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
-359 \\
b = (281) \\
85
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F04 -- Simultaneous Linear Equations  F04AXF
F04AXF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

F04AXF calculates the approximate solution of a set of real sparse linear equations with a single right-hand side, \( Ax=b \) or \( A^T x=b \), where \( A \) has been factorized by F01BRF or F01BSF.

2. Specification

```fortran
SUBROUTINE F04AXF (N, A, LICN, ICN, IKEEP, RHS, W, MTYPE, 
                   IDISP, RESID)
  INTEGER N, LICN, ICN(LICN), IKEEP(5*N), MTYPE, 
          IDISP(2)
  DOUBLE PRECISION A(LICN), RHS(N), W(N), RESID
```

3. Description

To solve a system of real linear equations \( Ax=b \) or \( A^T x=b \), where \( A \) is a general sparse matrix, \( A \) must first be factorized by F01BRF or F01BSF. F04AXF then computes \( x \) by block forward or backward substitution using simple forward and backward substitution within each diagonal block.

The method is fully described in Duff [1].

4. References


5. Parameters

1: N -- INTEGER  
   Input
   On entry: \( n \), the order of the matrix \( A \).

2: A(LICN) -- DOUBLE PRECISION array  
   Input
   On entry: the non-zero elements in the factorization of the matrix \( A \), as returned by F01BRF or F01BSF.

3: LICN -- INTEGER  
   Input
   On entry: the dimension of the arrays \( A \) and \( ICN \) as declared in the (sub)program from which F04AXF is called.

4: ICN(LICN) -- INTEGER array  
   Input
   On entry: the column indices of the non-zero elements of the factorization, as returned by F01BRF or F01BSF.
5: IKEEP(5*N) -- INTEGER array
   On entry: the indexing information about the factorization,
   as returned by F01BRF or F01BSF.

6: RHS(N) -- DOUBLE PRECISION array
   On entry: the right-hand side vector b. On exit: RHS is
   overwritten by the solution vector x.

7: W(N) -- DOUBLE PRECISION array
   Workspace

8: MTYPE -- INTEGER
   On entry: MTYPE specifies the task to be performed:
   if MTYPE = 1, solve Ax=b,
   if MTYPE /= 1, solve A x=b.

9: IDISP(2) -- INTEGER array
   On entry: the values returned in IDISP by F01BRF.

10: RESID -- DOUBLE PRECISION
    On exit: the value of the maximum residual,
    --
    max(|b - a x|), over all the unsatisfied equations, in
    i -- ij j
    case F01BRF or F01BSF has been used to factorize a singular
    or rectangular matrix.

6. Error Indicators and Warnings

None.

7. Accuracy

The accuracy of the computed solution depends on the conditioning
of the original matrix. Since F04AXF is always used with either
F01BRF or F01BSF, the user is recommended to set GROW = .TRUE. on
entry to these routines and to examine the value of W(1) on exit
(see the routine documents for F01BRF and F01BSF). For a detailed
error analysis see Duff [1] page 17.

If storage for the original matrix is available then the error
can be estimated by calculating the residual

\[ T \]

\[ r = b - Ax \] (or \[ b - A x \])

and calling F04AXF again to find a correction (delta) for x by
solving
T
A(delta)=r (or A(delta)=r).

8. Further Comments

If the factorized form contains (tau) non-zeros (IDISP(2) = (tau) ) then the time taken is very approximately 2(tau) times longer than the inner loop of full matrix code. Some advantage is taken of zeros in the right-hand side when solving A \( \times b \) (MTYPE /= 1).

9. Example

To solve the set of linear equations \( A \times b \) where

\[
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & -1 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 & 0 & 0 \\
\end{pmatrix}
\begin{pmatrix}
-2 & 0 & 0 & 1 & 1 & 0 \\
-1 & 0 & 0 & -1 & 2 & -3 \\
-1 & -1 & 0 & 0 & 0 & 6 \\
\end{pmatrix}
\]

and

\[
\begin{pmatrix}
15 \\
12 \\
18 \\
\end{pmatrix}
\]

\[
\begin{pmatrix}
3 \\
-6 \\
0 \\
\end{pmatrix}
\]

The non-zero elements of A and indexing information are read in by the program, as described in the document for F01BRF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

F04FAF calculates the approximate solution of a set of real symmetric positive-definite tridiagonal linear equations.

2. Specification

```fortran
SUBROUTINE F04FAF (JOB, N, D, E, B, IFAIL)
    INTEGER JOB, N, IFAIL
    DOUBLE PRECISION D(N), E(N), B(N)
```

3. Description

F04FAF is based upon the Linpack routine DPTSL (see Dongarra et al [1]) and solves the equations

\[ Tx = b, \]

where \( T \) is a real \( n \) by \( n \) symmetric positive-definite tridiagonal matrix, using a modified symmetric Gaussian elimination algorithm to factorize \( T \) as \( T = MKM \), where \( K \) is diagonal and \( M \) is a matrix of multipliers as described in Section 8.

When the input parameter JOB is supplied as 1, then the routine assumes that a previous call to F04FAF has already factorized \( T \); otherwise JOB must be supplied as 0.

4. References


5. Parameters

1: JOB -- INTEGER

Input

On entry: specifies the job to be performed by F04FAF as follows:

\( \text{JOB} = 0 \)

The matrix \( T \) is factorized and the equations \( Tx = b \) are solved for \( x \).

\( \text{JOB} = 1 \)

The matrix \( T \) is assumed to have already been factorized by a previous call to F04FAF with \( \text{JOB} = 0 \); the equations \( Tx = b \) are solved for \( x \).

2: N -- INTEGER

Input

On entry: \( n \), the order of the matrix \( T \). Constraint: \( N \geq 1 \).
3: D(N) -- DOUBLE PRECISION array  Input/Output
   On entry: if JOB = 0, D must contain the diagonal elements
   of T. If JOB = 1, D must contain the diagonal matrix K, as
   returned by a previous call of F04FAF with JOB = 0. On
   exit: if JOB = 0, D is overwritten by the diagonal matrix K
   of the factorization. If JOB = 1, D is unchanged.

4: E(N) -- DOUBLE PRECISION array  Input/Output
   On entry: if JOB = 0, E must contain the super-diagonal
   elements of T, stored in E(2) to E(n). If JOB = 1, E must
   contain the off-diagonal elements of the matrix M, as
   returned by a previous call of F04FAF with JOB = 0. E(1) is
   not used. On exit: if JOB = 0, E(2) to E(n) are overwritten
   by the off-diagonal elements of the matrix M of the
   factorization. If JOB = 1, E is unchanged.

5: B(N) -- DOUBLE PRECISION array  Input/Output
   On entry: the right-hand side vector b. On exit: B is
   overwritten by the solution vector x.

6: IFAIL -- INTEGER  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry N < 1,
   or     JOB /= 0 or 1.

IFAIL= 2
   The matrix T is either not positive-definite or is nearly
   singular. This failure can only occur when JOB = 0 and
   inspection of the elements of D will give an indication of
   why failure has occurred. If an element of D is close to
   zero, then T is probably nearly singular; if an element of D
   is negative but not close to zero, then T is not positive-
   definite.

IFAIL=Overflow
   If overflow occurs during the execution of this routine,
   then either T is very nearly singular or an element of the
   right-hand side vector b is very large. In this latter case
the equations should be scaled so that no element of \( b \) is very large. Note that to preserve symmetry it is necessary to scale by a transformation of the form \((\mathbf{PTP})\mathbf{b} = \mathbf{Px} \), where \( \mathbf{P} \) is a diagonal matrix.

IFAILUnderflow

Any underflows that occur during the execution of this routine are harmless.

7. Accuracy

The computed factorization (see Section 8) will satisfy the equation

\[
\mathbf{T} = \mathbf{MKM} = \mathbf{T} + \mathbf{E}
\]

where \( \|\mathbf{E}\| \leq 2(\varepsilon)\|\mathbf{T}\| \), \( p = 1, F, \infty \),

\[
\mathbf{P} = \mathbf{P}
\]

(\( \varepsilon \)) being the machine precision. The computed solution of the equations \( \mathbf{T}\mathbf{x} = \mathbf{b} \), say \( \mathbf{x} \), will satisfy an equation of the form

\[
(\mathbf{T} + \mathbf{F})\mathbf{x} = \mathbf{b},
\]

where \( \mathbf{F} \) can be expected to satisfy a bound of the form

\[
\|\mathbf{F}\| \leq (\alpha)(\varepsilon)\|\mathbf{T}\|,
\]

(\( \alpha \)) being a modest constant. This implies that the relative error in \( \mathbf{x} \) satisfies

\[
\|\mathbf{x} - \mathbf{x}\| \\
\|\mathbf{x}\| \leq c(\mathbf{T})\alpha\varepsilon,
\]

where \( c(\mathbf{T}) \) is the condition number of \( \mathbf{T} \) with respect to inversion. Thus if \( \mathbf{T} \) is nearly singular, \( \mathbf{x} \) can be expected to have a large relative error.
8. Further Comments

The time taken by the routine is approximately proportional to n.

The routine eliminates the off-diagonal elements of T by simultaneously performing symmetric Gaussian elimination from the top and the bottom of T. The result is that T is factorized as

$$T = MKM,$$

where K is a diagonal matrix and M is a matrix of the form

$$M = \begin{pmatrix}
1 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\
0 & m & 1 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 0 \\
(2) \\
0 & 0 & 0 & \ldots & 1 & 0 & 0 & \ldots & 0 & 0 & 0 \\
(3) \\
& \ddots & \ddots & \ddots \\
0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 1 & 0 & 0 \\
( j+1 ) \\
0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 1 & m \\
( j+2 ) \\
0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \\
( n-1 ) \\
0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 1 \\
( n ) \\
0 & 0 & 0 & \ldots & 0 & 0 & 0 & \ldots & 0 & 0 & 1
\end{pmatrix}$$

j being the integer part of n/2. (For example when n=5, j=2.) The diagonal elements of K are returned in D with k in the ith element of D and m is returned in the ith element of E.

The routine fails with IFAIL = 2 if any diagonal element of K is non-positive. It should be noted that T may be nearly singular even if all the diagonal elements of K are positive, but in this case at least one element of K is almost certain to be small relative to ||||T|||. If there is any doubt as to whether or not T is nearly singular, then the user should consider examining the diagonal elements of K.

9. Example

To solve the symmetric positive-definite equations
The equations are solved by two calls to F04FAF, the first with 
JOB = 0 and the second, using the factorization from the first call, with JOB = 1.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

F04 -- Simultaneous Linear Equations

F04JGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04JGF finds the solution of a linear least-squares problem, $A x = b$, where $A$ is a real $m$ by $n$ ($m \geq n$) matrix and $b$ is an $m$ element vector. If the matrix of observations is not of full rank, then the minimal least-squares solution is returned.

2. Specification

```fortran
SUBROUTINE F04JGF (M, N, A, NRA, B, TOL, SVD, IRANK, WORK, LWORK, IFAIL)
    INTEGER M, N, NRA, IRANK, LWORK, IFAIL
    DOUBLE PRECISION A(NRA,N), B(M), TOL, SIGMA, WORK(LWORK)
    LOGICAL SVD
```

---
3. Description

The minimal least-squares solution of the problem $Ax = b$ is the vector $x$ of minimum (Euclidean) length which minimizes the length of the residual vector $r = b - Ax$.

The real $m$ by $n$ ($m \geq n$) matrix $A$ is factorized as

$$A = QU$$

where $Q$ is an $m$ by $m$ orthogonal matrix and $U$ is an $n$ by $n$ upper triangular matrix. If $U$ is of full rank, then the least-squares solution is given by

$$x = (U^{-1}T)Qb.$$  

If $U$ is not of full rank, then the singular value decomposition of $U$ is obtained so that $U$ is factorized as

$$U = RD^TP,$$

where $R$ and $P$ are $n$ by $n$ orthogonal matrices and $D$ is the $n$ by $n$ diagonal matrix

$$D = \text{diag}((\sigma_1), (\sigma_2), \ldots, (\sigma_n)),$$

with $(\sigma_1) \geq (\sigma_2) \geq \ldots \geq (\sigma_n) \geq 0$, these being the singular values of $A$. If the singular values $(\sigma_1), \ldots, (\sigma_n)$ are negligible, but $(\sigma_{k+1})$ is not negligible, relative to the data errors in $A$, then the rank of $A$ is taken to be $k$ and the minimal least-squares solution is given by

$$x = (S^{-1}T)(R^{-1}P)b,$$

where $S = \text{diag}((\sigma_1), (\sigma_2), \ldots, (\sigma_k))$.

This routine obtains the factorizations by a call to F02WDF(*).

The routine also returns the value of the standard error.
(sigma) = \begin{cases} 
  \frac{T}{m-k}, & \text{if } m > k, \\
  0, & \text{if } m = k, 
\end{cases}

if \( m \) is the residual sum of squares and \( k \) the rank of \( A \).

4. References


5. Parameters

1: \( M \) -- INTEGER Input
   On entry: \( m \), the number of rows of \( A \). Constraint: \( M \geq N \).

2: \( N \) -- INTEGER Input
   On entry: \( n \), the number of columns of \( A \). Constraint: \( 1 \leq N \leq M \).

3: \( A(\text{NRA},N) \) -- DOUBLE PRECISION array Input/Output
   On entry: the \( m \) by \( n \) matrix \( A \). On exit: if \( \text{SVD} \) is returned as \( \text{.FALSE.} \), \( A \) is overwritten by details of the QU factorization of \( A \) (see F02WDF(*) for further details). If SVD is returned as \( \text{.TRUE.} \), the first \( n \) rows of \( A \) are overwritten by the right-hand singular vectors, stored by rows; and the remaining rows of the array are used as workspace.

4: \( \text{NRA} \) -- INTEGER Input
   On entry: the first dimension of the array \( A \) as declared in the (sub)program from which F04JGF is called.
   Constraint: \( \text{NRA} \geq M \).

5: \( B(M) \) -- DOUBLE PRECISION array Input/Output
   On entry: the right-hand side vector \( b \). On exit: the first \( n \) elements of \( B \) contain the minimal least-squares solution vector \( x \). The remaining \( m-n \) elements are used for workspace.

6: \( \text{TOL} \) -- DOUBLE PRECISION Input
   On entry: a relative tolerance to be used to determine the rank of \( A \). \( \text{TOL} \) should be chosen as approximately the largest relative error in the elements of \( A \). For example, if the elements of \( A \) are correct to about 4 significant figures
then TOL should be set to about $5 \times 10^{-11}$. See Section 8 for a description of how TOL is used to determine rank. If TOL is outside the range $(\epsilon, 1.0)$, where $\epsilon$ is the machine precision, then the value $\epsilon$ is used in place of TOL. For most problems this is unreasonably small.

7: SVD -- LOGICAL Output
On exit: SVD is returned as .FALSE. if the least-squares solution has been obtained from the QU factorization of A. In this case A is of full rank. SVD is returned as .TRUE. if the least-squares solution has been obtained from the singular value decomposition of A.

8: SIGMA -- DOUBLE PRECISION Output
\[
\sigma / T
\]
On exit: the standard error, i.e., the value $\sqrt{\|r\|^2/(m-k)}$ when $m > k$, and the value zero when $m = k$. Here $r$ is the residual vector $b - Ax$ and $k$ is the rank of $A$.

9: IRANK -- INTEGER Output
On exit: $k$, the rank of the matrix $A$. It should be noted that it is possible for IRANK to be returned as $n$ and SVD to be returned as .TRUE.. This means that the matrix $U$ only just failed the test for non-singularity.

10: WORK(LWORK) -- DOUBLE PRECISION array Output
On exit: if SVD is returned as .FALSE., then the first $n$ elements of WORK contain information on the QU factorization of $A$ (see parameter A above and F02WDF(*)), and WORK$(n+1)$ contains the condition number $\|U\| E / E$ of the upper triangular matrix $U$.
If SVD is returned as .TRUE., then the first $n$ elements of WORK contain the singular values of $A$ arranged in descending order and WORK$(n+1)$ contains the total number of iterations taken by the QR algorithm. The rest of WORK is used as workspace.

11: LWORK -- INTEGER Input
On entry: the dimension of the array WORK as declared in the (sub)program from which F04JGF is called.
Constraint: $LWORK \geq 4 \times N$.

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry N < 1,
or M < N,
or NRA < M,
or LWORK < 4*N.

IFAIL= 2
The QR algorithm has failed to converge to the singular values in 50*N iterations. This failure can only happen when the singular value decomposition is employed, but even then it is not likely to occur.

7. Accuracy

The computed factors Q, U, R, D and P satisfy the relations

\[
(U) \begin{pmatrix} R & 0 \\ 0 & D \end{pmatrix}^T Q(0) = A+E, \quad Q(0) P(0) = A+F,
\]

where

\[
\| E \| \leq c(\text{epsilon}) \| A \|,
\]

\[
\| F \| \leq c(\text{epsilon}) \| A \|,
\]

(epsilon) being the machine precision, and c and c being modest functions of m and n. Note that \| A \| = (\text{sigma}).

For a fuller discussion, covering the accuracy of the solution x see Lawson and Hanson [1], especially pp 50 and 95.
8. Further Comments

If the least-squares solution is obtained from the QU factorization then the time taken by the routine is approximately \( \frac{2}{2} \) proportional to \( n(3m-n) \). If the least-squares solution is obtained from the singular value decomposition then the time taken is approximately proportional to \( n(3m+19n) \). The approximate proportionality factor is the same in each case.

This routine is column biased and so is suitable for use in paged environments.

Following the QU factorization of \( A \) the condition number

\[
\begin{align*}
c(U) &= \frac{||U||}{||U||} \\
     &= \frac{E}{E}
\end{align*}
\]

is determined and if \( c(U) \) is such that

\[
c(U) > TOL \times c(U)\]

then \( U \) is regarded as singular and the singular values of \( A \) are computed. If this test is not satisfied, \( U \) is regarded as non-singular and the rank of \( A \) is set to \( n \). When the singular values are computed the rank of \( A \), say \( k \), is returned as the largest integer such that

\[
\sigma_i > TOL \times \sigma_i\]

unless \( \sigma_i = 0 \) in which case \( k \) is returned as zero. That is, \( \sigma_i \) singular values which satisfy \( \sigma_i \leq TOL \times \sigma_i \) are regarded as negligible because relative perturbations of order \( TOL \) can make such singular values zero.

9. Example

To obtain a least-squares solution for \( r = b - Ax \), where

\[
\begin{align*}
A &= \begin{pmatrix} 1.75 & 1.75 & 0.35 & -0.35 \end{pmatrix}, \quad B = \begin{pmatrix} 4 \end{pmatrix} \\
(0.05 & 0.05 & 0.25 & -0.25) & (0.25 & 0.25 & 0.05 & -0.05) & (0.35 & 0.35 & 1.75 & -1.75) \\
(0.30 & -0.30 & 0.30 & 0.30) & (0.40 & -0.40 & 0.40 & 0.40)
\end{align*}
\]
and the value TOL is to be taken as 5*10^{-4}.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

F04MAF solves the n linear equations

\[ Ax = b, \]

where \( A \) is a sparse symmetric positive-definite matrix, following the incomplete Cholesky factorization by F01MAF, given by

\[ T^T \]

\[ C = PLDL^T \quad \text{and} \quad WAW = C + E, \]

where \( P \) is a permutation matrix, \( L \) is a unit lower triangular matrix, \( D \) is a diagonal matrix with positive diagonal elements, \( E \) is an error matrix representing elements dropped during the factorization and diagonal elements that have been modified to...
ensure that $C$ is positive-definite, and $W$ is a diagonal matrix, chosen to make the diagonal elements of $WAW$ unity.

Equation (1) is solved by applying a pre-conditioned conjugate gradient method to the equations

$$
(WAW)(W x) = Wb,
$$

(2)

using $C$ as the pre-conditioning matrix. Details of the conjugate gradient method are given in Munksgaard [1].

The iterative procedure is terminated if

$$
||Wr|| \leq (\eta),
$$

(3)

where $r$ is the residual vector $r = b - Ax$, $||r||$ denotes the Euclidean length of $r$, $(\eta)$ is a user-supplied tolerance and $x$ is the current approximation to the solution. Notice that

$$
-Wr = Wb - (WAW)(W x)
$$

so that $Wr$ is the residual of the normalised equations (2).

F04MAF is based on the Harwell Library routine MA31B.

4. References


5. Parameters

1:  N -- INTEGER  Input
    On entry: n, the order of the matrix $A$. Constraint: $N \geq 1$.

2:  NZ -- INTEGER  Input
    On entry: the number of non-zero elements in the upper triangular part of the matrix $A$, including the number of elements on the leading diagonal. Constraint: $NZ \geq N$.

3:  A(LICN) -- DOUBLE PRECISION array  Input
    On entry: the first LROW elements, where LROW is the value supplied in INFORM(1), must contain details of the factorization, as returned by F01MAF.
4: LICN -- INTEGER
   On entry: the length of the array A, as declared in the
   (sub)program from which F04MAF is called. It need never be
   larger than the value of LICN supplied to F01MAF.
   Constraint: LICN >= INFORM(1).

5: IRN(LIRN) -- INTEGER array
   On entry: the first LCOL elements, where LCOL is the value
   supplied in INFORM(2), must contain details of the
   factorization, as returned by F01MAF.

6: LIRN -- INTEGER
   On entry: the length of the array IRN, as declared in the
   (sub)program from which F04MAF is called. It need never be
   larger than the value of LIRN supplied to F01MAF.
   Constraint: LIRN >= INFORM(2).

7: ICN(LICN) -- INTEGER array
   On entry: the first LROW elements, where LROW is the value
   supplied in INFORM(1), must contain details of the
   factorization, as returned by F01MAF.

8: B(N) -- DOUBLE PRECISION array
   On entry: the right-hand side vector b. On exit: B is
   overwritten by the solution vector x.

9: ACC(2) -- DOUBLE PRECISION array
   On entry: ACC(1) specifies the tolerance for convergence,
   (eta), in equation (3) of Section 3. If ACC(1) is outside
   the range [(epsilon),1], where (epsilon) is the machine
   precision, then the value (epsilon) is used in place of ACC
   (1). ACC(2) need not be set. On exit: ACC(2) contains the
   actual value of ||Wr|| at the final point. ACC(1) is
   unchanged.

10: NOITS(2) -- INTEGER array
    On entry: NOITS(1) specifies the maximum permitted number of
    iterations. If NOITS(1) < 1, then the value 100 is used in
    its place. NOITS(2) need not be set. On exit: NOITS(2)
    contains the number of iterations taken to converge. NOITS
    (1) is unchanged.

11: WKEEP(3*N) -- DOUBLE PRECISION array
    On entry: WKEEP must be unchanged from the previous call of
    F01MAF.

12: WORK(3*N) -- DOUBLE PRECISION array
    On exit: WORK(1) contains a lower bound for the condition
    number of A. The rest of the array is used for workspace.
13: IKEEP(2*N) -- INTEGER array          Input
    On entry: IKEEP must be unchanged from the previous call of
              F01MAF.

14: INFORM(4) -- INTEGER array          Input
    On entry: INFORM must be unchanged from the previous call of
              F01MAF.

15: IFAIL -- INTEGER          Input/Output
    For this routine, the normal use of IFAIL is extended to
    control the printing of error and warning messages as well
    as specifying hard or soft failure (see the Essential
    Introduction).

    Before entry, IFAIL must be set to a value with the decimal
    expansion cba, where each of the decimal digits c, b and a
    must have a value of 0 or 1.
    a=0 specifies hard failure, otherwise soft failure;
    b=0 suppresses error messages, otherwise error messages
    will be printed (see Section 6);
    c=0 suppresses warning messages, otherwise warning
    messages will be printed (see Section 6).

    The recommended value for inexperienced users is 110 (i.e.,
    hard failure with all messages printed).

    Unless the routine detects an error (see Section 6), IFAIL
    contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
    On entry N < 1,
    or    NZ < N,
    or    LICN < INFORM(1),
    or    LIRN < INFORM(2).

IFAIL= 2
    Convergence has not taken place within the requested NOITS
(1) number of iterations. ACC(2) gives the value $||Wr||_2$ for the final point. Either too few iterations have been allowed, or the requested convergence criterion cannot be met.

**IFAIL= 3**
The matrix A is singular, or nearly singular. Singularity has been detected during the conjugate gradient iterations, so that the computations are not complete.

**IFAIL= 4**
The matrix A is singular, or nearly singular. The message output on the current error message channel will include an estimate of the condition number of A. In the case of soft failure an approximate solution is returned such that the value $||Wr||_2$ is given by ACC(2) and the estimate (an lower bound) of the condition number is returned in WORK(1).

7. Accuracy
On successful return, or on return with IFAIL = 2 or IFAIL = 4 the computed solution will satisfy equation (3) of Section 3, with $(\eta) = ACC(2)$.

8. Further Comments
The time taken by the routine will depend upon the sparsity of the factorization and the number of iterations required. The number of iterations will be affected by the nature of the factorization supplied by F01MAF. The more incomplete the factorization, the higher the number of iterations required by F04MAF.

When the solution of several systems of equations, all with the same matrix of coefficients, A, is required, then F01MAF need be called only once to factorize A. This is illustrated in the context of an eigenvalue problem in the example program for F02FJF.

9. Example
The example program illustrates the use of F01MAF in conjunction with F04MAF to solve the 16 linear equations $Ax=b$, where

$\begin{pmatrix}
1 & a & & \\
(1) & a & 1 & a \\
(1) & & a & 1 \\
& & 0 & a
\end{pmatrix}$
The n by n matrix $A$ arises in the solution of Laplace's equation in a unit-square, using a five-point formula with a 6 by 6 discretisation, with unity on the boundaries.

The drop tolerance, DROPTL, is taken as 0.1 and the density factor, DENSW, is taken as 0.8. The value IFAIL = 111 is used so that advisory and error messages will be printed, but soft failure would occur if IFAIL were returned as non-zero.

A relative accuracy of about 0.0001 is requested in the solution from F04MAF, with a maximum of 50 iterations.

The example program for F02FJF illustrates the use of routines F01MAF and F04MAF in solving an eigenvalue problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

F04MBF solves a system of real sparse symmetric linear equations using a Lanczos algorithm.

2. Specification

```fortran
SUBROUTINE F04MBF (N, B, X, APROD, MSOLVE, PRECON, SHIFT,
  1 RTOL, ITNLIM, MSGVLVL, ITN, ANORM,
  2 ACOND, RNORM, XNORM, WORK, RWORK,
  3 LWORK, IWORK, LIWORK, INFORM, IFAIL)
INTEGER N, ITNLIM, MSGVLVL, ITN, LWORK, IWORK, LIWORK, INFORM, IFAIL
DOUBLE PRECISION B(N), X(N), SHIFT, RTOL, ANORM, ACOND,
  1 RNORM, XNORM, WORK(N,5), RWORK(LWORK)
LOGICAL PRECON
EXTERNAL APROD, MSOLVE
```

3. Description

F04MBF solves the system of linear equations

\[ (A-(\lambda)I)x = b \quad (3.1) \]

where \( A \) is an \( n \) by \( n \) sparse symmetric matrix and \( \lambda \) is a scalar, which is of course zero if the solution of the equations \( Ax = b \) is required. It should be noted that neither \( A \) nor \( (A-(\lambda)I) \) need be positive-definite.

\( \lambda \) is supplied as the parameter SHIFT, and allows F04MBF to be used for finding eigenvectors of \( A \) in methods such as Rayleigh quotient iteration (see for example Lewis \[1\]), in which case \( \lambda \) will be an approximation to an eigenvalue of \( A \) and \( b \) an approximation to an eigenvector of \( A \).

The routine also provides an option to allow pre-conditioning and this will often reduce the number of iterations required by F04MBF.

F04MBF is based upon algorithm SYMMLQ (see Paige and Saunders \[2\]) and solves the equations by an algorithm based upon the Lanczos process. Details of the method are given in Paige and Saunders \[2\]. The routine does not require \( A \) explicitly, but \( A \) is specified via a user-supplied routine APROD which, given an \( n \) element vector \( c \), must return the vector \( z \) given by
z = Ac.

The pre-conditioning option is based on the following reasoning. If $A$ can be expressed in the form

$$ A = I + B $$

where $B$ is of rank ($\rho$), then the Lanczos process converges (in exact arithmetic) in at most ($\rho$) iterations. If more generally $A$ can be expressed in the form

$$ A = M + C $$

where $M$ is symmetric positive-definite and $C$ has rank ($\rho$), then

$$ -(1/2) \quad -(1/2) \quad -(1/2) \quad -(1/2) $$

$$ M \quad AM \quad = I + M \quad CM $$

$$ -(1/2) \quad -(1/2) $$

and $M$ $AM$ also has rank ($\rho$), and the Lanczos process

$$ -(1/2) \quad -(1/2) $$

applied to $M$ $AM$ would again converge in at most ($\rho$) iterations. On a computer, the number of iterations may be greater than ($\rho$), but the Lanczos process may still be expected

$$ -(1/2) \quad -(1/2) $$

to converge rapidly. F04MBF does not require $M$ $AM$ to be formed implicitly, but implicitly solves the equations

$$ -(1/2) \quad -(1/2) \quad -(1/2) \quad 1/2 $$

$$ M \quad (A-(\lambda)I)M \quad y = M \quad b, \quad y = M \quad x \quad (3.2) $$

with the user being required to supply a routine MSOLVE to solve the equations

$$ Mz = c. \quad (3.3) $$

For the pre-conditioning option to be effective, it is desirable that equations (3.3) can be solved efficiently. The example program in Section 9 illustrates the use of this option.

If we let $r$ denote the residual vector

$$ r = b - (A-(\lambda)I)x $$

corresponding to an iterate $x$, then, when pre-conditioning has not been requested, the iterative procedure is terminated if it is estimated that

$$ ||r|| \leq tol.||A-(\lambda)I||||x||, \quad (3.4) $$
where tol is a user-supplied tolerance, \(|r|\) denotes the Euclidean length of the vector \(r\) and \(|A|\) denotes the Frobenius (Euclidean) norm of the matrix \(A\). When pre-conditioning has been requested, the iterative procedure is terminated if it is estimated that
\[
\| M r \| \leq \text{tol} \| M (A - \lambda I) M \| \| M x \|. \quad (3.5)
\]
Note that
\[
M r = (M b) - M (A - \lambda I) M (M x)
\]
so that \(M r\) is the residual vector corresponding to equation (3.2). The routine will also terminate if it is estimated that
\[
\| A - \lambda I \| \| x \| \geq \| b \| / \epsilon, \quad (3.6)
\]
where \(\epsilon\) is the machine precision, when pre-conditioning has not been requested; or if it is estimated that
\[
\| M (A - \lambda I) M \| \| M x \| \geq \| M b \| / \epsilon \quad (3.7)
\]
when pre-conditioning has been requested. If (3.6) is satisfied then \(x\) is almost certainly an eigenvector of \(A\) corresponding to the eigenvalue \(\lambda\). If \(\lambda\) was set to 0 (for the solution of \(Ax=b\)), then this condition simply means that \(A\) is effectively singular.

4. References


5. Parameters

1: \(N\) -- INTEGER Input
On entry: \(n\), the order of the matrix \(A\). Constraint: \(N \geq 1\).
2: B(N) -- DOUBLE PRECISION array
   Input
   On entry: the right-hand side vector b.

3: X(N) -- DOUBLE PRECISION array
   Output
   On exit: the solution vector x.

4: APROD -- SUBROUTINE, supplied by the user.
   External Procedure
   APROD must return the vector y=Ax for a given vector x.

Its specification is:

```fortran
SUBROUTINE APROD (IFLAG, N, X, Y, RWORK, LRWORK,
  1   IWORK, LIWORK)
   INTEGER IFLAG, N, LRWORK, LIWORK, IWORK
  1   (LIWORK)
   DOUBLE PRECISION X(N), Y(N), RWORK(LRWORK)
```

1: IFLAG -- INTEGER
   Input/Output
   On entry: IFLAG is always non-negative. On exit: IFLAG
   may be used as a flag to indicate a failure in the
   computation of Ax. If IFLAG is negative on exit from
   APROD, F04MBF will exit immediately with IFAIL set to
   IFLAG.

2: N -- INTEGER
   Input
   On entry: n, the order of the matrix A.

3: X(N) -- DOUBLE PRECISION array
   Input
   On entry: the vector x for which Ax is required.

4: Y(N) -- DOUBLE PRECISION array
   Output
   On exit: the vector y=Ax.

5: RWORK(LRWORK) -- DOUBLE PRECISION array
   User Workspace

6: LRWORK -- INTEGER
   Input

7: IWORK(LIWORK) -- INTEGER array
   User Workspace

8: LIWORK -- INTEGER
   Input
   APROD is called from F04MBF with the parameters RWORK,
   LRWORK, IWORK and LIWORK as supplied to F04MBF. The
   user is free to use the arrays RWORK and IWORK to
   supply information to APROD and MSOLVE as an
   alternative to using COMMON.

APROD must be declared as EXTERNAL in the (sub)program
from which F04MBF is called. Parameters denoted as
Input must not be changed by this procedure.
5: MSOLVE -- SUBROUTINE, supplied by the user.

External Procedure

MSOLVE is only referenced when PRECON is supplied as .TRUE..
When PRECON is supplied as .FALSE., then F04MBF may be
called with APROD as the actual argument for MSOLVE. When
PRECON is supplied as .TRUE., then MSOLVE must return the
solution y of the equations My=x for a given vector x, where
M must be symmetric positive-definite.

Its specification is:

SUBROUTINE MSOLVE (IFLAG, N, X, Y, RWORK,
  1      LRWORK, IWORK,LIWORK)
  INTEGER      IFLAG, N, LRWORK, LIWORK, IWORK
  1      (LIWORK)
  DOUBLE PRECISION X(N), Y(N), RWORK(LRWORK)

1: IFLAG -- INTEGER     Input/Output
On entry: IFLAG is always non-negative. On exit: IFLAG
may be used as a flag to indicate a failure in the
solution of My=x.

If IFLAG is negative on exit from MSOLVE, F04MBF will
exit immediately with IFAIL set to IFLAG.

2: N -- INTEGER     Input
On entry: n, the order of the matrix M.

3: X(N) -- DOUBLE PRECISION array     Input
On entry: the vector x for which the equations My=x are
to be solved.

4: Y(N) -- DOUBLE PRECISION array     Output
On exit: the solution to the equations My=x.

5: RWORK(LRWORK) -- DOUBLE PRECISION array     User Workspace

6: LRWORK -- INTEGER     Input

7: IWORK(LIWORK) -- INTEGER array     User Workspace

8: LIWORK -- INTEGER     Input
MSOLVE is called from F04MBF with the parameters RWORK,
LRWORK, IWORK and LIWORK as supplied to F04MBF. The
user is free to use the arrays RWORK and IWORK to
supply information to APROD and MSOLVE as an
alternative to using COMMON.

MSOLVE must be declared as EXTERNAL in the (sub)program
from which F04MBF is called. Parameters denoted as
Input must not be changed by this procedure.
6: PRECON -- LOGICAL
   On entry: PRECON specifies whether or not pre-conditioning
   is required. If PRECON = .TRUE., then pre-conditioning will
   be invoked and MSOLVE will be referenced by F04MBF; if
   PRECON = .FALSE., then MSOLVE is not referenced.

7: SHIFT -- DOUBLE PRECISION
   On entry: the value of (lambda). If the equations Ax=b are
   to be solved, then SHIFT must be supplied as zero.

8: RTOL -- DOUBLE PRECISION
   On entry: the tolerance for convergence, tol, of equation
   (3.4). RTOL should not normally be less than (epsilon),
   where (epsilon) is the machine precision.

9: ITNLIM -- INTEGER
   On entry: an upper limit on the number of iterations. If
   ITNLIM <= 0, then the value N is used in place of ITNLIM.

10: MSGLVL -- INTEGER
    On entry: the level of printing from F04MBF. If MSGLVL <= 0,
    then no printing occurs, but otherwise messages will be
    output on the advisory message channel (see X04ABF). A
    description of the printed output is given in Section 5.1
    below. The level of printing is determined as follows:
    MSGLVL <= 0
       No printing.
    MSGLVL = 1
       A brief summary is printed just prior to return from
       F04MBF.
    MSGLVL >= 2
       A summary line is printed periodically to monitor the
       progress of F04MBF, together with a brief summary just
       prior to return from F04MBF.

11: ITN -- INTEGER
    On exit: the number of iterations performed.

12: ANORM -- DOUBLE PRECISION
    On exit: an estimate of ||A-(lambda)I|| when PRECON =
             -(1/2) -(1/2)
    .FALSE., and ||M (A-(lambda)I)M || when PRECON =
             .TRUE..

13: ACOND -- DOUBLE PRECISION
    On exit: an estimate of the condition number of (A-
             (lambda)I) when PRECON = .FALSE., and of
\begin{align*}
\frac{-(1/2)}{-(1/2)}
M \quad (A-(\text{lambda}I))M \quad \text{when PRECON = .TRUE.}. \quad \text{This will usually be a substantial under-estimate.}
\end{align*}

14: RNORM -- DOUBLE PRECISION \quad \text{Output}
On exit: \|r\|, where \(r=b-(A-(\text{lambda}I))x\) and \(x\) is the solution returned in X.

15: XNORM -- DOUBLE PRECISION \quad \text{Output}
On exit: \|x\|, where \(x\) is the solution returned in X.

16: WORK(5*N) -- DOUBLE PRECISION array \quad \text{Workspace}

17: RWORK(LRWORK) -- DOUBLE PRECISION array \quad \text{User Workspace}
RWORK is not used by F04MBF, but is passed directly to routines APROD and MSOLVE and may be used to pass information to these routines.

18: LRWORK -- INTEGER \quad \text{Input}
On entry: the length of the array RWORK as declared in the (sub)program from which F04MBF is called. Constraint: LRWORK >= 1.

19: IWORK(LIWORK) -- INTEGER array \quad \text{User Workspace}
IWORK is not used by F04MBF, but is passed directly to routines APROD and MSOLVE and may be used to pass information to these routines.

20: LIWORK -- INTEGER \quad \text{Input}
On entry: the length of the array IWORK as declared in the (sub)program from which F04MBF is called. Constraint: LIWORK >= 1.

21: INFORM -- INTEGER \quad \text{Output}
On exit: the reason for termination of F04MBF as follows:
\begin{align*}
\text{INFORM} &= 0 \\
& \quad \text{The right-hand side vector } b=0 \text{ so that the exact solution is } x=0. \text{ No iterations are performed in this case.}
\end{align*}
\begin{align*}
\text{INFORM} &= 1 \\
& \quad \text{The termination criterion of equation (3.4) has been satisfied with tol as the value supplied in RTOL.}
\end{align*}
\begin{align*}
\text{INFORM} &= 2 \\
& \quad \text{The termination criterion of equation (3.4) has been satisfied with tol equal to (epsilon), where (epsilon) is the machine precision. The value supplied in RTOL must have been less than (epsilon) and was too small for the machine.}
\end{align*}
INFORM = 3

The termination criterion of equation (3.5) has been satisfied so that X is almost certainly an eigenvector of A corresponding to the eigenvalue SHIFT. The values INFORM = 4 and INFORM = 5 correspond to failure with IFAIL = 3 or IFAIL = 2 respectively (see Section 6) and when IFAIL is negative, INFORM will be set to the same negative value.

22: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

5.1. Description of the Printed Output

When MSGLVL > 0, then F04MBF will produce output (except in the case where the routine fails with IFAIL = 1) on the advisory message channel (see X04ABF).

The following notation is used in the output.

Output Meaning

-\frac{1}{2} (b-(A-(\lambda)I)x)=r
\text{RBAR} (b-(A-(\lambda)I)x)=r

-\frac{1}{2} -\frac{1}{2} (A-(\lambda)I)M =A
\text{ABAR} (A-(\lambda)I)M =A

\frac{1}{2}
Y M x

R b-(A-(\lambda)I)x

NORM(A) ||A||

Of course, when pre-conditioning has not been requested then the first three reduce to (b-(A-(\lambda)I)x), (A-(\lambda)I) and x respectively. When MSGLVL >= 2 then some initial information is printed and the following notation is used.

Output Meaning

T -1 1/2
BETA1 (b M b) ==(beta)
and a summary line is printed periodically giving the following information:

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITN</td>
<td>Iteration number, k.</td>
</tr>
<tr>
<td>(X_1(LQ))</td>
<td>The first element of the vector (x), where (x) is the kth current iterate. See Paige and Saunders [2] for details.</td>
</tr>
<tr>
<td>(X_1(CG))</td>
<td>The first element of the vector (x), where (x) is the kth vector that would be obtained by conjugate gradients. See Paige and Saunders [2] for details.</td>
</tr>
<tr>
<td>(\text{NORM}(\text{RBAR}))</td>
<td>(</td>
</tr>
<tr>
<td>(\text{NORM}(T))</td>
<td>The value (</td>
</tr>
<tr>
<td>(\text{COND}(L))</td>
<td>A monotonically increasing lower bound on the condition number of (A), (</td>
</tr>
<tr>
<td>(LQ/CG)</td>
<td>L is printed if (x) is the best current (k) approximation to the solution and C is printed otherwise.</td>
</tr>
</tbody>
</table>
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL< 0
   A negative value of IFAIL indicates an exit from F04MBF
   because the user has set IFLAG negative in APROD or MSOLVE.
   The value of IFAIL will be the same as the user's setting of
   IFLAG.

IFAIL= 1
   On entry N < 1,
   or                      LRWORK < 1,
   or                      LIWORK < 1.

IFAIL= 2
   The pre-conditioning matrix M does not appear to be
   positive-definite. The user should check that MSOLVE is
   working correctly.

IFAIL= 3
   The limit on the number of iterations has been reached. If
   IFAIL = 1 on entry then the latest approximation to the
   solution is returned in X and the values ANORM, ACOND, RNORM
   and XNORM are also returned.

The value of INFORM contains additional information about the
termination of the routine and users must examine INFORM to judge
whether the routine has performed successfully for the problem in
hand. In particular INFORM = 3 denotes that the matrix A-
(\lambda)I is effectively singular: if the purpose of calling
F04MBF is to solve a system of equations Ax=b, then this
condition must be regarded as a failure, but if the purpose is to
compute an eigenvector, this result would be very satisfactory.

7. Accuracy

The computed solution X will satisfy the equation
\[ r = b - (A-(\lambda)I)x \]
where the value $\|r\|$ is as returned in the parameter RNORM.

8. Further Comments

The time taken by the routine is likely to be principally
determined by the time taken in APROD and, when pre-conditioning
has been requested, in MSOLVE. Each of these routines is called once every iteration.

The time taken by the remaining operations in F04MBF is approximately proportional to \( n \).

9. Example

To solve the 10 equations \( Ax = b \) given by

\[
\begin{align*}
(2 & 1 0 0 0 0 0 0 0 3) \\
(1 & 2 1 0 0 0 0 0 0 0) \\
(0 & 1 2 1 0 0 0 0 0 0) \\
(0 & 0 1 2 1 0 0 0 0 0) \\
(0 & 0 0 1 2 1 0 0 0 0) \\
(0 & 0 0 0 1 2 1 0 0 0), & \quad b = (4). \\
(0 & 0 0 0 0 1 2 1 0 0) \\
(0 & 0 0 0 0 0 1 2 1 0) \\
(0 & 0 0 0 0 0 0 1 2 1) \\
(3 & 0 0 0 0 0 0 1 2) \\
\end{align*}
\]

The tridiagonal part of \( A \) is positive-definite and such tridiagonal equations can be solved efficiently by F04FAF. The form of \( A \) suggests that this tridiagonal part is a good candidate for the pre-conditioning matrix \( M \) and so we illustrate the use of F04MBF by pre-conditioning with the 10 by 10 matrix

\[
\begin{pmatrix}
(2 & 1 & 0 & \ldots & 0) \\
(1 & 2 & 1 & \ldots & 0) \\
(0 & 1 & 2 & \ldots & 0) \\
\end{pmatrix}
\]

\[
M = (\ldots \ldots \ldots) . \\
(\ldots \ldots) \\
(\ldots \ldots) \\
(0 \ 0 \ 0 \ \ldots \ 2)
\]

Since \( A - M \) has only 2 non-zero elements and is obviously of rank 2, we can expect F04MBF to converge very quickly in this example. Of course, in practical problems we shall not usually be able to make such a good choice of \( M \).

The example sets the tolerance \( \text{RTOL} = 10^{-5} \).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F04 -- Simultaneous Linear Equations
F04MCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F04MCF computes the approximate solution of a system of real linear equations with multiple right-hand sides, $AX=B$, where $A$ is a symmetric positive-definite variable-bandwidth matrix, which has previously been factorized by F01MCF. Related systems may also be solved.

2. Specification

```fortran
SUBROUTINE F04MCF (N, AL, LAL, D, NROW, IR, B, NRB,
                     ISELCT, X, NRX, IFAIL)

INTEGER N, LAL, NROW(N), IR, NRB, ISELCT, NRX,
       IFAIL
DOUBLE PRECISION AL(LAL), D(N), B(NRB,IR), X(NRX,IR)
```

3. Description

The normal use of this routine is the solution of the systems $AX=B$, following a call of F01MCF to determine the Cholesky factorization $A=LDL^T$ of the symmetric positive-definite variable-bandwidth matrix $A$.

However, the routine may be used to solve any one of the following systems of linear algebraic equations:

- $T$
  - (1) $LDL^T X = B$ (usual system),
  - (2) $LDX = B$ (lower triangular system),
  - (3) $DL^T X = B$ (upper triangular system),
  - (4) $LL^T X = B$
  - (5) $LX = B$ (unit lower triangular system),
  - (6) $L^T X = B$ (unit upper triangular system).
L denotes a unit lower triangular variable-bandwidth matrix of order n, D a diagonal matrix of order n, and B a set of right-hand sides.

The matrix L is represented by the elements lying within its envelope i.e., between the first non-zero of each row and the diagonal (see Section 9 for an example). The width NROW(i) of the ith row is the number of elements between the first non-zero element and the element on the diagonal inclusive.

4. References


5. Parameters

1: N -- INTEGER Input
On entry: n, the order of the matrix L. Constraint: N >= 1.

2: AL(LAL) -- DOUBLE PRECISION array Input
On entry: the elements within the envelope of the lower triangular matrix L, taken in row by row order, as returned by F01MCF. The unit diagonal elements of L must be stored explicitly.

3: LAL -- INTEGER Input
On entry: the dimension of the array AL as declared in the (sub)program from which F04MCF is called. Constraint: LAL >= NROW(1) + NROW(2) + ... + NROW(n).

4: D(N) -- DOUBLE PRECISION array Input
On entry: the diagonal elements of the diagonal matrix D. D is not referenced if ISELCT >= 4.

5: NROW(N) -- INTEGER array Input
On entry: NROW(i) contains the width of row i of L, i.e., the number of elements between the first (leftmost) non-zero element and the element on the diagonal, inclusive. Constraint: 1 <= NROW(i) <= i.

6: IR -- INTEGER Input
On entry: r, the number of right-hand sides. Constraint: IR >= 1.

7: B(NRB,IR) -- DOUBLE PRECISION array Input
On entry: the n by r right-hand side matrix B. See also Section 8.
8: NRB -- INTEGER
   Input
   On entry:
   the first dimension of the array B as declared in the
   (sub)program from which F04MCF is called.
   Constraint: NRB >= N.

9: ISELCT -- INTEGER
   Input
   On entry: ISELCT must specify the type of system to be
   solved, as follows:

   T
   ISELCT = 1: solve LDL X = B,
   ISELCT = 2: solve LDX = B,

   T
   ISELCT = 3: solve DL X = B,

   T
   ISELCT = 4: solve LL X = B,
   ISELCT = 5: solve LX = B,

   T
   ISELCT = 6: solve L X = B.

10: X(NRX,IR) -- DOUBLE PRECISION array
    Output
    On exit: the n by r solution matrix X. See also Section 8.

11: NRX -- INTEGER
    Input
    On entry:
    the first dimension of the array X as declared in the
    (sub)program from which F04MCF is called.
    Constraint: NRX >= N.

12: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry N < 1,
or for some $i$, $\text{NROW}(i) < 1$ or $\text{NROW}(i) > i$,
or $\text{LAL} < \text{NROW}(1) + \text{NROW}(2) + \ldots + \text{NROW}(N)$.

**IFAIL= 2**

On entry $\text{IR} < 1$,
or $\text{NRB} < N$,
or $\text{NRX} < N$.

**IFAIL= 3**

On entry $\text{ISELCT} < 1$,
or $\text{ISELCT} > 6$.

**IFAIL= 4**
The diagonal matrix $D$ is singular, i.e., at least one of the elements of $D$ is zero. This can only occur if $\text{ISELCT} \leq 3$.

**IFAIL= 5**
At least one of the diagonal elements of $L$ is not equal to unity.

7. Accuracy

The usual backward error analysis of the solution of triangular system applies: each computed solution vector is exact for slightly perturbed matrices $L$ and $D$, as appropriate (cf. Wilkinson and Reinsch [1] pp 25--27, 54--55).

8. Further Comments

The time taken by the routine is approximately proportional to $pr$, where

$$p = \text{NROW}(1) + \text{NROW}(2) + \ldots + \text{NROW}(n).$$

Unless otherwise stated in the Users' Note for your implementation, the routine may be called with the same actual array supplied for the parameters $B$ and $X$, in which case the solution matrix will overwrite the right-hand side matrix. However this is not standard Fortran 77 and may not work in all implementations.

9. Example

To solve the system of equations $AX = B$, where
Here \( A \) is symmetric and positive-definite and must first be factorized by F01MCF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
CHAPTER 15. CHAPTER N

3. Description

F04QAF can be used to solve a system of linear equations

\[ Ax = b \quad (3.1) \]

where \( A \) is an \( n \) by \( n \) sparse unsymmetric matrix, or can be used to solve linear least-squares problems, so that F04QAF minimizes the value \( \rho \) given by

\[ \rho = ||r||, \quad r = b - Ax \quad (3.2) \]

where \( A \) is an \( m \) by \( n \) sparse matrix and \( ||r|| \) denotes the Euclidean length of \( r \) so that \( ||r|| = \sqrt{r^T r} \). A damping parameter, \( \lambda \), may be included in the least-squares problem in which case F04QAF minimizes the value \( \rho \) given by

\[ \rho = ||r||^2 + \lambda ||x||^2 \quad (3.3) \]

(\( \lambda \)) is supplied as the parameter DAMP and should of course be zero if the solution to problems (3.1) or (3.2) is required. Minimizing \( \rho \) in (3.3) is often called ridge regression.

F04QAF is based upon algorithm LSQR (see Paige and Saunders [1] and [2]) and solves the problems by an algorithm based upon the Lanczos process. Details of the method are given in [1]. The routine does not require \( A \) explicitly, but \( A \) is specified via a user-supplied routine APROD which must perform the operations \( y + Ax \) and \( x + A^T y \) for a given \( n \) element vector \( x \) and \( m \) element vector \( y \). A parameter to APROD specifies which of the two operations is required on a given entry.

The routine also returns estimates of the standard errors of the sample regression coefficients \( x_i \) for \( i = 1, 2, ..., n \) given by the diagonal elements of the estimated variance-covariance matrix \( V \). When problem (3.2) is being solved and \( A \) is of full rank, then \( V \) is given by

\[ V = s \left( A^T A \right)^{-1} s, \quad s = \frac{\rho}{(m-n)}, \quad m > n \]

and when problem (3.3) is being solved then \( V \) is given by
\[
V = s \left( A A + (\lambda) I \right), \quad s = (\rho) / m, \quad (\lambda) /= 0.
\]

Let \( A \) denote the matrix

\[
A = A, \quad (\lambda) = 0 ; \quad A = ((\lambda) I), \quad (\lambda) /= 0, \quad (3.4)
\]

let \( r \) denote the residual vector

\[
r = r, \quad (\lambda) = 0 ; \quad r = (0) - Ax, \quad (\lambda) /= 0 \quad (3.5)
\]

corresponding to an iterate \( x \), so that \((\rho) = ||r||\) is the function being minimized, and let \( ||A|| \) denote the Frobenius (Euclidean) norm of \( A \). Then the routine accepts \( x \) as a solution if it is estimated that one of the following two conditions is satisfied:

\[
(rho) <= tol ||A|| ||x|| + tol ||b|| \quad (3.6)
\]

\[
||A r|| <= tol ||A|| (rho) \quad (3.7)
\]

where \( tol \) and \( tol \) are user-supplied tolerances which estimate the relative errors in \( A \) and \( b \) respectively. Condition (3.6) is appropriate for compatible problems where, in theory, we expect the residual to be zero and will be satisfied by an acceptable solution \( x \) to a compatible problem. Condition (3.7) is appropriate for incompatible systems where we do not expect the residual to be zero and is based upon the observation that, in theory,

\[
T
A r = 0
\]

when \( x \) is a solution to the least-squares problem, and so (3.7) will be satisfied by an acceptable solution \( x \) to a linear least-
squares problem.

The routine also includes a test to prevent convergence to solutions, \( \mathbf{x} \), with unacceptably large elements. This can happen if \( \mathbf{A} \) is nearly singular or is nearly rank deficient. If we let the singular values of \( \mathbf{A} \) be

\[
(\sigma_1) \geq (\sigma_2) \geq \ldots \geq (\sigma_n) \geq 0
\]

then the condition number of \( \mathbf{A} \) is defined as

\[
\text{cond}(\mathbf{A}) = \frac{(\sigma_1)}{(\sigma_k)}
\]

where \((\sigma_1)\) is the smallest non-zero singular value of \( \mathbf{A} \) and \( k \) hence \( k \) is the rank of \( \mathbf{A} \). When \( k < n \), then \( \mathbf{A} \) is rank deficient, the least-squares solution is not unique and F04QAF will normally converge to the minimal length solution. In practice \( \mathbf{A} \) will not have exactly zero singular values, but may instead have small singular values that we wish to regard as zero.

The routine provides for this possibility by terminating if

\[
\text{cond}(\mathbf{A}) \geq c \quad \text{(3.8)}
\]

where \( c \) is a user-supplied limit on the condition number of \( \mathbf{A} \).

For problem (3.1) termination with this condition indicates that \( \mathbf{A} \) is nearly singular and for problem (3.2) indicates that \( \mathbf{A} \) is nearly rank deficient and so has near linear dependencies in its
In this case inspection of $\|r\|$, $\|Ar\|$ and $\|x\|$, which are all returned by the routine, will indicate whether or not an acceptable solution has been found. Condition (3.8), perhaps in conjunction with $(\lambda)/=0$, can be used to try and 'regularise' least-squares solutions. A full discussion of the stopping criteria is given in Section 6 of reference Paige and Saunders [1].

Introduction of a non-zero damping parameter $(\lambda)$ tends to reduce the size of the computed solution and to make its components less sensitive to changes in the data, and F04QAF is applicable when a value of $(\lambda)$ is known a priori. To have an effect, $(\lambda)$ should normally be at least $\sqrt{\epsilon}\|A\|$ where $(\epsilon)$ is the machine precision. For further discussion see Paige and Saunders [2] and the references given there.

Whenever possible the matrix $A$ should be scaled so that the relative errors in the elements of $A$ are all of comparable size. Such a scaling helps to prevent the least-squares problem from being unnecessarily sensitive to data errors and will normally reduce the number of iterations required. At the very least, in the absence of better information, the columns of $A$ should be scaled to have roughly equal column length.

4. References


5. Parameters

1: $M$ -- INTEGER  
   Input
   On entry: $m$, the number of rows of the matrix $A$.
   Constraint: $M \geq 1$.

2: $N$ -- INTEGER  
   Input
   On entry: $n$, the number of columns of the matrix $A$.
   Constraint: $N \geq 1$.

3: $B(M)$ -- DOUBLE PRECISION array  
   Input/Output
   On entry: the right-hand side vector $b$. On exit: the array is overwritten.
<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
</table>
| 4:  | **X(N)** -- DOUBLE PRECISION array  
     On exit: the solution vector x. |
| 5:  | **SE(N)** -- DOUBLE PRECISION array  
     On exit: the estimates of the standard errors of the components of x. Thus SE(i) contains an estimate of the element $v_{ii}$ of the estimated variance-covariance matrix $V$.  
     The estimates returned in SE will be the lower bounds on the actual estimated standard errors, but will usually have at least one correct figure. |
| 6:  | **APROD** -- SUBROUTINE, supplied by the user.  
     **External Procedure**  
     APROD must perform the operations $y := y + Ax$ and $x := x + A^T y$ for given vectors $x$ and $y$. |

Its specification is:

```fortran
SUBROUTINE APROD (MODE, M, N, X, Y, RWORK, LRWORK, IWORK, LIWORK)
    INTEGER MODE, M, N, LRWORK, LIWORK,
    IWORK(LIWORK)
    DOUBLE PRECISION X(N), Y(M), RWORK(LRWORK)
```

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
</tr>
</thead>
</table>
| 1:  | **MODE** -- INTEGER  
     Input/Output  
     On entry: MODE specifies which operation is to be performed:  
     If MODE = 1, then APROD must compute $y + Ax$.  
     If MODE = 2, then APROD must compute $x + A^T y$.  
     On exit: MODE may be used as a flag to indicate a failure in the computation of $y + Ax$ or $x + A^T y$. If MODE is negative on exit from APROD, F04QAF will exit immediately with IFAIL set to MODE. |
| 2:  | **M** -- INTEGER  
     Input  
     On entry: m, the number of rows of A. |
| 3:  | **N** -- INTEGER  
     Input  
     On entry: n, the number of columns of A. |
| 4:  | **X(N)** -- DOUBLE PRECISION array  
     Input/Output  
     On entry: the vector x. On exit: if MODE = 1, X must be unchanged; |
If \( \text{MODE} = 2 \), \( X \) must contain \( x + A \ y \).

5: \( Y(M) \) -- DOUBLE PRECISION array
   Input/Output
   On entry: the vector \( y \). On exit: if \( \text{MODE} = 1 \), \( Y \) must contain \( y + Ax \);

   If \( \text{MODE} = 2 \), \( Y \) must be unchanged.

6: \( \text{RWORK( LRWORK) } \) -- DOUBLE PRECISION array User Workspace

7: \( \text{LRWORK} \) -- INTEGER Input

8: \( \text{IWORK( LIWORK) } \) -- INTEGER array User Workspace

9: \( \text{LIWORK} \) -- INTEGER Input
   APROD is called from F04QAF with the parameters \( \text{RWORK} \), \( \text{LRWORK} \), \( \text{IWORK} \) and \( \text{LIWORK} \) as supplied to F04QAF. The user is free to use the arrays \( \text{RWORK} \) and \( \text{IWORK} \) to supply information to APROD as an alternative to using COMMON.
   APROD must be declared as EXTERNAL in the (sub)program from which F04QAF is called. Parameters denoted as Input must not be changed by this procedure.

7: \( \text{DAMP} \) -- DOUBLE PRECISION Input
   On entry: the value (\( \lambda \)). If either problem (3.1) or problem (3.2) is to be solved, then\( \text{DAMP} \) must be supplied as zero.

8: \( \text{ATOL} \) -- DOUBLE PRECISION Input
   On entry: the tolerance, \( \text{tol} \), of the convergence criteria (3.6) and (3.7); it should be an estimate of the largest relative error in the elements of \( A \). For example, if the elements of \( A \) are correct to about 4 significant figures, then \( \text{ATOL} \) should be set to about \( 5 \times 10^{-4} \). If \( \text{ATOL} \) is supplied as less than \( \text{epsilon} \), where \( \text{epsilon} \) is the machine precision, the value (\( \text{epsilon} \)) is used in place of \( \text{ATOL} \).

9: \( \text{BTOL} \) -- DOUBLE PRECISION Input
   On entry: the tolerance, \( \text{tol} \), of the convergence criterion (3.6); it should be an estimate of the largest relative error in the elements of \( B \). For example, if the elements of \( B \) are correct to about 4 significant figures, then \( \text{BTOL} \) should be set to about \( 5 \times 10^{-4} \). If \( \text{BTOL} \) is supplied as less than \( \text{epsilon} \), where \( \text{epsilon} \) is the machine precision,
then the value (epsilon) is used in place of BTOL.

10: CONLIM -- DOUBLE PRECISION Input
On entry: the value \( c \) of equation (3.8); it should be an upper limit on the condition number of \( A \). CONLIM should not normally be chosen much larger than \( 1.0/\text{ATOL} \). If CONLIM is supplied as zero then the value \( 1.0/(\text{epsilon}) \), where (epsilon) is the machine precision, is used in place of CONLIM.

11: ITNLIM -- INTEGER Input
On entry: an upper limit on the number of iterations. If ITNLIM <= 0, then the value \( N \) is used in place of ITNLIM, but for ill-conditioned problems a higher value of ITNLIM is likely to be necessary.

12: MSG_LVL -- INTEGER Input
On entry: the level of printing from F04QAF. If MSG_LVL <= 0, then no printing occurs, but otherwise messages will be output on the advisory message channel (see X04ABF). A description of the printed output is given in Section 5.2 below. The level of printing is determined as follows:

\[
\begin{align*}
\text{MSG_LVL} & \leq 0 \\
\text{MSG_LVL} & = 1 \\
\text{MSG_LVL} & \geq 2 \\
\end{align*}
\]

No printing.

\[
\begin{align*}
\text{MSG_LVL} = 1 & \\
& \text{A brief summary is printed just prior to return from F04QAF.} \\
\text{MSG_LVL} = 2 & \\
& \text{A summary line is printed periodically to monitor the progress of F04QAF, together with a brief summary just prior to return from F04QAF.} \\
\end{align*}
\]

13: ITN -- INTEGER Output
On exit: the number of iterations performed.

14: ANORM -- DOUBLE PRECISION Output
On exit: an estimate of \( ||A|| \) for the matrix \( A \) of equation (3.4).

15: ACOND -- DOUBLE PRECISION Output
On exit: an estimate of \( \text{cond}(A) \) which is a lower bound.
16: RNORM -- DOUBLE PRECISION
Output
On exit: an estimate of $\|r\|$ for the residual, $r$, of equation (3.5) corresponding to the solution $x$ returned in $X$. Note that $\|r\|$ is the function being minimized.

17: ARNORM -- DOUBLE PRECISION
Output
$T$
On exit: an estimate of the $\|A r\|$ corresponding to the solution $x$ returned in $X$.

18: XNORM -- DOUBLE PRECISION
Output
On exit: an estimate of $\|x\|$ for the solution $x$ returned in $X$.

19: WORK(2*N) -- DOUBLE PRECISION array
Workspace

20: RWORK(LRWORK) -- DOUBLE PRECISION array
User Workspace
RWORK is not used by F04QAF, but is passed directly to routine APROD and may be used to pass information to that routine.

21: LRWORK -- INTEGER
Input
On entry: the length of the array RWORK as declared in the (sub)program from which F04QAF is called. Constraint: LRWORK $\geq 1$.

22: IWORK(LIWORK) -- INTEGER array
User Workspace
IWORK is not used by F04QAF, but is passed directly to routine APROD and may be used to pass information to that routine.

23: LIWORK -- INTEGER
Input
On entry: the length of the array IWORK as declared in the (sub)program from which F04QAF is called. Constraint: LIWORK $\geq 1$.

24: INFORM -- INTEGER
Output
On exit: the reason for termination of F04QAF as follows:
INFORM = 0
The exact solution is $x=0$. No iterations are performed in this case.

INFORM = 1
The termination criterion of equation (3.6) has been satisfied with tol and tol as the values supplied in
ATOL and BTOL respectively.

INFORM = 2
The termination criterion of equation (3.7) has been satisfied with \( \text{tol} \) as the value supplied in ATOL.

INFORM = 3
The termination criterion of equation (3.6) has been satisfied with \( \text{tol} \) and/or \( \text{tol} \) as the value (epsilon), where (epsilon) is the machine precision. One or both of the values supplied in ATOL and BTOL must have been less than (epsilon) and was too small for this machine.

INFORM = 4
The termination criterion of equation (3.7) has been satisfied with \( \text{tol} \) as the value (epsilon), where (epsilon) is the machine precision. The value supplied in ATOL must have been less than (epsilon) and was too small for this machine.

The values \( \text{INFORM} = 5 \), \( \text{INFORM} = 6 \) and \( \text{INFORM} = 7 \) correspond to failure with \( \text{IFAIL} = 2 \), \( \text{IFAIL} = 3 \) and \( \text{IFAIL} = 4 \) respectively (see Section 6) and when \( \text{IFAIL} \) is negative \( \text{INFORM} \) will be set to the same negative value.

25: IFAIL -- INTEGER
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

5.1. Description of the printed output

When MSGLVL > 0, then F04QAF will produce output (except in the case where the routine fails with IFAIL = 1) on the advisory message channel (see X04ABF ).

When MSGLVL >= 2 then a summary line is printed periodically giving the following information:

Output Meaning
ITN Iteration number, k.
X(1) The first element of the current iterate $x_k$.

FUNCTION The current value of the function, $(\rho)$, being minimized.

COMPAT An estimate of $||r||/||b||$, where $r$ is the residual corresponding to $x_k$. This value should converge to zero (in theory) if and only if the problem is compatible. COMPAT decreases monotonically.

INCOMPAT An estimate of $||A||/(||A||||r||)$ which should converge to zero if and only if at the solution $(\rho)$ is non-zero. INCOMPAT is not usually monotonic.

NRM(ABAR) A monotonically increasing estimate of $||A||$.

COND(ABAR) A monotonically increasing estimate of the condition number cond($A$).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL< 0
A negative value of IFAIL indicates an exit from F04QAF because the user has set MODE negative in APROD. The value of IFAIL will be the same as the user's setting of MODE.

IFAIL= 1
On entry $M < 1$,
or $N < 1$,
or $LRWORK < 1$,
or $LIWORK < 1$. 
CHAPTER 15. CHAPTER N

IFAIL= 2
The condition of equation (3.8) has been satisfied for the value of \( c \) supplied in CONLIM. If this failure is unexpected the user should check that APROD is working correctly. Although conditions (3.6) or (3.7) have not been satisfied, the values returned in RNORM, ARNORM and XNORM may nevertheless indicate that an acceptable solution has been reached.

IFAIL= 3
The conditions of equation (3.8) has been satisfied for the value \( c = 1.0/(\epsilon) \), where \( \epsilon \) is the machine precision. The matrix \( A \) is nearly singular or rank deficient and the problem is too ill-conditioned for this machine. If this failure is unexpected, the user should check that APROD is working correctly.

IFAIL= 4
The limit on the number of iterations has been reached. The number of iterations required by F04QAF and the condition of the matrix \( A \) can depend strongly on the scaling of the problem. Poor scaling of the rows and columns of \( A \) should be avoided whenever possible.

7. Accuracy

When the problem is compatible, the computed solution \( x \) will satisfy the equation

\[
\mathbf{r} = \mathbf{b} - A\mathbf{x},
\]

where an estimate of \( \|\mathbf{r}\| \) is returned in the parameter RNORM. When the problem is incompatible, the computed solution \( x \) will satisfy the equation

\[
\mathbf{r}^T A = \mathbf{e},
\]

where an estimate of \( \|\mathbf{e}\| \) is returned in the parameter ARNORM. See also Section 6.2 of Paige and Saunders [1].

8. Further Comments

The time taken by the routine is likely to be principally
determined by the time taken in APROD, which is called twice on each iteration, once with MODE = 1 and once with MODE = 2. The time taken per iteration by the remaining operations in F04QAF is approximately proportional to max(m,n).

The Lanczos process will usually converge more quickly if A is pre-conditioned by a non-singular matrix M that approximates A in some sense and is also chosen so that equations of the form My=c can efficiently be solved for y. Some discussion of pre-conditioning in the context of symmetric matrices is given in Section 3 of the document for F04MBF. In the context of F04QAF, problem (3.1) is equivalent to

\[ (AM^{-1})y = b, \quad Mx = y \]

and problem (3.2) is equivalent to minimizing

\[ -(\rho) = ||r||, \quad r = b - (AM^{-1})y, \quad Mx = y. \]

\[ -(\rho) = ||r||, \quad r = b - (AM^{-1})y, \quad Mx = y. \]

Note that the normal matrix \((AM^{-1})(AM^{-1}) = M(AA^{-1})M^{-1}\) so that the pre-conditioning \(AM^{-1}\) is equivalent to the pre-conditioning

\[ -(\rho) = ||r||, \quad r = b - (AM^{-1})y, \quad Mx = y. \]

Pre-conditioning can be incorporated into F04QAF simply by coding the routine APROD to compute \(y + AM^{-1}x\) and \(x + M^{-1}A^{-1}y\) in place of \(y + Ax\) and \(x + A^{-1}y\) respectively, and then solving the equations \(Mx = y\) for \(x\) on return from F04QAF. \(y + AM^{-1}x\) should be computed by solving \(Mz = x\) for \(z\) and then computing \(y + Az\), and \(x + M^{-1}A^{-1}y\) should be computed by solving \(M^{-1}z = A^{-1}y\) for \(z\) and then forming \(x + z\).

9. Example

To solve the linear least-squares problem

\[ \minimize (\rho) = ||r||, \quad r = b - Ax \]

where \(A\) is the 13 by 12 matrix and \(b\) is the 13 element vector given by

\[
\begin{pmatrix}
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\]
Such a problem can arise by considering the Neumann problem on a rectangle

\[ \frac{\partial u}{\partial n} = 0 \]

where \( C \) is the boundary of the rectangle, and discretising as illustrated below with the square mesh.
Please see figure in printed Reference Manual

The 12 by 12 symmetric part of A represents the difference equations and the final row comes from the normalising condition. The example program has g(x,y)=1 at all the internal mesh points, but apart from this is written in a general manner so that the number of rows (NROWS) and columns (NCOOLS) in the grid can readily be altered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

NagLinearEquationSolvingPackage (NAGF04)

Exports:
  f04adf  f04arf  f04asf  f04atf  f04axf
  f04faf  f04jgf  f04maf  f04mbf  f04mcf
  f04qaf

--- package NAGF04 NagLinearEquationSolvingPackage ---

)abbrev package NAGF04 NagLinearEquationSolvingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:31 1994
++ Description:
++ This package uses the NAG Library to solve the matrix equation \br
++ \tab(5)axiom{AX=B}, where \axiom{B}\br
++ may be a single vector or a matrix of multiple right-hand sides.
++ The matrix \axiom{A} may be real, complex, symmetric, Hermitian positive-
CHAPTER 15. CHAPTER N

++ definite, or sparse. It may also be rectangular, in which case a
++ least-squares solution is obtained.

NagLinearEquationSolvingPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==> with
  f04adf : (Integer,Matrix Complex DoubleFloat,Integer,Integer,_
       Integer,Integer,Matrix Complex DoubleFloat,Integer) -> Result
    ++ f04adf(ia,b,ib,n,m,ic,a,ifail)
    ++ calculates the approximate solution of a set of complex
    ++ linear equations with multiple right-hand sides, using an LU
    ++ factorization with partial pivoting.
    ++ See \downlink{Manual Page}{manpageXXf04adf}.
  f04arf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
       Integer) -> Result
    ++ f04arf(ia,b,n,a,ifail)
    ++ calculates the approximate solution of a set of real
    ++ linear equations with a single right-hand side, using an LU
    ++ factorization with partial pivoting.
    ++ See \downlink{Manual Page}{manpageXXf04arf}.
  f04asf : (Integer,Matrix DoubleFloat,Integer,Matrix DoubleFloat,_
       Integer) -> Result
    ++ f04asf(ia,b,n,a,ifail)
    ++ calculates the accurate solution of a set of real
    ++ symmetric positive-definite linear equations with a single right-
    ++ hand side, Ax=b, using a Cholesky factorization and iterative
    ++ refinement.
    ++ See \downlink{Manual Page}{manpageXXf04asf}.
  f04atf : (Matrix DoubleFloat,Integer,Matrix DoubleFloat,Integer,_
       Integer,Integer) -> Result
    ++ f04atf(a,ia,b,n,iaa,ifail)
    ++ calculates the accurate solution of a set of real linear
    ++ equations with a single right-hand side, using an LU
    ++ factorization with partial pivoting, and iterative refinement.
    ++ See \downlink{Manual Page}{manpageXXf04atf}.
  f04axf : (Integer,Matrix DoubleFloat,Integer,Matrix Integer,_
       Matrix Integer,Integer,Matrix Integer,Integer,Matrix DoubleFloat) -> Result
    ++ f04axf(n,a,licn,icn,ikeep,mtype,idisp,rhs)
    ++ calculates the approximate solution of a set of real
    ++ sparse linear equations with a single right-hand side, Ax=b or
    ++ T
    ++ A x=b, where A has been factorized by F01BRF or F01BSF.
    ++ See \downlink{Manual Page}{manpageXXf04axf}.
  f04faf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_
       Matrix DoubleFloat,Integer) -> Result
    ++ f04faf(job,n,d,e,b,ifail)
    ++ calculates the approximate solution of a set of real
    ++ symmetric positive-definite tridiagonal linear equations.
f04jgf : (Integer, Integer, Integer, DoubleFloat, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Integer) \rightarrow Result
++ f04jgf(m, n, nra, tol, lwork, a, b, ifail)
++ finds the solution of a linear least-squares problem, Ax=b
++ , where A is a real m by n (m>=n) matrix and b is an m element
++ vector. If the matrix of observations is not of full rank, then
++ the minimal least-squares solution is returned.
++ See \downlink{Manual Page}{manpageXXf04jgf}.

f04maf : (Integer, Integer, Matrix DoubleFloat, Integer, Matrix Integer, Integer, Matrix Integer, Matrix DoubleFloat, Matrix Integer, Integer) \rightarrow Result
++ f04maf(n, nz, avals, licn, irn, lirn, icn, wkeep, ikeep,
++ inform, b, acc, noits, ifail)
++ e a sparse symmetric positive-definite system of linear
++ equations, Ax=b, using a pre-conditioned conjugate gradient
++ method, where A has been factorized by F01MAF.
++ See \downlink{Manual Page}{manpageXXf04maf}.

f04mbf : (Integer, Matrix DoubleFloat, Boolean, DoubleFloat, Integer, Integer, Integer, Integer, DoubleFloat, Integer, Union(fn:FileName, fp:Asp28(APROD)),
++ f04mbf(n, b, precon, shift, itnlim, msglvl, lrwork, liwork
++ rwork, rtol, ifail, aprod, msolve)
++ solves a system of real sparse symmetric linear equations
++ using a Lanczos algorithm.
++ See \downlink{Manual Page}{manpageXXf04mbf}.

f04mcf : (Integer, Matrix DoubleFloat, Integer, Matrix DoubleFloat, Integer, Integer, Integer, Integer) \rightarrow Result
++ f04mcf(n, al, la, alal, d, nrow, b, nrb, iselct, nr, ifail)
++ computes the approximate solution of a system of real
++ linear equations with multiple right-hand sides, AX=B, where A
++ is a symmetric positive-definite variable-bandwidth matrix, which
++ has previously been factorized by F01MCF. Related systems may
++ also be solved.
++ See \downlink{Manual Page}{manpageXXf04mcf}.

f04qaf : (Integer, Integer, DoubleFloat, DoubleFloat, DoubleFloat, Integer, Integer, Integer, Integer, Matrix DoubleFloat, Integer, Union(fn:FileName,
++ f04qaf(m, n, damp, atol, btol, conlim, itnlim, msglvl, lrwork, liwork, b, ifail, aprod)
++ solves sparse unsymmetric equations, sparse linear least-
++ squares problems and sparse damped linear least-squares problems,
++ using a Lanczos algorithm.
++ See \downlink{Manual Page}{manpageXXf04qaf}.

Implementation 
add
import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Boolean)
import AnyFunctions1(Matrix Complex DoubleFloat)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Integer)

f04adf(iaArg:Integer,bArg:Matrix Complex DoubleFloat,ibArg:Integer, 
  nArg:Integer,mArg:Integer,icArg:Integer, 
  aArg:Matrix Complex DoubleFloat,ifailArg:Integer): Result == 
  [(invokeNagman(NIL$Lisp, 
    "f04adf", 
    ["ia"::S,"ib"::S,"n"::S,"m"::S,"ic"::S, 
      "ifail"::S,"b"::S,"c"::S,"a"::S,"wkspce"::S]$Lisp, 
    ["c"::S,"wkspce"::S]$Lisp, 
    [["double"::S,"wkspce"::S,"n"::S]$Lisp]$Lisp, 
    ["integer"::S,"ia"::S,"ib"::S,"n"::S,"m"::S, 
      "ic"::S,"ifail"::S]$Lisp, 
    ["double complex"::S,"b"::S,"ib"::S,"m"::S]$Lisp, 
  "$c$"::S,"a"::S,"ifail"::S]$Lisp, 
  [[[iaArg::Any,ibArg::Any,nArg::Any,mArg::Any,icArg::Any, 
    ifailArg::Any,bArg::Any,aArg::Any ]]_ 
  @List Any]$Lisp)$Lisp)_ 
  pretend List (Record(key:Symbol,entry:Any))$Result

f04arf(iaArg:Integer,bArg:Matrix DoubleFloat,nArg:Integer, 
  aArg:Matrix DoubleFloat,ifailArg:Integer): Result == 
  [(invokeNagman(NIL$Lisp, 
    "f04arf", 
    ["c"::S,"wkspce"::S]$Lisp, 
    [["double"::S,"b"::S,"n"::S]$Lisp,["c"::S,"n"::S]$Lisp, 
      ["a"::S,"ia"::S,"n"::S]$Lisp,["wkspce"::S,"n"::S]$Lisp]$Lisp, 
    ["integer"::S,"ia"::S,"n"::S,"ifail"::S]$Lisp]$Lisp, 
  "$c$"::S,"a"::S,"ifail"::S]$Lisp, 
  [[[iaArg::Any,nArg::Any,ifailArg::Any,bArg::Any,aArg::Any ]]_ 
  @List Any]$Lisp)$Lisp)_ 
  pretend List (Record(key:Symbol,entry:Any))$Result
pretend List (Record(key:Symbol,entry:Any))$Result

f04asf(iaArg:Integer,bArg:Matrix DoubleFloat,nArg:Integer,_
aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "f04asf",_
  ]$Lisp,_
  ["c":S,"wk1":S,"wk2":S]$Lisp,_
  ["double":S,["b":S,"n":S]$Lisp,["c":S,"n":S]$Lisp,_
  ["a":S,"ia":S,"n":S]$Lisp,["wk1":S,"n":S]$Lisp,_
  ["wk2":S,"n":S]$Lisp]$Lisp$_
 ]$Lisp,_
 ["c":S,"a":S,"ifail":S]$Lisp,_
 [(iaArg::Any,nArg::Any,ifailArg::Any,bArg::Any,aArg::Any ])
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04atf(aArg:Matrix DoubleFloat,iaArg:Integer,bArg:Matrix DoubleFloat,_
[(invokeNagman(NIL$Lisp,_
  "f04atf",_
  "aa":S,"wks1":S,"wks2":S]$Lisp,_
  ["c":S,"aa":S,"wks1":S,"wks2":S]$Lisp,_
  ["double":S,["a":S,"ia":S,"n":S]$Lisp,_
  ,["b":S,"n":S]$Lisp,["c":S,"n":S]$Lisp,_
  ,["aa":S,"ia":S,"n":S]$Lisp,["wks1":S,"n":S]$Lisp,_
  ,["wk2":S,"n":S]$Lisp]$Lisp$_
 ["c":S,"aa":S,"ifail":S]$Lisp,_
 [(iaArg::Any,nArg::Any,iaArg::Any,ifailArg::Any,_
  aArg::Any,bArg::Any)]@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04axf(nArg:Integer,aArg:Matrix DoubleFloat,licnArg:Integer,_
icArg:Matrix Integer,ikeepArg:Matrix Integer,mtypeArg:Integer,_
idispArg:Matrix Integer,rhsArg:Matrix DoubleFloat): Result ==
[(invokeNagman(NIL$Lisp,_
  "f04axf",_
  "ikeep":S,"idisp":S,"rhs":S,"w":S]$Lisp,_
  ["resid":S,"w":S]$Lisp,_
  ,["double":S,["a":S,"licn":S]$Lisp,"resid":S_
  ,["rhs":S,"n":S]$Lisp,["w":S,"n":S]$Lisp]$Lisp$_
 ,["integer":S,"n":S,"licn":S,"icn":S]$Lisp,_
 ,["ikeep":S,["":S,"n":S]$Lisp]$Lisp]$Lisp,_
  ,["mtype":S,"idisp":S,2$Lisp]$Lisp]$Lisp)$Lisp,_
 )$Lisp,}
"resid":S,"rhs":S$Lisp,

[(nArg::Any,licnArg::Any,mtypeArg::Any,aArg::Any,icnArg::Any,_
ikeepArg::Any,idispArg::Any,rhsArg::Any )]
@List Any$Lisp$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04faf(jobArg:Integer,nArg:Integer,dArg:Matrix DoubleFloat,_
eArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
[]$Lisp,_
["double":S,["d":S,"n":S]$Lisp,["e":S,"n":S]$Lisp_,
["b":S,"n":S]$Lisp$Lisp_,
$Lisp,_
[(jobArg::Any,nArg::Any,ifailArg::Any,dArg::Any,eArg::Any,bArg::Any]_
@List Any$Lisp$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04jgf(mArg:Integer,nArg:Integer,nraArg:Integer,_
tolArg:DoubleFloat,lworkArg:Integer,aArg:Matrix DoubleFloat,_
bArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"f04jgf",_"m":S,"n":S,"nra":S,"tol":S,"lwork":S_,
"a":S,"b":S]$Lisp_,
["svd":S,"sigma":S,"irank":S,"work":S]$Lisp_,
["a":S,"nra":S,"n":S]$Lisp,["b":S,"m":S]$Lisp$Lisp_,
","irank":S,"ifail":S]$Lisp_,
["logical":S,"svd":S]$Lisp_]
$Lisp_,
["svd":S,"sigma":S,"irank":S,"work":S,"a":S_,
"b":S,"ifail":S]$Lisp_,
[(mArg::Any,nArg::Any,nraArg::Any,tolArg::Any,lworkArg::Any,_
ifailArg::Any,aArg::Any,bArg::Any]_
@List Any$Lisp$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f04maf(nArg:Integer,nzArg:Integer,avalsArg:Matrix DoubleFloat,_
licnArg:Integer,irnArg:Matrix Integer,lirnArg:Integer,_
icnArg:Matrix Integer,wkeepArg:Matrix DoubleFloat,_
ikeepArg:Matrix Integer,_
informArg:Matrix Integer,bArg:Matrix DoubleFloat,_
accArg:Matrix DoubleFloat,_
noitsArg: Matrix Integer, ifailArg: Integer): Result ==
[(invokeNagman(NIL$Lisp,"f04maf",_]
["work":S]$Lisp,_
[["double":S,"avals":S,"licn":S]$Lisp,_
["wkeep":S,["*":S,"n":S]$Lisp]$Lisp_,
["work":S,["*":S,"n":S]$Lisp]$Lisp,_
["b":S,"n":S]$Lisp,["acc":S,2$Lisp]$Lisp_]
)$Lisp_,
,"lirn":S,"licn":S]$Lisp,["ikeep":S,_
,["noits":S,2$Lisp]$Lisp,"ifail":S]$Lisp,,
[[nArg::Any,nzArg::Any,"licn":S,"lirn":S,ifailArg::Any,,
@List Any]$Lisp)$Lisp_)
pretend List (Record(key: Symbol, entry: Any))$Result

f04mbf(nArg: Integer, bArg: Matrix DoubleFloat, preconArg: Boolean,_
shiftArg: DoubleFloat, itnlimArg: Integer, msglvlArg: Integer,_
lrworkArg: Integer, liworkArg: Integer, rtolArg: DoubleFloat,_
ifailArg: Integer, aprodArg: Union(fn: FileName, fp: Asp28(APROD)),_
msolveArg: Union(fn: FileName, fp: Asp34(MSOLVE))): Result ==
-- if both asps are AXIOM generated we do not need lrwork liwork
-- and will set to 1.
-- else believe the user but check that they are >0.
if (aprodArg case fn) and (msolveArg case fn)
then
  lrworkArg:=1
  liworkArg:=1
else
  lrworkArg:=max(1,lrworkArg)
  liworkArg:=max(1,liworkArg)
pushFortranOutputStack(aprodFilename := aspFilename "aprod")$FOP
if aprodArg case fn
  then outputAsFortran(aprodArg.fn)
  else outputAsFortran(aprodArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(msolveFilename := aspFilename "msolve")$FOP
if msolveArg case fn
  then outputAsFortran(msolveArg.fn)
  else outputAsFortran(msolveArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([aprodFilename,msolveFilename]$Lisp_,
"f04mbf",_
CHAPTER 15. CHAPTER N

,"rtol"::S,"n"::S,5"Lisp]"Lisp,_
["rwork"::S,"lrwork"::S]"Lisp_,
,"aprod"::S,"msolve"::S]"Lisp_,
,["integer"::S,"n"::S,"itnlim"::S,"msglvl"::S_,
,"iwork"::S,"msolve"::S]"Lisp]"Lisp_,
,["logical"::S,"precon"::S]"Lisp_]
]"Lisp_,
[(nArg::Any,preconArg::Any,shiftArg::Any,itnlimArg::Any_,
,msglvlArg::Any,lrworkArg::Any,liworkArg::Any,rtolArg::Any_,
,ifailArg::Any,bArg::Any )]
@List Any]"Lisp)"Lisp)_,
pretend List (Record(key:Symbol,entry:Any))]]Result

f04mcf(nArg:Integer,alArg:Matrix DoubleFloat,lalArg:Integer,_,
dArg:Matrix DoubleFloat,nrowArg:Matrix Integer,irArg:Integer,_,
bArg:Matrix DoubleFloat,nrbArg:Integer,iselctArg:Integer,_,
nrxArg:Integer,ifailArg:Integer): Result ==
[(invokeNagman(NIL"Lisp_,
"f04mcf",_
]"Lisp_,
["x"::S]"Lisp_,
]"Lisp_,
["x"::S,"ifail"::S]"Lisp_,
[[[nArg::Any,lalArg::Any,irArg::Any,nrbArg::Any,iselctArg::Any_,
nrxArg::Any,ifailArg::Any,alArg::Any,dArg::Any,nrowArg::Any_,
bArg::Any ]]@List Any]"Lisp)"Lisp_)_,
pretend List (Record(key:Symbol,entry:Any))]]Result

f04qaf(mArg:Integer,nArg:Integer,dampArg:DoubleFloat,_,
atolArg:DoubleFloat,btolArg:DoubleFloat,conlimArg:DoubleFloat,_,
itrnlmArg:Integer,msglvlArg:Integer,lrworkArg:Integer,_,
liworkArg:Integer,bArg:Matrix DoubleFloat,ifailArg:Integer,_,
aprodArg:Union(fn:FileName,fp:Asp30(APROD))): Result ==
pushFortranOutputStack(aprodFilename := aspFilename "aprod")$FOP
if aprodArg case fn
   then outputAsFortran(aprodArg.fn)
   else outputAsFortran(aprodArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([aprodFilename]$Lisp,_
   "f04qaf",_
   ["m":S,"n":S,"damp":S,"atol":S,"btol":S_,
   "b":S,"work":S,"rwork":S_,
   "iwork":S]$Lisp,_
   "iwork":S,"aprod":S]$Lisp,_
   [["double":S,"damp":S,"atol":S,"btol":S_,
   "conlim":S,["x":S,"n":S]$Lisp,[_"se":S,"n":S]$Lisp,_
   ["b":S,"m":S]$Lisp_
   "aprod":S]$Lisp_,
   ["integer":S,"m":S,"n":S,"msglvl":S,"msglvl":S_,
   [[MArg::Any,nArg::Any,dampArg::Any,atolArg::Any,btolArg::Any_,
   conlimArg::Any,itnlimArg::Any,msglvlArg::Any,lrworkArg::Any,_
   liworkArg::Any,ifailArg::Any,bArg::Any ]]
@List Any]$Lisp]$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

— NAGF04.dotabb —

"NAGF04" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGF04"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"NAGF04" -> "COMPCAT"

— NAGLinkSupportPackage.input —
NAGLinkSupportPackage examples

Support functions for the NAG Library Link functions

See Also:
  o )show NAGLinkSupportPackage
NAGLinkSupportPackage (NAGSP)

Exports:
aspFilename    checkPrecision    dimensionsOf fortranCompilerName
fortranLinkerArgs    restorePrecision

--- package NAGSP NAGLinkSupportPackage ---

)abbrev package NAGSP NAGLinkSupportPackage
++ Author: Mike Dewar and Godfrey Nolan
++ Date Created: March 1993
++ Date Last Updated: October 6 1994
++ Description:
++ Support functions for the NAG Library Link functions

NAGLinkSupportPackage() : exports == implementation where

exports ==> with
  fortranCompilerName : () -> String
    ++ fortranCompilerName() returns the name of the currently selected
    ++ Fortran compiler
  fortranLinkerArgs : () -> String
    ++ fortranLinkerArgs() returns the current linker arguments
  aspFilename : String -> String
    ++ aspFilename("f") returns a String consisting of "f" suffixed with
    ++ an extension identifying the current AXIOM session.
  dimensionsOf : (Symbol, Matrix DoubleFloat) -> SExpression
    ++ dimensionsOf(s,m) \ undocumented()
  dimensionsOf : (Symbol, Matrix Integer) -> SExpression
    ++ dimensionsOf(s,m) \ undocumented()
  checkPrecision : () -> Boolean
    ++ checkPrecision() \ undocumented()
  restorePrecision : () -> Void
    ++ restorePrecision() \ undocumented()

implementation ==> add
  makeAs: (Symbol,Symbol) -> Symbol
changeVariables: (Expression Integer, Symbol) -> Expression Integer
changeVariablesF: (Expression Float, Symbol) -> Expression Float

import String
import Symbol

checkPrecision(): Boolean ==
  (_$fortranPrecision$Lisp = "single": Symbol) and _
  (_$nagEnforceDouble$Lisp) =>
    systemCommand("set fortran precision double")$MoreSystemCommands
    if _$nagMessages$Lisp then
      print("*** Warning: Resetting fortran precision to double")_ $PrintPackage
      true
    false

restorePrecision(): Void ==
  systemCommand("set fortran precision single")$MoreSystemCommands
  if _$nagMessages$Lisp then
    print("** Warning: Restoring fortran precision to single")$PrintPackage
    void()$Void

uniqueId : String := ""
counter : Integer := 0
getUniqueId(): String ==
  if uniqueId = "" then
    uniqueId := concat(getEnv("HOST")$Lisp, getEnv("SPADNUM")$Lisp)
    concat(uniqueId, string (counter:=counter+1))

fortranCompilerName() == string _$fortranCompilerName$Lisp
fortranLinkerArgs() == string _$fortranLibraries$Lisp

aspFilename(f: String): String == concat ["/tmp/", f, getUniqueId(), ".f"]

dimensionsOf(u: Symbol, m: Matrix DoubleFloat): SExpression ==
  [u, nrows m, ncols m]$Lisp
dimensionsOf(u: Symbol, m: Matrix Integer): SExpression ==
  [u, nrows m, ncols m]$Lisp

——

— NAGSP.dotabb —

"NAGSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGSP"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGSP" -> "ALIST"
package NAGD01 NagIntegrationPackage

-- NagIntegrationPackage.input --

)set break resume
)sys rm -f NagIntegrationPackage.output
)spool NagIntegrationPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show NagIntegrationPackage
--R
--R NagIntegrationPackage is a package constructor
--R Abbreviation for NagIntegrationPackage is NAGD01
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGD01
--R
--R------------------------------- Operations --------------------------------
--R d01ajf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(F))) -> Result
--R d01akf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(F))) -> Result
--R d01alf : (DoubleFloat,DoubleFloat,Integer,Matrix(DoubleFloat),DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(G))) -> Result
--R d01amf : (DoubleFloat,Integer,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(G))) -> Result
--R d01anf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(G))) -> Result
--R d01apf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(G))) -> Result
--R d01aqf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(G))) -> Result
--R d01asf : (DoubleFloat,DoubleFloat,Integer,DoubleFloat,Integer,Integer,Integer,Integer,Union(fn: FileName,fp: Asp1(G))) -> Result
--R d01bbf : (DoubleFloat,DoubleFloat,Integer,Integer,Integer,Integer) -> Result
--R d01fcf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp4(FUNCTN))) -> Result
--R d01gaf : (Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer) -> Result
--R d01gbf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,DoubleFloat,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp4(FUNCTN))) -> Result
--R
--E 1

)spool
)lisp (bye)

---

-- NagIntegrationPackage.help --

This package uses the NAG Library to calculate the numerical value of definite integrals in one or more dimensions and to evaluate weights and abscissae of integration rules.

D01(3NAG) Foundation Library (12/10/92) D01(3NAG)
Chapter D01

Quadrature

1. Scope of the Chapter

This chapter provides routines for the numerical evaluation of definite integrals in one or more dimensions and for evaluating weights and abscissae of integration rules.

2. Background to the Problems

The routines in this chapter are designed to estimate:

(a) the value of a one-dimensional definite integral of the form:

\[
\int_a^b f(x) \, dx
\]  

(1)

where \( f(x) \) is defined by the user, either at a set of points \( (x_i, f(x_i)) \), for \( i=1,2,...,n \) where \( a=x_1<x_2<...<x_n=b \), or in the form of a function; and the limits of integration \( a,b \) may be finite or infinite.

Some methods are specially designed for integrands of the form

\[
f(x) = w(x)g(x)
\]  

(2)

which contain a factor \( w(x) \), called the weight-function, of a specific form. These methods take full account of any peculiar behaviour attributable to the \( w(x) \) factor.

(b) the value of a multi-dimensional definite integral of the form:

\[
\int_{R_n} f(x_1,x_2,...,x_n) \, dx_1 \, dx_2 \cdots dx_n
\]  

(3)

where \( f(x_1,x_2,...,x_n) \) is a function defined by the user and \( R \) is some region of \( n \)-dimensional space.

The simplest form of \( R \) is the \( n \)-rectangle defined by:

\[
a \leq x_i \leq b, \quad i=1,2,...,n
\]  

(4)
where \( a \) and \( b \) are constants. When \( a \) and \( b \) are functions of \( x \) \((j<i)\), the region can easily be transformed to the rectangular form (see Davis and Rabinowitz [1] page 266). Some of the methods described incorporate the transformation procedure.

2.1. One-dimensional Integrals

To estimate the value of a one-dimensional integral, a quadrature rule uses an approximation in the form of a weighted sum of integrand values, i.e.,

\[
\int_a^b f(x) \, dx \approx \sum_{i=1}^N w_i f(x_i).
\]  

(5)

The points \( x \) within the interval \([a,b]\) are known as the abscissae, and the \( w \) are known as the weights.

More generally, if the integrand has the form (2), the corresponding formula is

\[
\int_a^b w(x)g(x) \, dx \approx \sum_{i=1}^N w_i g(x_i).
\]  

(6)

If the integrand is known only at a fixed set of points, these points must be used as the abscissae, and the weighted sum is calculated using finite-difference methods. However, if the functional form of the integrand is known, so that its value at any abscissa is easily obtained, then a wide variety of quadrature rules are available, each characterised by its choice of abscissae and the corresponding weights.

The appropriate rule to use will depend on the interval \([a,b]\) - whether finite or otherwise - and on the form of any \( w(x) \) factor in the integrand. A suitable value of \( N \) depends on the general behaviour of \( f(x) \); or of \( g(x) \), if there is a \( w(x) \) factor present.

Among possible rules, we mention particularly the Gaussian formulae, which employ a distribution of abscissae which is
optimal for \( f(x) \) or \( g(x) \) of polynomial form.

The choice of basic rules constitutes one of the principles on which methods for one-dimensional integrals may be classified. The other major basis of classification is the implementation strategy, of which some types are now presented.

(a) Single rule evaluation procedures

A fixed number of abscissae, \( N \), is used. This number and the particular rule chosen uniquely determine the weights and abscissae. No estimate is made of the accuracy of the result.

(b) Automatic procedures

The number of abscissae, \( N \), within \([a,b]\) is gradually increased until consistency is achieved to within a level of accuracy (absolute or relative) requested by the user. There are essentially two ways of doing this; hybrid forms of these two methods are also possible:

(i) whole interval procedures (non-adaptive)

A series of rules using increasing values of \( N \) are successively applied over the whole interval \([a,b]\). It is clearly more economical if abscissae already used for a lower value of \( N \) can be used again as part of a higher-order formula. This principle is known as optimal extension. There is no overlap between the abscissae used in Gaussian formulae of different orders. However, the Kronrod formulae are designed to give an optimal \((2N+1)\)-point formula by adding \((N+1)\) points to an \(N\)-point Gauss formula. Further extensions have been developed by Patterson.

(ii) adaptive procedures

The interval \([a,b]\) is repeatedly divided into a number of sub-intervals, and integration rules are applied separately to each sub-interval. Typically, the subdivision process will be carried further in the neighbourhood of a sharp peak in the integrand, than where the curve is smooth. Thus, the distribution of abscissae is adapted to the shape of the integrand.

Subdivision raises the problem of what constitutes an acceptable accuracy in each sub-interval. The usual global acceptability criterion demands that the sum of the absolute values of the error estimates in the
sub-intervals should meet the conditions required of the error over the whole interval. Automatic extrapolation over several levels of subdivision may eliminate the effects of some types of singularities.

An ideal general-purpose method would be an automatic method which could be used for a wide variety of integrands, was efficient (i.e., required the use of as few abscissae as possible), and was reliable (i.e., always gave results within the requested accuracy). Complete reliability is unobtainable, and generally higher reliability is obtained at the expense of efficiency, and vice versa. It must therefore be emphasised that the automatic routines in this chapter cannot be assumed to be 100% reliable. In general, however, the reliability is very high.

2.2. Multi-dimensional Integrals

A distinction must be made between cases of moderately low dimensionality (say, up to 4 or 5 dimensions), and those of higher dimensionality. Where the number of dimensions is limited, a one-dimensional method may be applied to each dimension, according to some suitable strategy, and high accuracy may be obtainable (using product rules). However, the number of integrand evaluations rises very rapidly with the number of dimensions, so that the accuracy obtainable with an acceptable amount of computational labour is limited; for example a product of 3-point rules in 20 dimensions would require more than 10^9 integrand evaluations. Special techniques such as the Monte Carlo methods can be used to deal with high dimensions.

(a) Products of one-dimensional rules

Using a two-dimensional integral as an example, we have

$$\int_{a}^{b} \int_{a}^{b} f(x,y) \, dy \, dx = \sum_{i=1}^{N} \sum_{j=1}^{N} w_{i} \int_{a}^{b} f(x,y) \, dy$$  \hspace{1cm} (7)

and

$$\int_{a}^{b} \int_{a}^{b} f(x,y) \, dy \, dx = \sum_{i=1}^{N} \sum_{j=1}^{N} v_{i} v_{j} f(x,y)$$  \hspace{1cm} (8)

where \((w_{i},x_{i})\) and \((v_{j},y_{j})\) are the weights and abscissae of
the rules used in the respective dimensions.

A different one-dimensional rule may be used for each dimension, as appropriate to the range and any weight function present, and a different strategy may be used, as appropriate to the integrand behaviour as a function of each independent variable.

For a rule-evaluation strategy in all dimensions, the formula (8) is applied in a straightforward manner. For automatic strategies (i.e., attempting to attain a requested accuracy), there is a problem in deciding what accuracy must be requested in the inner integral(s). Reference to formula (7) shows that the presence of a limited but random error in the y-integration for different values of x can produce a 'jagged' function of x, which may be difficult to integrate to the desired accuracy and for this reason products of automatic one-dimensional routines should be used with caution (see also Lyness [3]).

(b) Monte Carlo methods

These are based on estimating the mean value of the integrand sampled at points chosen from an appropriate statistical distribution function. Usually a variance reducing procedure is incorporated to combat the fundamentally slow rate of convergence of the rudimentary form of the technique. These methods can be effective by comparison with alternative methods when the integrand contains singularities or is erratic in some way, but they are of quite limited accuracy.

(c) Number theoretic methods

These are based on the work of Korobov and Conroy and operate by exploiting implicitly the properties of the Fourier expansion of the integrand. Special rules, constructed from so-called optimal coefficients, give a particularly uniform distribution of the points throughout n-dimensional space and from their number theoretic properties minimize the error on a prescribed class of integrals. The method can be combined with the Monte Carlo procedure.

(d) Sag-Szekeres method

By transformation this method seeks to induce properties into the integrand which make it accurately integrable by
the trapezoidal rule. The transformation also allows effective control over the number of integrand evaluations.

(e) Automatic adaptive procedures

An automatic adaptive strategy in several dimensions normally involves division of the region into subregions, concentrating the divisions in those parts of the region where the integrand is worst behaved. It is difficult to arrange with any generality for variable limits in the inner integral(s). For this reason, some methods use a region where all the limits are constants; this is called a hyper-rectangle. Integrals over regions defined by variable or infinite limits may be handled by transformation to a hyper-rectangle. Integrals over regions so irregular that such a transformation is not feasible may be handled by surrounding the region by an appropriate hyper-rectangle and defining the integrand to be zero outside the desired region. Such a technique should always be followed by a Monte Carlo method for integration.

The method used locally in each subregion produced by the adaptive subdivision process is usually one of three types: Monte Carlo, number theoretic or deterministic. Deterministic methods are usually the most rapidly convergent but are often expensive to use for high dimensionality and not as robust as the other techniques.

2.3. References

Comprehensive reference:


Special topics:


3. Recommendations on Choice and Use of Routines

The following three sub-sections consider in turn routines for: one-dimensional integrals over a finite interval, and over a semi-infinite or an infinite interval; and multi-dimensional integrals. Within each sub-section, routines are classified by the type of method, which ranges from simple rule evaluation to automatic adaptive algorithms. The recommendations apply particularly when the primary objective is simply to compute the value of one or more integrals, and in these cases the automatic adaptive routines are generally the most convenient and reliable, although also the most expensive in computing time.

Note however that in some circumstances it may be counter-productive to use an automatic routine. If the results of the quadrature are to be used in turn as input to a further computation (e.g. an 'outer' quadrature or an optimization problem), then this further computation may be adversely affected by the 'jagged performance profile' of an automatic routine; a simple rule-evaluation routine may provide much better overall performance. For further guidance, the article Lyness [3] is recommended.

3.1. One-dimensional Integrals over a Finite Interval

(a) Integrand defined as a set of points

If \( f(x) \) is defined numerically at four or more points, then the Gill-Miller finite difference method (D01GAF) should be used. The interval of integration is taken to coincide with the range of \( x \)-values of the points supplied. It is in the nature of this problem that any routine may be unreliable. In order to check results independently and so as to provide an alternative technique the user may fit the integrand by Chebyshev series using E02ADF and then use routines E02AJF and E02AKF to evaluate its integral (which need not be restricted to the range of the integration points, as is the case for D01GAF). A further alternative is to fit a cubic spline to the data using E02BAF and then to evaluate its integral using E02BDF.

(b) Integrand defined as a function

If the functional form of \( f(x) \) is known, then one of the following approaches should be taken. They are arranged in the order from most specific to most general, hence the first applicable procedure in the list will be the most
efficient. However, if the user does not wish to make any assumptions about the integrand, the most reliable routine to use will be D01AJF, although this will in general be less efficient for simple integrals.

(i) Rule-evaluation routines

If \( f(x) \) is known to be sufficiently well-behaved (more precisely, can be closely approximated by a polynomial of moderate degree), a Gaussian routine with a suitable number of abscissae may be used.

D01BBF may be used if it is required to examine the weights and abscissae. In this case, the user should write the code for the evaluation of quadrature summation (6).

(ii) Automatic adaptive routines

Firstly, several routines are available for integrands of the form \( w(x)g(x) \) where \( g(x) \) is a 'smooth' function (i.e., has no singularities, sharp peaks or violent oscillations in the interval of integration) and \( w(x) \) is a weight function of one of the following forms:

\[
\begin{align*}
\text{if } w(x) &= (b-x)^{\alpha} (x-a)^{\beta} \left(\log(b-x)\right)^{k} \left(\log(x-a)\right)^{l}, \\
& \quad \text{where } k,l=0 \text{ or } 1, \alpha,\beta>-1: \text{use D01APF}; \\
\text{if } w(x) &= 1/(x-c): \text{use D01AQF (this integral is called the Hilbert transform of } g); \\
\text{if } w(x) &= \cos(\omega x) \text{ or } \sin(\omega x): \text{use D01ANF (this routine can also handle certain types of singularities in } g(x)).
\end{align*}
\]

Secondly, there are some routines for general \( f(x) \). If \( f(x) \) is known to be free of singularities, though it may be oscillatory, D01AKF may be used.

The most powerful of the finite interval integration routine is D01AJF (which can cope with singularities of several types). It may be used if none of the more specific situations described above applies. D01AJF is very reliable, particularly where the integrand has singularities other than at an end-point, or has discontinuities or cusps, and is therefore recommended where the integrand is known to be badly-behaved, or where its nature is completely unknown.
Most of the routines in this chapter require the user to supply a function or subroutine to evaluate the integrand at a single point.

If \( f(x) \) has singularities of certain types, discontinuities or sharp peaks occurring at known points, the integral should be evaluated separately over each of the subranges or D01ALF may be used.

3.2. One-dimensional Integrals over a Semi-infinite or Infinite Interval

(a) Integrand defined as a set of points

If \( f(x) \) is defined numerically at four or more points, and the portion of the integral lying outside the range of the points supplied may be neglected, then the Gill-Miller finite difference method, D01GAF, should be used.

(b) Integrand defined as a function

(i) Rule evaluation routines

If \( f(x) \) behaves approximately like a polynomial in \( x \), apart from a weight function of the form

\[-(\beta)x \quad \text{e}^{\beta} > 0 \text{ (semi-infinite interval, lower limit finite);}
\]

\[-(\beta)x \quad \text{e}^{\beta} < 0 \text{ (semi-infinite interval, upper limit finite);}
\]

\[2 \quad -(\beta)(x-(\alpha)) \quad \text{e}^{\beta} > 0 \text{ (infinite interval);}
\]

or if \( f(x) \) behaves approximately like a

\[-1 \quad \text{polynomial in } (x+B) \text{ (semi-infinite range);}
\]

then the Gaussian routines may be used.

D01BBF may be used if it is required to examine the weights and abscissae. In this case, the user should write the code for the evaluation of quadrature summation (6).

(ii) Automatic adaptive routines

D01AMF may be used, except for integrands which decay
slowly towards an infinite end-point, and oscillate in sign over the entire range. For this class, it may be possible to calculate the integral by integrating between the zeros and invoking some extrapolation process.

D01ASF may be used for integrals involving weight functions of the form cos((omega)x) and sin((omega)x) over a semi-infinite interval (lower limit finite).

The following alternative procedures are mentioned for completeness, though their use will rarely be necessary:

1. If the integrand decays rapidly towards an infinite end-point, a finite cut-off may be chosen, and the finite range methods applied.

2. If the only irregularities occur in the finite part (apart from a singularity at the finite limit, with which D01AMF can cope), the range may be divided, with D01AMF used on the infinite part.

3. A transformation to finite range may be employed, e.g.

\[
\int_{0}^{\infty} f(x) \, dx = \int_{0}^{1} [f(x) + f(-x)] \, dx.
\]

This saves computing unnecessary function values in the semi-infinite range where the function is well behaved.

3.3. Multi-dimensional Integrals

A number of techniques are available in this area and the choice
depends to a large extent on the dimension and the required accuracy. It can be advantageous to use more than one technique as a confirmation of accuracy particularly for high dimensional integrations. Many of the routines incorporate the transformation procedure REGION which allows general product regions to be easily dealt with in terms of conversion to the standard n-cube region.

(a) Products of one-dimensional rules (suitable for up to about 5 dimensions)

If \( f(x_1, x_2, \ldots, x_n) \) is known to be a sufficiently well-behaved function of each variable \( x_i \) apart possibly from weight functions of the types provided, a product of Gaussian rules may be used. These are provided by D01BBF. In this case, the user should write the code for the evaluation of quadrature summation (6). Rules for finite, semi-infinite and infinite ranges are included.

The one-dimensional routines may also be used recursively. For example, the two-dimensional integral

\[
\int_a^b \int_a^b f(x,y) \, dy \, dx
\]

may be expressed as

\[
\int_a^b F(x) \, dx,
\]

where

\[
F(x) = \int_a^2 f(x,y) \, dy.
\]

The user segment to evaluate \( F(x) \) will call the integration routine for the \( y \)-integration, which will call another user segment for \( f(x,y) \) as a function of \( y \) (\( x \) being effectively a constant). Note that, as Fortran is not a recursive
language, a different library integration routine must be used for each dimension. Apart from this restriction, the full range of one-dimensional routines are available, for finite/infinite intervals, constant/variable limits, rule evaluation/automatic strategies etc.

(b) Automatic routines (D01GBF and D01FCF)

Both routines are for integrals of the form

\[
\int_a^b f(x_{1}, x_{2}, \ldots, x_{n}) dx_{1} \ldots dx_{n-1} \]

D01GBF is an adaptive Monte Carlo routine. This routine is usually slow and not recommended for high accuracy work. It is a robust routine that can often be used for low accuracy results with highly irregular integrands or when n is large.

D01FCF is an adaptive deterministic routine. Convergence is fast for well-behaved integrands. Highly accurate results can often be obtained for n between 2 and 5, using significantly fewer integrand evaluations than would be required by D01GBF. The routine will usually work when the integrand is mildly singular and for n\leq10 should be used before D01GBF. If it is known in advance that the integrand is highly irregular, it is best to compare results from at least two different routines.

There are many problems for which one or both of the routines will require large amounts of computing time to obtain even moderately accurate results. The amount of computing time is controlled by the number of integrand evaluations allowed by the user, and users should set this parameter carefully, with reference to the time available and the accuracy desired.

3.4. Decision Trees

(i) One-dimensional integrals over a finite interval. (If in doubt, follow the downward branch.)

Please see figure in printed Reference Manual
(ii) One-dimensional integrals over a semi-infinite or infinite interval. (If in doubt, follow the downward branch.)

Please see figure in printed Reference Manual

D01 -- Quadrature

Chapter D01

Quadrature

D01AJF 1-D quadrature, adaptive, finite interval, strategy due to Piessens and de Doncker, allowing for badly-behaved integrands

D01AKF 1-D quadrature, adaptive, finite interval, method suitable for oscillating functions

D01ALF 1-D quadrature, adaptive, finite interval, allowing for singularities at user-specified break-points

D01AMF 1-D quadrature, adaptive, infinite or semi-infinite interval

D01ANF 1-D quadrature, adaptive, finite interval, weight function \( \cos(\omega x) \) or \( \sin(\omega x) \)

D01APF 1-D quadrature, adaptive, finite interval, weight function with end-point singularities of algebraic-logarithmic type

D01AQF 1-D quadrature, adaptive, finite interval, weight function \( 1/(x-c) \), Cauchy principal value (Hilbert transform)

D01ASF 1-D quadrature, adaptive, semi-infinite interval, weight function \( \cos(\omega x) \) or \( \sin(\omega x) \)

D01BBF Weights and abscissae for Gaussian quadrature rules

D01FCF Multi-dimensional adaptive quadrature over hyper-rectangle

D01GAF 1-D quadrature, integration of function defined by data values, Gill-Miller method

D01GBF Multi-dimensional quadrature over hyper-rectangle, Monte...
Carlo method

D01A

D01A

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01A

is a general-purpose integrator which calculates an approximation to the integral of a function f(x) over a finite interval [a, b]:

b
I= \int f(x)dx.
a

2. Specification

SUBROUTINE D01A (F, A, B, EPSABS, EPSREL, RESULT, ABSE

EXTERNAL F

3. Description

D01A

is based upon the QUADPACK routine QAGS (Piessens et al [3]). It is an adaptive routine, using the Gauss 10-point and Kronrod 21-point rules. The algorithm, described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The local error estimation is described by Piessens et al [3].

The routine is suitable as a general purpose integrator, and can be used when the integrand has singularities, especially when these are of algebraic or logarithmic type.

D01A

requires the user to supply a function to evaluate the
integrand at a single point.

The routine D01ATF(*) uses an identical algorithm but requires the user to supply a subroutine to evaluate the integrand at an array of points. Therefore D01ATF(*) will be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user.
   External Procedure
   F must return the value of the integrand \( f \) at a given point.

   Its specification is:

   DOUBLE PRECISION FUNCTION F (X)
   DOUBLE PRECISION X

1: X -- DOUBLE PRECISION
   Input
   On entry: the point at which the integrand \( f \) must be evaluated.

   F must be declared as EXTERNAL in the (sub)program from
   which D01AJF is called. Parameters denoted as Input
   must not be changed by this procedure.

2: A -- DOUBLE PRECISION
   Input
   On entry: the lower limit of integration, \( a \).

3: B -- DOUBLE PRECISION
   Input
   On entry: the upper limit of integration, \( b \). It is not necessary that \( a < b \).
4: EPSABS -- DOUBLE PRECISION Input
   On entry: the absolute accuracy required. If EPSABS is negative, the absolute value is used. See Section 7.

5: EPSREL -- DOUBLE PRECISION Input
   On entry: the relative accuracy required. If EPSREL is negative, the absolute value is used. See Section 7.

6: RESULT -- DOUBLE PRECISION Output
   On exit: the approximation to the integral I.

7: ABSERR -- DOUBLE PRECISION Output
   On exit: an estimate of the modulus of the absolute error, which should be an upper bound for |I-RESULT|.

8: W(LW) -- DOUBLE PRECISION array Output
   On exit: details of the computation, as described in Section 8.

9: LW -- INTEGER Input
   On entry:
   the dimension of the array W as declared in the (sub)program from which D01AJF is called.
   The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems.
   Constraint: LW >= 4.

10: IW(LIW) -- INTEGER array Output
    On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

11: LIW -- INTEGER Input
    On entry:
    the dimension of the array IW as declared in the (sub)program from which D01AJF is called.
    The number of sub-intervals into which the interval of integration may be divided cannot exceed LIW. Suggested value: LIW = LW/4. Constraint: LIW >= 1.

12: IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).
For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) you will probably gain from splitting up the interval at this point and calling the integrator on the subranges. If necessary, another integrator, which is designed for handling the type of difficulty involved, must be used. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. The error may be under-estimated. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval.

IFAIL= 4
The requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily; the returned result is the best which can be obtained. The same advice applies as in the case of IFAIL = 1.

IFAIL= 5
The integral is probably divergent, or slowly convergent. Please note that divergence can occur with any non-zero value of IFAIL.

IFAIL= 6
On entry LW < 4,
7. Accuracy

The routine cannot guarantee, but in practice usually achieves, the following accuracy:

$$|I-\text{RESULT}| \leq \text{tol},$$

where

$$\text{tol} = \max\{\text{EPSABS}, |\text{EPSREL}| \cdot |I|\},$$

and EPSABS and EPSREL are user-specified absolute and relative error tolerance. Moreover it returns the quantity ABSERR which, in normal circumstances, satisfies

$$|I-\text{RESULT}| \leq \text{ABSERR} \leq \text{tol}.$$ 

8. Further Comments

The time taken by the routine depends on the integrand and the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AJF along with the integral contributions and error estimates over the sub-intervals.

Specifically, for i=1,2,...,n, let r denote the approximation to the value of the integral over the sub-interval \([a_i, b_i]\) in the partition of \([a, b]\) and e be the corresponding absolute error estimate.

\[
\int_{a_i}^{b_i} f(x) \, dx \approx r_i
\]

Then, \( \int_{a}^{b} f(x) \, dx \geq r \), unless D01AJF terminates while testing for divergence of the integral (see Piessens et al [3], Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of \( n \) is returned in IW(1), and the values \( a_i, b_i, e_i \) and \( r_i \)
are stored consecutively in the array W, that is:

\[
\begin{align*}
    a &= W(i), \\
    b &= W(n+i), \\
    e &= W(2n+i), \text{ and} \\
    r &= W(3n+i).
\end{align*}
\]

9. Example

To compute

\[
\frac{2(\pi)}{x \sin(30x)} \left|_{0}^{\sqrt{1 - \left(\frac{x}{2(\pi)}\right)^2}} \right.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
\[ I = \int_a^b f(x) \, dx. \]

2. Specification

```fortran
SUBROUTINE D01AKF (F, A, B, EPSABS, EPSREL, RESULT, \
1                            ABSERR, W, LW, IW, LIW, IFAIL)
INTEGER LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION F, A, B, EPSABS, EPSREL, RESULT, ABSERR, W
1 (LW)
EXTERNAL F
```

3. Description

D01AKF is based upon the QUADPACK routine QAG (Piessens et al [3]). It is an adaptive routine, using the Gauss 30-point and Kronrod 61-point rules. A 'global' acceptance criterion (as defined by Malcolm and Simpson [1]) is used. The local error estimation is described in Piessens et al [3].

Because this routine is based on integration rules of high order, it is especially suitable for non-singular oscillating integrands.

D01AKF requires the user to supply a function to evaluate the integrand at a single point.

The routine D01AUF(*) uses an identical algorithm but requires the user to supply a subroutine to evaluate the integrand at an array of points. Therefore D01AUF(*) will be more efficient if the evaluation can be performed in vector mode on a vector-processing machine.

D01AUF(*) also has an additional parameter KEY which allows the user to select from six different Gauss-Kronrod rules.

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user.  
   External Procedure  
   F must return the value of the integrand f at a given point.  
   Its specification is:  
   
   \[
   \text{DOUBLE PRECISION FUNCTION} \quad F (X)  
   \]
   \[
   \text{DOUBLE PRECISION} \quad X  
   \]

1: X -- DOUBLE PRECISION  
   On entry: the point at which the integrand f must be  
   evaluated.  
   F must be declared as EXTERNAL in the (sub)program from  
   which D01AKF is called. Parameters denoted as Input  
   must not be changed by this procedure.  

2: A -- DOUBLE PRECISION  
   On entry: the lower limit of integration, a.  

3: B -- DOUBLE PRECISION  
   On entry: the upper limit of integration, b. It is not  
   necessary that a<b.  

4: EPSABS -- DOUBLE PRECISION  
   On entry: the absolute accuracy required. If EPSABS is  
   negative, the absolute value is used. See Section 7.  

5: EPSREL -- DOUBLE PRECISION  
   On entry: the relative accuracy required. If EPSREL is  
   negative, the absolute value is used. See Section 7.  

6: RESULT -- DOUBLE PRECISION  
   On exit: the approximation to the integral I.  

7: ABSERR -- DOUBLE PRECISION  
   On exit: an estimate of the modulus of the absolute error,  
   which should be an upper bound |I-RESULT|.  

8: W(LW) -- DOUBLE PRECISION array  
   On exit: details of the computation, as described in  
   Section 8.  

9: LW -- INTEGER  
   On entry: the dimension of W, as declared in the (sub)  
   program from which D01AKF is called. The value of LW  
   (together with that of LIW below) imposes a bound on the  
   number of sub-intervals into which the interval of  
   integration may be divided by the routine. The number of
sub-intervals cannot exceed LW/4. The more difficult the
integrand, the larger LW should be. Suggested value: a value
in the range 800 to 2000 is adequate for most problems.
Constraint: LW >= 4. See IW below.

10: IW(LIW) -- INTEGER array Output
On exit: IW(1) contains the actual number of sub-intervals
used. The rest of the array is used as workspace.

11: LIW -- INTEGER Input
On entry:
the dimension of the array IW as declared in the
(sub)program from which D01AKF is called.
The number of sub-intervals into which the interval of
integration may be divided cannot exceed LIW. Suggested

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given
workspace has been reached without the accuracy requirements
being achieved. Look at the integrand in order to determine
the integration difficulties. Probably another integrator
which is designed for handling the type of difficulty
involved must be used. Alternatively, consider relaxing the
accuracy requirements specified by EPSABS and EPSREL, or
increasing the amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being
achieved. Consider requesting less accuracy.
CHAPTER 15. CHAPTER N

IFAIL = 3
Extremely bad local integrand behaviour causes a very strong
subdivision around one (or more) points of the interval.
The same advice applies as in the case of IFAIL = 1.

IFAIL = 4
On entry LW < 4,
or LIW < 1.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

\[ |I - RESULT| \leq \text{tol}, \]

where

\[ \text{tol} = \max\{|\text{EPSABS}|, |\text{EPSREL}||I|\}, \]

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR which,
in normal circumstances satisfies

\[ |I - RESULT| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments

The time taken by the routine depends on the integrand and the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01AKF along with the integral
contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let \( r_i \) denote the approximation to
the value of the integral over the sub-interval \([a_i, b_i]\) in the
partition of \([a, b]\) and \( e_i \) be the corresponding absolute error
\( i \)

\[
\begin{align*}
|f(x)dx| &= r_i \\
\text{RESULT} &= r_i
\end{align*}
\]

estimate. Then, \( |f(x)dx| = r_i \) and RESULT> \( r_i \). The value of n

\[
\begin{align*}
\text{RESULT} &= \sum_{i=1}^{n} r_i \\
\end{align*}
\]
is returned in IW(i), and the values a, b, e and r are stored consecutively in the array W, that is:

\[ a = W(i), \]
\[ b = W(n+i), \]
\[ e = W(2n+i) \text{ and} \]
\[ r = W(3n+i). \]

9. Example

To compute

\[
\frac{2}{\pi} \int_{0}^{\frac{\pi}{2}} x \sin(30x) \cos x \, dx.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
\[ I = \int_a^b f(x) \, dx \]

where the integrand may have local singular behaviour at a finite number of points within the integration interval.

2. Specification

```fortran
SUBROUTINE D01ALF (F, A, B, NPTS, POINTS, EPSABS, EPSREL, RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
  INTEGER NPTS, LW, IW(LIW), LIW, IFAIL
  DOUBLE PRECISION F, A, B, POINTS(*), EPSABS, EPSREL
  EXTERNAL F
  RESULT, ABSERR, W(LW)
END
```

3. Description

D01ALF is based upon the QUADPACK routine QAGP (Piessens et al [3]). It is very similar to D01AJF, but allows the user to supply difficult. It is an adaptive routine, using the Gauss 10-point and Kronrod 21-point rules. The algorithm described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The user-supplied 'break-points' always occur as the end-points of some sub-interval during the adaptive process. The local error estimation is described by Piessens et al [3].

4. References


      m n

5. Parameters

1. F -- DOUBLE PRECISION FUNCTION, supplied by the user.
External Procedure
F must return the value of the integrand f at a given point.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION } F (X) \\
\text{DOUBLE PRECISION } X \\
\]

1: \text{X -- DOUBLE PRECISION} \hspace{1cm} \text{Input} \\
On entry: the point at which the integrand f must be 
evaluated.
F must be declared as EXTERNAL in the (sub)program from 
which D01ALF is called. Parameters denoted as Input 
must not be changed by this procedure.

2: \text{A -- DOUBLE PRECISION} \hspace{1cm} \text{Input} \\
On entry: the lower limit of integration, a.

3: \text{B -- DOUBLE PRECISION} \hspace{1cm} \text{Input} \\
On entry: the upper limit of integration, b. It is not 
necessary that a<b.

4: \text{NPTS -- INTEGER} \hspace{1cm} \text{Input} \\
On entry: the number of user-supplied break-points within 
the integration interval. Constraint: NPTS ≥ 0.

5: \text{POINTS(NPTS) -- DOUBLE PRECISION array} \hspace{1cm} \text{Input} \\
On entry: the user-specified break-points. Constraint: the 
break-points must all lie within the interval of integration 
(but may be supplied in any order).

6: \text{EPSABS -- DOUBLE PRECISION} \hspace{1cm} \text{Input} \\
On entry: the absolute accuracy required. If EPSABS is 
negative, the absolute value is used. See Section 7.

7: \text{EPSREL -- DOUBLE PRECISION} \hspace{1cm} \text{Input} \\
On entry: the relative accuracy required. If EPSREL is 
negative, the absolute value is used. See Section 7.

8: \text{RESULT -- DOUBLE PRECISION} \hspace{1cm} \text{Input} \\
On entry: the approximation to the integral I.

9: \text{ABERR -- DOUBLE PRECISION} \hspace{1cm} \text{Output} \\
On exit: an estimate of the modulus of the absolute error, 
which should be an upper bound for |I-RESULT|.

10: \text{W(LW) -- DOUBLE PRECISION array} \hspace{1cm} \text{Output} \\
On exit: details of the computation, as described in 
Section 8.
11: LW -- INTEGER Input
On entry:
the dimension of the array W as declared in the (sub)program from which D01ALF is called.
The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed \((\text{LW}-2\times\text{NPTS}-4)/4\). The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems. Constraint: \(\text{LW}\geq 2\times\text{NPTS}+8\).

12: IW(LIW) -- INTEGER array Output
On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

13: LIW -- INTEGER Input
On entry:
the dimension of the array IW as declared in the (sub)program from which D01ALF is called.
The number of sub-intervals into which the interval of integration may be divided cannot exceed \((\text{LIW}-\text{NPTS}-2)/2\). Suggested value: \(\text{LIW} = \text{LW}/2\). Constraint: \(\text{LIW} \geq \text{NPTS} + 4\).

14: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached, without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined
(e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) it should be supplied to the routine as an element of the vector POINTS. If necessary, another integrator should be used, which is designed for handling the type of difficulty involved. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL = 2
Round-off error prevents the requested tolerance from being achieved. The error may be under-estimated. Consider requesting less accuracy.

IFAIL = 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL = 4
The requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily; the result returned is the best which can be obtained. The same advice applies as in the case IFAIL = 1.

IFAIL = 5
The integral is probably divergent, or slowly convergent. Please note that divergence can also occur with any other non-zero value of IFAIL.

IFAIL = 6
The input is invalid: break-points are specified outside the integration range, NPTS > LIMIT or NPTS < 0. RESULT and ABSERR are set to zero.

IFAIL = 7
On entry LW<2*NPTS+8,
or LIW<NPTS+4.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I-RESULT| \leq to1, \]

where

\[ to1 = \max\{ |EPSABS|, |EPSREL| \times |I| \}, \]

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR which, in normal circumstances, satisfies

\[ |I - \text{RESULT}| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments

The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01ALF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for \( i=1,2,...,n \), let \( r \) denote the approximation to the value of the integral over the sub-interval \([a_i, b_i]\) in the partition of \([a, b]\) and \( e \) be the corresponding absolute error estimate. Then,

\[ \int_{a_i}^{b_i} f(x) \, dx \approx r \] and \( \text{RESULT} = r \) unless D01ALF terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, \( \text{RESULT} \) (and \( \text{ABSERR} \)) are taken to be the values returned from the extrapolation process. The value of \( n \) is returned in IW(1), and the values \( a_i, b_i, e_i \) and \( r_i \) are stored consecutively in the array \( W \), that is:

\[ a_i = W(i), \]

\[ b_i = W(n+i), \]

\[ e_i = W(2n+i) \] and

\[ r_i = W(3n+i). \]

9. Example
To compute

\[
\int_{0}^{\infty} \frac{1}{|x-1/7|} \, dx.
\]

A break-point is specified at \( x=1/7 \), at which point the integrand is infinite. (For definiteness the function FST returns the value 0.0 at this point.)

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**1. Purpose**

D01AMF calculates an approximation to the integral of a function \( f(x) \) over an infinite or semi-infinite interval \([a,b]\):

\[
\int_{a}^{b} |f(x)| \, dx
\]

**2. Specification**

```fortran
SUBROUTINE D01AMF (F, BOUND, INF, EPSABS, EPSREL, RESULT, 1 ABSERR, W, LW, IW, LIW, IFAIL)
INTEGER INF, LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION F, BOUND, EPSABS, EPSREL, RESULT, ABSERR, 1 W(LW)
EXTERNAL F
```

**3. Description**
D01AMF is based on the QUADPACK routine QAGI (Piessens et al [3]) [0,1] using one of the identities:

\[ \int_{-\infty}^{\infty} f(x) \, dx = \int_{-\infty}^{0} f(a-\frac{1}{1-t}) \, dt \]
\[ + \int_{0}^{\infty} f(a+\frac{1}{1-t}) \, dt \]

where \( a \) represents a finite integration limit. An adaptive procedure, based on the Gauss seven-point and Kronrod 15-point rules, is then employed on the transformed integral. The algorithm, described by de Doncker [1], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [2]) together with the (epsilon)-algorithm (Wynn [4]) to perform extrapolation. The local error estimation is described by Piessens et al [3].

4. References


5. Parameters

1: F -- DOUBLE PRECISION FUNCTION, supplied by the user.
External Procedure

On entry: the point at which the integrand \( f \) must be evaluated.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION } F( X )
\]

\[
\text{DOUBLE PRECISION } X
\]

1: \( X \) -- DOUBLE PRECISION  
On entry: the point at which the integrand \( f \) must be evaluated.

\( F \) must be declared as EXTERNAL in the (sub)program from which D01AMF is called. Parameters denoted as Input must not be changed by this procedure.

2: \( BOUND \) -- DOUBLE PRECISION  
On entry: the finite limit of the integration range (if present). \( BOUND \) is not used if the interval is doubly infinite.

3: \( INF \) -- INTEGER  
On entry: indicates the kind of integration range:
   if \( INF \) =1, the range is \([BOUND, +\infty)\)
   if \( INF \) =-1, the range is \((-\infty, BOUND]\)
   if \( INF \) =+2, the range is \((-\infty, +\infty)\).
Constraint: \( INF \) =-1, 1 or 2.

4: \( EPSABS \) -- DOUBLE PRECISION  
On entry: the absolute accuracy required. If \( EPSABS \) is negative, the absolute value is used. See Section 7.

5: \( EPSREL \) -- DOUBLE PRECISION  
On entry: the relative accuracy required. If \( EPSREL \) is negative, the absolute value is used. See Section 7.

6: \( RESULT \) -- DOUBLE PRECISION  
On exit: the approximation to the integral \( I \).

7: \( ABSERR \) -- DOUBLE PRECISION  
On exit: an estimate of the modulus of the absolute error, which should be an upper bound for \(|I-RESULT|\).

8: \( W(LW) \) -- DOUBLE PRECISION array  
On exit: details of the computation, as described in Section 8.

9: \( LW \) -- INTEGER  

On entry:
the dimension of the array W as declared in the (sub)program from which D01AMF is called.
The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems. Constraint: LW >= 4.

10: IW(LIW) -- INTEGER array Output
On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

11: LIW -- INTEGER Input
On entry:
the dimension of the array IW as declared in the (sub)program from which D01AMF is called.
The number of sub-intervals into which the interval of integration may be divided cannot exceed LIW. Suggested value: LIW = LW/4. Constraint: LIW >= 1.

12: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the requested accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. If the position of a local difficulty within the interval can be determined (e.g. a singularity of the integrand or its derivative, a peak, a discontinuity, etc) you will probably gain from
splitting up the interval at this point and calling D01AMF on the infinite subrange and an appropriate integrator on the finite subrange. Alternatively, consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the amount of workspace.

IFAIL = 2
Round-off error prevents the requested tolerance from being achieved. The error may be underestimated. Consider requesting less accuracy.

IFAIL = 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL = 4
The requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily; the returned result is the best which can be obtained. The same advice applies as in the case of IFAIL = 1.

IFAIL = 5
The integral is probably divergent, or slowly convergent. It must be noted that divergence can also occur with any other non-zero value of IFAIL.

IFAIL = 6
On entry LW < 4, or LIW < 1, or INF /= -1, 1 or 2.
Please note that divergence can occur with any non-zero value of IFAIL.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves, the following accuracy:

|I-RESULT| <= tol,

where

tol = max{||EPSABS|, |EPSREL|*|I|},

and EPSABS and EPSREL are user-specified absolute and relative error tolerances. Moreover it returns the quantity ABSERR, which, in normal circumstances, satisfies
8. Further Comments

The time taken by the routine depends on the integrand and the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AMF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r_i denote the approximation to the value of the integral over the sub-interval [a_i,b_i] in the partition of [a,b] and e_i be the corresponding absolute error estimate. Then, \[ \int_{a_i}^{b_i} f(x) \, dx \approx r_i \] and RESULT = \( r_i \) unless D01AMF terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, RESULT (and ABSERR) are taken to be the values returned from the extrapolation process. The value of n is returned in IW(1), and the values a_i, b_i, e_i, and r_i are stored consecutively in the array W, that is:

\[ a = W(i), b = W(n+i), e = W(2n+i) \text{ and } r = W(3n+i). \]

Note: that this information applies to the integral transformed to (0,1) as described in Section 3, not to the original integral.

9. Example

To compute

\[ \int_{0}^{1} \frac{1}{(x+1)\sqrt{x}} \, dx. \]
The exact answer is \((\pi)\).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**1. Purpose**

D01ANF calculates an approximation to the sine or the cosine transform of a function \(g\) over \([a,b]\):

\[
\int_{a}^{b} |g(x)||\sin((\omega)x)||\,dx \quad \text{or} \quad \int_{a}^{b} |g(x)||\cos((\omega)x)||\,dx
\]

(for a user-specified value of \((\omega)\)).

**2. Specification**

```fortran
SUBROUTINE D01ANF (G, A, B, OMEGA, KEY, EPSABS, EPSREL, RESULT, ABSERR, W, LW, IW, LIW, IFAIL)
INTEGER KEY, LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION G, A, B, OMEGA, EPSABS, EPSREL, RESULT, ABSERR, W(LW)
EXTERNAL G
```

**3. Description**

D01ANF is based upon the QUADPACK routine QFOUR (Piessens et al [3]). It is an adaptive routine, designed to integrate a function of the form \(g(x)w(x)\), where \(w(x)\) is either \(\sin((\omega)x)\) or \(\cos((\omega)x)\). If a sub-interval has length

\[-1
L=|b-a|2\]
then the integration over this sub-interval is performed by means of a modified Clenshaw-Curtis procedure (Piessens and Branders [2]) if $L(\omega)>4$ and $1<=20$. In this case a Chebyshev-series approximation of degree 24 is used to approximate $g(x)$, while an error estimate is computed from this approximation together with that obtained using Chebyshev-series of degree 12. If the above conditions do not hold then Gauss 7-point and Kronrod 15-point rules are used. The algorithm, described in [3], incorporates a global acceptance criterion (as defined in Malcolm and Simpson [1]) together with the (epsilon)-algorithm Wynn [4] to perform extrapolation. The local error estimation is described in [3].

4. References


5. Parameters

1: G -- DOUBLE PRECISION FUNCTION, supplied by the user. 
   External Procedure
   G must return the value of the function $g$ at a given point.
   
   Its specification is:

   \[
   \text{DOUBLE PRECISION FUNCTION } G \text{ (X)} \\
   \text{DOUBLE PRECISION } X
   \]

1: X -- DOUBLE PRECISION
   Input
   On entry: the point at which the function $g$ must be evaluated.
   G must be declared as EXTERNAL in the (sub)program from which D01ANF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION
   Input
   On entry: the lower limit of integration, $a$. 
3: B -- DOUBLE PRECISION  
   Input  
   On entry: the upper limit of integration, b. It is not necessary that a<b.

4: OMEGA -- DOUBLE PRECISION  
   Input  
   On entry: the parameter (omega) in the weight function of the transform.

5: KEY -- INTEGER  
   Input  
   On entry: indicates which integral is to be computed:  
   if KEY = 1, w(x)=\cos((omega)x);  
   if KEY = 2, w(x)=\sin((omega)x).  
   Constraint: KEY = 1 or 2.

6: EPSABS -- DOUBLE PRECISION  
   Input  
   On entry: the absolute accuracy required. If EPSABS is negative, the absolute value is used. See Section 7.

7: EPSREL -- DOUBLE PRECISION  
   Input  
   On entry: the relative accuracy required. If EPSREL is negative, the absolute value is used. See Section 7.

8: RESULT -- DOUBLE PRECISION  
   Output  
   On exit: the approximation to the integral I.

9: ABSERR -- DOUBLE PRECISION  
   Output  
   On exit: an estimate of the modulus of the absolute error, which should be an upper bound for |I-RESULT|.

10: W(LW) -- DOUBLE PRECISION array  
    Output  
    On exit: details of the computation, as described in Section 8.

11: LW -- INTEGER  
    Input  
    On entry:  
    the dimension of the array W as declared in the (sub)program from which D01ANF is called.  
    The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: a value in the range 800 to 2000 is adequate for most problems.  
    Constraint: LW >= 4.

12: IW(LIW) -- INTEGER array  
    Output  
    On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.
13: LIW -- INTEGER
    On entry:
    the dimension of the array IW as declared in the
    (sub)program from which D01ANF is called.
    The number of sub-intervals into which the interval of
    integration may be divided cannot exceed LIW/2. Suggested

14: IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.

    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).

    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
    The maximum number of subdivisions allowed with the given
    workspace has been reached without the accuracy requested
    being achieved. Look at the integrand in order to determine
    the integration difficulties. If the position of a local
difficulty within the interval can be determined (e.g. a
singularity of the integrand or its derivative, a peak, a
discontinuity, etc) you will probably gain from splitting up
the interval at this point and calling the integrator on the
subranges. If necessary, another integrator, which is
    designed for handling the type of difficulty involved, must
    be used. Alternatively consider relaxing the accuracy
    requirements specified by EPSABS and EPSREL, or increasing
    amount of workspace.

IFAIL= 2
    Round-off error prevents the requested tolerance from being
    achieved. The error may be underestimated. Consider
    requesting less accuracy.

IFAIL= 3
    Extremely bad local behaviour of g(x) causes a very strong
subdivision around one (or more) points of the interval.
The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
The requested tolerance cannot be achieved because the
extrapolation does not increase the accuracy satisfactorily;
the returned result is the best which can be obtained. The
same advice applies as in the case of IFAIL = 1.

IFAIL= 5
The integral is probably divergent, or slowly convergent. It
must be noted that divergence can occur with any non-zero
value of IFAIL.

IFAIL= 6
On entry KEY < 1,
or       KEY > 2.

IFAIL= 7
On entry LW < 4,
or       LIW < 2.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves,
the following accuracy:

\[ |I-RESULT| \leq \text{tol}, \]

where

\[ \text{tol} = \max(|\text{EPSABS}|, |\text{EPSREL}| \times |I|), \]

and EPSABS and EPSREL are user-specified absolute and relative
tolerances. Moreover it returns the quantity ABSERR, which, in
normal circumstances, satisfies

\[ |I-RESULT| \leq \text{ABSERR} \leq \text{tol}. \]

8. Further Comments
The time taken by the routine depends on the integrand and on the
accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the
contents of the array W, which contains the end-points of the
sub-intervals used by D01ANF along with the integral
contributions and error estimates over these sub-intervals.
Specifically, for \( i = 1, 2, \ldots, n \), let \( r \) denote the approximation to the value of the integral over the sub-interval \([a, b]\) in the partition of \([a, b]\) and \( e \) be the corresponding absolute error estimate. Then, \( \int g(x)w(x)dx \approx r \) and \( \text{RESULT} = r \) unless \( \text{D01ANF} \) terminates while testing for divergence of the integral (see Piessens et al [3] Section 3.4.3). In this case, \( \text{RESULT} \) (and \( \text{ABSERR} \)) are taken to be the values returned from the extrapolation process. The value of \( n \) is returned in \( \text{IW}(1) \), and the values \( a \), \( b \), \( e \) and \( r \) are stored consecutively in the array \( W \), that is:

\[
\begin{align*}
& a = W(i), \\
& b = W(n+i), \\
& e = W(2n+i) \quad \text{and} \\
& r = W(3n+i).
\end{align*}
\]

9. Example

To compute

\[
\int_0^1 \ln(x)\sin(10\pi x)dx.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D01APF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01APF is an adaptive integrator which calculates an approximation to the integral of a function \( g(x)w(x) \) over a finite interval \([a, b]\):

\[
\int_{a}^{b} g(x)w(x)\,dx
\]

where the weight function \( w \) has end-point singularities of algebraico-logarithmic type.

2. Specification

```fortran
SUBROUTINE D01APF (G, A, B, ALFA, BETA, KEY, EPSABS,
  1   EPSREL, RESULT, ABSERR, W, LW, IW, LIW,
  2   IFAIL)
  INTEGER KEY, LW, IW(LIW), LIW, IFAIL
  DOUBLE PRECISION G, A, B, ALFA, BETA, EPSABS, EPSREL,
  1   RESULT, ABSERR, W(LW)
  EXTERNAL G
```

3. Description

D01APF is based upon the QUADPACK routine QAWSE (Piessens et al [3]) and integrates a function of the form \( g(x)w(x) \), where the weight function \( w(x) \) may have algebraico-logarithmic singularities at the end-points \( a \) and/or \( b \). The strategy is a modification of that in D01AKF. We start by bisecting the original interval and applying modified Clenshaw-Curtis integration of orders 12 and 24 to both halves. Clenshaw-Curtis integration is then used on all sub-intervals which have \( a \) or \( b \) as one of their end-points (Piessens et al [2]). On the other sub-intervals Gauss-Kronrod (7-15 point) integration is carried out.

A ‘global’ acceptance criterion (as defined by Malcolm and
Simpson [1]) is used. The local error estimation control is described by Piessens et al [3].

4. References


5. Parameters

1: G -- DOUBLE PRECISION FUNCTION, supplied by the user. External Procedure

G must return the value of the function g at a given point X.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION } G \ (X) \\
\text{DOUBLE PRECISION } X
\]

1: X -- DOUBLE PRECISION Input

On entry: the point at which the function g must be evaluated.

G must be declared as EXTERNAL in the (sub)program from which D01APF is called. Parameters denoted as Input must not be changed by this procedure.

2: A -- DOUBLE PRECISION Input

On entry: the lower limit of integration, a.

3: B -- DOUBLE PRECISION Input

On entry: the upper limit of integration, b. Constraint: B > A.

4: ALFA -- DOUBLE PRECISION Input


5: BETA -- DOUBLE PRECISION Input

6: KEY -- INTEGER       Input  
    On entry: indicates which weight function is to be used:
          (alpha)   (beta)         
    if KEY = 1, w(x)=(x-a)   (b-x)        
          (alpha)   (beta)         
    if KEY = 2, w(x)=(x-a)   (b-x)   ln(x-a)      
          (alpha)   (beta)         
    if KEY = 3, w(x)=(x-a)   (b-x)   ln(b-x)      
          (alpha)   (beta)         
    if KEY = 4, w(x)=(x-a)   (b-x)   ln(x-a)ln(b-x) 

    Constraint: KEY = 1, 2, 3 or 4

7: EPSABS -- DOUBLE PRECISION       Input 
    On entry: the absolute accuracy required. If EPSABS is 
    negative, the absolute value is used. See Section 7.

8: EPSREL -- DOUBLE PRECISION       Input  
    On entry: the relative accuracy required. If EPSREL is 
    negative, the absolute value is used. See Section 7.

9: RESULT -- DOUBLE PRECISION       Output  
    On exit: the approximation to the integral I.

10: ABSERR -- DOUBLE PRECISION      Output 
    On exit: an estimate of the modulus of the absolute error, 
    which should be an upper bound for |I-RESULT|.

11: W(LW) -- DOUBLE PRECISION array  Output 
    On exit: details of the computation, as described in 
    Section 8.

12: LW -- INTEGER        Input  
    On entry: the dimension of the array W as declared in the (sub)program 
    from which D01APF is called.  
    The value of LW (together with that of LIW below) imposes a 
    bound on the number of sub-intervals into which the interval 
    of integration may be divided by the routine. The number of 
    sub-intervals cannot exceed LW/4. The more difficult the 
    integrand, the larger LW should be. Suggested value: LW = 
    800 to 2000 is adequate for most problems. Constraint: LW >= 
    8.

13: IW(LIW) -- INTEGER array       Output 
    On exit: IW(1) contains the actual number of sub-intervals
used. The rest of the array is used as workspace.

14: LIW -- INTEGER  
Input
On entry:  
the dimension of the array IW as declared in the  
(sub)program from which D01APF is called.  
The number of sub-intervals into which the interval of  
integration may be divided cannot exceed LIW. Suggested  

15: IFAIL -- INTEGER  
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are  
unfamiliar with this parameter should refer to the Essential  
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or  
gives a warning (see Section 6).

For this routine, because the values of output parameters  
may be useful even if IFAIL /=0 on exit, users are  
recommended to set IFAIL to -1 before entry. It is then  
essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are  
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given  
workspace has been reached without the accuracy requirements  
being achieved. Look at the integrand in order to determine  
the integration difficulties. If the position of a  
discontinuity or a singularity of algebraico-logarithmic  
type within the interval can be determined, the interval  
must be split up at this point and the integrator called on  
the subranges. If necessary, another integrator, which is  
designed for handling the difficulty involved, must be used.  
Alternatively consider relaxing the accuracy requirements  
specified by EPSABS and EPSREL, or increasing the amount of  
workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being  
achieved. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong
subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL = 4
On entry B <= A,
or ALFA <= -1,
or BETA <= -1,
or KEY < 1,
or KEY > 4.

IFAIL = 5
On entry LW < 8,
or LIW < 2.

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

|I-RESULT| <= to1,

where

\[
tol = \max\{|\text{EPSABS}|, |\text{EPSREL}| \times |I|\},
\]

and EPSABS and EPSREL are user-specified absolute and relative error tolerances.

Moreover it returns the quantity ABSERR which, in normal circumstances, satisfies:

|I-RESULT| <= ABSERR <= to1.

8. Further Comments
The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01APF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let ri denote the approximation to
the value of the integral over the sub-interval $[a_i, b_i]$ in the
partition of $[a, b]$ and $e_i$ be the corresponding absolute error
estimate. Then, $\int f(x)w(x)dx = r_i$ and RESULT$> r_i$. The value of
$n_i$ is returned in IW(1), and the values $a_i$, $b_i$, $e_i$ and $r_i$ are
stored consecutively in the array $W_i$, that is:

\begin{align*}
  a_i &= W(i), \\
  b_i &= W(n+i), \\
  e_i &= W(2n+i), \\
  r_i &= W(3n+i).
\end{align*}

9. Example

To compute:

\begin{align*}
  \int_0^1 \ln(x)\cos(10\pi x)dx \quad \text{and} \\
  \int_0^{\ln(x)/\sin(10x)} 1 - x \quad \text{dx}.
\end{align*}

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
D01AQF(3NAG)  Foundation Library (12/10/92)  D01AQF(3NAG)
D01 -- Quadrature

D01AQF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01AQF calculates an approximation to the Hilbert transform of a function \( g(x) \) over \([a,b]\):

\[
\int_{a}^{b} \frac{g(x)}{x-c} \, dx
\]

for user-specified values of \( a, b \) and \( c \).

2. Specification

```fortran
SUBROUTINE D01AQF (G, A, B, C, EPSABS, EPSREL, RESULT, 1
  ABSERR, W, LW, IW, LIW, IFAIL)
  INTEGER LW, IW(LIW), LIW, IFAIL
  DOUBLE PRECISION G, A, B, C, EPSABS, EPSREL, RESULT,
  1 ABSERR, W(LW)
  EXTERNAL G
```

3. Description

D01AQF is based upon the QUADPACK routine QAWC (Piessens et al [3]) and integrates a function of the form \( g(x)w(x) \), where the weight function

\[
w(x) = \frac{1}{x-c}
\]

is that of the Hilbert transform. (If \( a < c < b \) the integral has to be interpreted in the sense of a Cauchy principal value.) It is an adaptive routine which employs a 'global' acceptance criterion (as defined by Malcolm and Simpson [1]). Special care is taken to ensure that \( c \) is never the end-point of a sub-interval (Piessens et al [2]). On each sub-interval \((c,c)\) modified Clenshaw-Curtis integration of orders 12 and 24 is performed if \( -d < c < c + d \)
where \( d = \frac{(c - c)}{20} \). Otherwise the Gauss 7-point and Kronrod 15-2 point rules are used. The local error estimation is described by Piessens et al [3].

4. References


5. Parameters

1: \( G \) -- DOUBLE PRECISION FUNCTION, supplied by the user. 

\( G \) must return the value of the function \( g \) at a given point. 

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION } G (X) \\
\text{DOUBLE PRECISION } X
\]

1: \( X \) -- DOUBLE PRECISION 

Input

On entry: the point at which the function \( g \) must be evaluated.

\( G \) must be declared as EXTERNAL in the (sub)program from which D01AQF is called. Parameters denoted as Input must not be changed by this procedure.

2: \( A \) -- DOUBLE PRECISION 

Input

On entry: the lower limit of integration, \( a \).

3: \( B \) -- DOUBLE PRECISION 

Input

On entry: the upper limit of integration, \( b \). It is not necessary that \( a < b \).

4: \( C \) -- DOUBLE PRECISION 

Input

On entry: the parameter \( c \) in the weight function. 

Constraint: \( C \) must not equal \( A \) or \( B \).

5: \( \text{EPSABS} \) -- DOUBLE PRECISION 

Input

On entry: the absolute accuracy required. If \( \text{EPSABS} \) is
negative, the absolute value is used. See Section 7.

6: EPSREL -- DOUBLE PRECISION Input
On entry: the relative accuracy required. If EPSREL is negative, the absolute value is used. See Section 7.

7: RESULT -- DOUBLE PRECISION Output
On exit: the approximation to the integral I.

8: ABSERR -- DOUBLE PRECISION Output
On exit: an estimate of the modulus of the absolute error, which should be an upper bound for |I-RESULT|.

9: W(LW) -- DOUBLE PRECISION array Output
On exit: details of the computation, as described in Section 8.

10: LW -- INTEGER Input
On entry: the dimension of the array W as declared in the (sub)program from which D01AQF is called.
The value of LW (together with that of LIW below) imposes a bound on the number of sub-intervals into which the interval of integration may be divided by the routine. The number of sub-intervals cannot exceed LW/4. The more difficult the integrand, the larger LW should be. Suggested value: LW = 800 to 2000 is adequate for most problems. Constraint: LW >= 4.

11: IW(LIW) -- INTEGER array Output
On exit: IW(1) contains the actual number of sub-intervals used. The rest of the array is used as workspace.

12: LIW -- INTEGER Input
On entry: the dimension of the array IW as declared in the (sub)program from which D01AQF is called.
The number of sub-intervals into which the interval of integration may be divided cannot exceed LIW. Suggested value: LIW = LW/4. Constraint: LIW >= 1.

13: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given workspace has been reached without the accuracy requirements being achieved. Look at the integrand in order to determine the integration difficulties. Another integrator which is designed for handling the type of difficulty involved, must be used. Alternatively consider relaxing the accuracy requirements specified by EPSABS and EPSREL, or increasing the workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being achieved. Consider requesting less accuracy.

IFAIL= 3
Extremely bad local behaviour of g(x) causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL= 4
On entry C = A or C = B.

IFAIL= 5
On entry LW < 4,
or LIW < 1.

7. Accuracy

The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I\text{-RESULT}| \leq to1, \]

where

\[ to1 = \max\{|EPSABS|, |EPSREL| \cdot |I|\}, \]

and EPSABS and EPSREL are user-specified absolute and relative
error tolerances. Moreover it returns the quantity ABSERR which, in normal circumstances satisfies:

$$|I-RESULT| \leq \text{ABSERR} \leq \text{tol.}$$

8. Further Comments

The time taken by the routine depends on the integrand and on the accuracy required.

If IFAIL /= 0 on exit, then the user may wish to examine the contents of the array W, which contains the end-points of the sub-intervals used by D01AQF along with the integral contributions and error estimates over these sub-intervals.

Specifically, for i=1,2,...,n, let r denote the approximation to the value of the integral over the sub-interval [a_i,b_i] in the partition of [a,b] and e be the corresponding absolute error estimate. Then, \( \int_{a_i}^{b_i} g(x) \, w(x) \, dx \approx r \) and RESULT > r . The value of \( \sum_{i=1}^{n} r_i \) is returned in IW(1), and the values a_i, b_i, e_i and r_i are stored consecutively in the array W, that is:

- \( a_i = W(i) \)
- \( b_i = W(n+i) \)
- \( e_i = W(2n+i) \)
- \( r_i = W(3n+i) \)

9. Example

To compute the Cauchy principal value of

\[
\int_{-1}^{1} \frac{1}{x} \, dx
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

1. Purpose

D01ASF calculates an approximation to the sine or the cosine transform of a function g over [a,\infty):

\[ I = \int_a^{\infty} g(x) \sin(\omega x) \, dx \] or \[ I = \int_a^{\infty} g(x) \cos(\omega x) \, dx \]

(for a user-specified value of \( \omega \)).

2. Specification

```fortran
SUBROUTINE D01ASF (G, A, OMEGA, KEY, EPSABS, RESULT, 1
ABSERR, LIMLST, LST, ERLST, RSLST, 2
IERLST, W, LW, IW, LIW, IFAIL)
INTEGER KEY, LIMLST, LST, IERLST(LIMLST), LW, IW, LIW, IFAIL
DOUBLE PRECISION G, A, OMEGA, EPSABS, RESULT, ABSERR, ERLST 1
(LIMLST), RSLST(LIMLST), W(LW)
EXTERNAL G
```
3. Description

D01ASF is based upon the QUADPACK routine QAWFE (Piessens et al [2]). It is an adaptive routine, designed to integrate a function of the form \( g(x)w(x) \) over a semi-infinite interval, where \( w(x) \) is either \( \sin((\omega)x) \) or \( \cos((\omega)x) \). Over successive intervals

\[ C = [a+(k-1)c, a+kc], \quad k=1,2,...,LST \]

integration is performed by the same algorithm as is used by D01ANF. The intervals \( C \) are of constant length

\[ c = \{2[|(\omega)|]+1\}(\pi)/|(\omega)|, \quad (\omega)\neq 0 \]

where \( \{x\} \) represents the largest integer less than or equal to \( x \). Since \( c \) equals an odd number of half periods, the integral contributions over succeeding intervals will alternate in sign when the function \( g \) is positive and monotonically decreasing over \([a,\infty)\). The algorithm, described by [2], incorporates a global acceptance criterion (as defined by Malcolm and Simpson [1]) together with the (epsilon)-algorithm (Wynn [3]) to perform extrapolation. The local error estimation is described by Piessens et al [2].

If \( (\omega)=0 \) and KEY = 1, the routine uses the same algorithm as D01AMF (with EPSREL = 0.0).

In contrast to the other routines in Chapter D01, D01ASF works only with a user-specified absolute error tolerance (EPSABS). Over the interval \( C \) it attempts to satisfy the absolute accuracy requirement

\[ \text{EPSA} = U \times \text{EPSABS} \]

\[ k \]

\[ k-1 \]

where \( U = (1-p)p \), for \( k=1,2,... \) and \( p=0.9 \).

However, when difficulties occur during the integration over the \( k \)th sub-interval \( C \) such that the error flag IERLST\((k)\) is non-zero, the accuracy requirement over subsequent intervals is relaxed. See Piessens et al [2] for more details.

4. References


5. Parameters

1: $G$ -- DOUBLE PRECISION FUNCTION, supplied by the user.
   External Procedure
   $G$ must return the value of the function $g$ at a given point.
   Its specification is:

   \[
   \text{DOUBLE PRECISION FUNCTION } G (X) \\
   \text{DOUBLE PRECISION } X
   \]

   1: $X$ -- DOUBLE PRECISION
   Input
   On entry: the point at which the function $g$ must be evaluated.
   $G$ must be declared as EXTERNAL in the (sub)program from which D01ASF is called. Parameters denoted as Input must not be changed by this procedure.

2: $A$ -- DOUBLE PRECISION
   Input
   On entry: the lower limit of integration, $a$.

3: $\text{OMEGA}$ -- DOUBLE PRECISION
   Input
   On entry: the parameter (omega) in the weight function of the transform.

4: $\text{KEY}$ -- INTEGER
   Input
   On entry: indicates which integral is to be computed:
   if $\text{KEY} = 1$, $w(x) = \cos((\text{OMEGA})x)$;
   if $\text{KEY} = 2$, $w(x) = \sin((\text{OMEGA})x)$.
   Constraint: $\text{KEY} = 1$ or $2$.

5: $\text{EPSABS}$ -- DOUBLE PRECISION
   Input
   On entry: the absolute accuracy requirement. If $\text{EPSABS}$ is negative, the absolute value is used. See Section 7.

6: $\text{RESULT}$ -- DOUBLE PRECISION
   Output
   On exit: the approximation to the integral $I$. 
7: ABSERR -- DOUBLE PRECISION 
   On exit: an estimate of the modulus of the absolute error, 
   which should be an upper bound for |I-RESULT|.

8: LIMLST -- INTEGER 
   On entry: an upper bound on the number of intervals C 
   needed for the integration. Suggested value: LIMLST = 50 is 
   adequate for most problems. Constraint: LIMLST >= 3.

9: LST -- INTEGER 
   On exit: the number of intervals C actually used for the 
   integration.

10: ERLST(LIMLST) -- DOUBLE PRECISION array 
    On exit: ERLST(k) contains the error estimate corresponding 
    to the integral contribution over the interval C, for 
    k = 1, 2, ..., LST.

11: RSLST(LIMLST) -- DOUBLE PRECISION array 
    On exit: RSLST(k) contains the integral contribution over 
    the interval C for k = 1, 2, ..., LST.

12: IERLST(LIMLST) -- INTEGER array 
    On exit: IERLST(k) contains the error flag corresponding to 
    RSLST(k), for k = 1, 2, ..., LST. See Section 6.

13: W(LW) -- DOUBLE PRECISION array 
    Workspace

14: LW -- INTEGER 
    On entry: the dimension of the array W as declared in the (sub)program 
    from which D01ASF is called. The value of LW (together with that of LIW below) 
    imposes a bound on the number of sub-intervals into which each 
    interval C may be divided by the routine. The number of 
    sub-intervals cannot exceed LW/4. The more difficult the 
    integrand, the larger LW should be. Suggested value: a value 
    in the range 800 to 2000 is adequate for most problems. 
    Constraint: LW >= 4.

15: IW(LIW) -- INTEGER array 
    On exit: IW(1) contains the maximum number of sub-intervals 
    actually used for integrating over any of the intervals C. 
    The rest of the array is used as workspace.
16: LIW -- INTEGER
    On entry:
    the dimension of the array IW as declared in the
    (sub)program from which D01ASF is called.
    The number of sub-intervals into which each interval C may
    be divided cannot exceed LIW/2. Suggested value: LIW = LW/2.
    Constraint: LIW >= 2.

17: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).
    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
The maximum number of subdivisions allowed with the given
workspace has been reached without the accuracy requirements
being achieved. Look at the integrand in order to determine
the integration difficulties. If the position of a local
difficulty within the interval can be determined (e.g. a
singularity of the integrand or its derivative, a peak, a
discontinuity, etc) you will probably gain from splitting up
the interval at this point and calling D01ASF on the
infinite subrange and an appropriate integrator on the
finite subrange. Alternatively, consider relaxing the
accuracy requirements specified by EPSABS or increasing the
amount of workspace.

IFAIL= 2
Round-off error prevents the requested tolerance from being
achieved. The error may be underestimated. Consider
requesting less accuracy.

IFAIL= 3
Extremely bad local integrand behaviour causes a very strong subdivision around one (or more) points of the interval. The same advice applies as in the case of IFAIL = 1.

IFAIL = 4
The requested tolerance cannot be achieved, because the extrapolation does not increase the accuracy satisfactorily; the returned result is the best which can be obtained. The same advice applies as in the case of IFAIL = 1.

IFAIL = 5
The integral is probably divergent, or slowly convergent. Please note that divergence can occur with any non-zero value of IFAIL.

IFAIL = 6
On entry KEY < 1,
or KEY > 2,
or LIMLST < 3.

IFAIL = 7
Bad integration behaviour occurs within one or more of the intervals C. Location and type of the difficulty involved can be determined from the vector IERLST (see below).

IFAIL = 8
Maximum number of intervals C (= LIMLST) allowed has been achieved. Increase the value of LIMLST to allow more cycles.

IFAIL = 9
The extrapolation table constructed for convergence acceleration of the series formed by the integral contribution over the intervals C, does not converge to the required accuracy.

IFAIL = 10
On entry LW < 4,
or LIW < 2.

In the cases IFAIL = 7, 8 or 9, additional information about the cause of the error can be obtained from the array IERLST, as follows:

IERLST(k)=1
The maximum number of subdivisions = \( \min(LW/4, LIW/2) \) has been achieved on the kth interval.

IERLST(k)=2
Occurrence of round-off error is detected and prevents the tolerance imposed on the kth interval from being achieved.

IERLST(k)=3
Extremely bad integrand behaviour occurs at some points of the kth interval.

IERLST(k)=4
The integration procedure over the kth interval does not converge (to within the required accuracy) due to round-off in the extrapolation procedure invoked on this interval. It is assumed that the result on this interval is the best which can be obtained.

IERLST(k)=5
The integral over the kth interval is probably divergent or slowly convergent. It must be noted that divergence can occur with any other value of IERLST(k).

7. Accuracy
The routine cannot guarantee, but in practice usually achieves, the following accuracy:

\[ |I - \text{RESULT}| \leq |\text{EPSABS}|, \]

where EPSABS is the user-specified absolute error tolerance. Moreover, it returns the quantity ABSERR, which, in normal circumstances, satisfies

\[ |I - \text{RESULT}| \leq \text{ABSERR} \leq |\text{EPSABS}|. \]

8. Further Comments
None.

9. Example
To compute

\[
\int_{0}^{\infty} \frac{\cos((\pi)x/2)}{x} \, dx
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.

1. Purpose

D01BBF returns the weights and abscissae appropriate to a Gaussian quadrature formula with a specified number of abscissae. The formulae provided are Gauss-Legendre, Gauss-Rational, Gauss-Laguerre and Gauss-Hermite.

2. Specification

```
SUBROUTINE D01BBF (A, B, ITYPE, N, WEIGHT, ABSCIS, GTYPE, IFAIL)
INTEGER ITYPE, N, GTYPE, IFAIL
DOUBLE PRECISION A, B, WEIGHT(N), ABSCIS(N)
```

3. Description

This routine returns the weights and abscissae for use in the Gaussian quadrature of a function $f(x)$. The quadrature takes the form

$$
\sum_{i=1}^{n} w_i f(x_i)
$$

where $w_i$ are the weights and $x_i$ are the abscissae (see Davis and Rabinowitz [1], Froberg [2], Ralston [3] or Stroud and Secrest [4]).
Weights and abscissae are available for Gauss-Legendre, Gauss-Rational, Gauss-Laguerre and Gauss-Hermite quadrature, and for a selection of values of \( n \) (see Section 5).

(a) Gauss-Legendre Quadrature:

\[
b \int_{a}^{b} f(x) \, dx
\]

where \( a \) and \( b \) are finite and it will be exact for any function of the form

\[
\sum_{i=0}^{2n-1} c_i x^i
\]

(b) Gauss-Rational quadrature:

\[
\int_{a}^{b} \frac{f(x)}{x} \, dx (a+b>0) \quad \text{or} \quad \int_{a}^{b} \frac{f(x)}{x} \, dx (a+b<0)
\]

and will be exact for any function of the form

\[
\sum_{i=0}^{2n-1} \frac{c_i}{x+b} (2n+1-i) x^{2n+1-i}
\]

(c) Gauss-Laguerre quadrature, adjusted weights option:

\[
\int_{a}^{b} e^{-bx} f(x) \, dx \quad \text{or} \quad \int_{a}^{b} e^{-bx} f(x) \, dx
\]

and will be exact for any function of the form

\[
\sum_{i=0}^{2n-1} c_i (x+b)^i
\]

(d) Gauss-Hermite quadrature, adjusted weights option:
\[ \int_{-\infty}^{+\infty} f(x) \, dx \]

and will be exact for any function of the form

\[ \sum_{i=0}^{2n-1} \frac{2^n}{n!} f(x) \, dx \]

(e) Gauss-Laguerre quadrature, normal weights option:

\[ \int_{a}^{b} e^{-bx} f(x) \, dx \quad \text{or} \quad \int_{-\infty}^{+\infty} e^{-bx} f(x) \, dx \]

and will be exact for any function of the form

\[ \sum_{i=0}^{2n-1} \frac{2^n}{n!} e^{-bx} f(x) \, dx \]

(b<0)

(f) Gauss-Hermite quadrature, normal weights option:

\[ \int_{-\infty}^{+\infty} e^{bx} f(x) \, dx \]

and will be exact for any function of the form

\[ \sum_{i=0}^{2n-1} \frac{2^n}{n!} e^{bx} f(x) \, dx \]

Note: that the Gauss-Legendre abscissae, with \( a=-1, b=+1 \), are the zeros of the Legendre polynomials; the Gauss-Laguerre abscissae, with \( a=0, b=1 \), are the zeros of the Laguerre polynomials; and the Gauss-Hermite abscissae, with \( a=0, b=1 \), are the zeros of the
Hermite polynomials.

4. References


5. Parameters

1: A -- DOUBLE PRECISION Input

2: B -- DOUBLE PRECISION Input
   On entry: the quantities a and b as described in the appropriate subsection of Section 3.

3: ITYPE -- INTEGER Input
   On entry: indicates the type of weights for Gauss-Laguerre or Gauss-Hermite quadrature (see Section 3):

   if ITYPE = 1, adjusted weights will be returned;

   if ITYPE = 0, normal weights will be returned.

   Constraint: ITYPE = 0 or 1.

   For Gauss-Legendre or Gauss-Rational quadrature, this parameter is not used.

4: N -- INTEGER Input
   On entry: the number of weights and abscissae to be returned, N. Constraint: N = 1,2,3,4,5,6,8,10,12,14,16,20,24,32,48 or 64.

5: WEIGHT(N) -- DOUBLE PRECISION array Output
   On exit: the N weights. For Gauss-Laguerre and Gauss-Hermite quadrature, these will be the adjusted weights if ITYPE = 1, and the normal weights if ITYPE = 0.

6: ABSCIS(N) -- DOUBLE PRECISION array Output
   On exit: the N abscissae.
7: **GTYPE** -- INTEGER  
    Input  
    On entry: The value of GTYPE indicates which quadrature  
    formula are to be used:  
    GTYPE = 0 for Gauss-Legendre weights and abscissae;  
    GTYPE = 1 for Gauss-Rational weights and abscissae;  
    GTYPE = 2 for Gauss-Laguerre weights and abscissae;  
    GTYPE = 3 for Gauss-Hermite weights and abscissae.

8: **IFAIL** -- INTEGER  
    Input/Output  
    On entry: IFAIL must be set to 0, -1 or 1. For users not  
    familiar with this parameter (described in the Essential  
    Introduction) the recommended value is 0.  
    On exit: IFAIL = 0 unless the routine detects an error (see  
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

**IFAIL**= 1  
The N-point rule is not among those stored. If the soft fail  
option is used, the weights and abscissae returned will be  
those for the largest valid value of N less than the  
requested value, and the excess elements of WEIGHT and  
ABSCIS (i.e., up to the requested N) will be filled with  
zeros.

**IFAIL**= 2  
The value of A and/or B is invalid.  
    Gauss-Rational: A + B = 0  
    Gauss-Laguerre: B = 0  
    Gauss-Hermite: B <= 0  
If the soft fail option is used the weights and abscissae  
are returned as zero.

**IFAIL**= 3  
Laguerre and Hermite normal weights only: underflow is  
occurring in evaluating one or more of the normal weights.  
If the soft fail option is used, the underflowing weights  
are returned as zero. A smaller value of N must be used; or  
adjusted weights should be used (ITYPE = 1). In the latter  
case, take care that underflow does not occur when  
evaluating the integrand appropriate for adjusted weights.
IFAIL=4
   GTYPE < 0 or GTYPE > 3

7. Accuracy

The weights and abscissae are stored for standard values of A and B to full machine accuracy.

8. Further Comments

Timing is negligible.

9. Example

This example program returns the abscissae and (adjusted) weights for the six-point Gauss-Laguerre formula.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The routine returns an estimate of a multi-dimensional integral over a hyper-rectangle (i.e., with constant limits), and also an estimate of the relative error. The user sets the relative accuracy required, supplies the integrand as a function subprogram (FUNCTN), and also sets the minimum and maximum acceptable number of calls to FUNCTN (in MINPTS and MAXPTS).

The routine operates by repeated subdivision of the hyper-rectangular region into smaller hyper-rectangles. In each subregion, the integral is estimated using a seventh-degree rule, and an error estimate is obtained by comparison with a fifth-degree rule which uses a subset of the same points. The fourth differences of the integrand along each co-ordinate axis are evaluated, and the subregion is marked for possible future subdivision in half along that co-ordinate axis which has the largest absolute fourth difference.

If the estimated errors, totalled over the subregions, exceed the requested relative error (or if fewer than MINPTS calls to FUNCTN have been made), further subdivision is necessary, and is performed on the subregion with the largest estimated error, that subregion being halved along the appropriate co-ordinate axis.

The routine will fail if the requested relative error level has not been attained by the time MAXPTS calls to FUNCTN have been made; or, if the amount LENWRK of working storage is insufficient. A formula for the recommended value of LENWRK is given in Section 5. If a smaller value is used, and is exhausted in the course of execution, the routine switches to a less efficient mode of operation; only if this mode also breaks down is insufficient storage reported.

D01FCF is based on the HALF subroutine developed by van Dooren and de Ridder [1]. It uses a different basic rule, described by Genz and Malik [2].

4. References


5. Parameters

1: NDIM -- INTEGER Input
   On entry: the number of dimensions of the integral, n.
Constraint: $2 \leq \text{NDIM} \leq 15$.

2: $A(\text{NDIM})$ -- DOUBLE PRECISION array, Input
   On entry: the lower limits of integration, $a_i$, for $i=1,2,...,n$.

3: $B(\text{NDIM})$ -- DOUBLE PRECISION array, Input
   On entry: the upper limits of integration, $b_i$, for $i=1,2,...,n$.

4: $\text{MINPTS}$ -- INTEGER, Input/Output
   On entry: $\text{MINPTS}$ must be set to the minimum number of integrand evaluations to be allowed. On exit: $\text{MINPTS}$ contains the actual number of integrand evaluations used by D01FCF.

5: $\text{MAXPTS}$ -- INTEGER, Input
   On entry: the maximum number of integrand evaluations to be allowed.
   Constraints:
   $\text{MAXPTS} \geq \text{MINPTS}$

   $\text{MAXPTS} \geq (\alpha)$,

   $\text{NDIM} \geq 2$

   where $(\alpha) = 2 + 2 \cdot \text{NDIM} + 2 \cdot \text{NDIM} + 1$.

6: $\text{FUNCTN}$ -- DOUBLE PRECISION FUNCTION, supplied by the user.
   External Procedure
   $\text{FUNCTN}$ must return the value of the integrand $f$ at a given point.

   Its specification is:

   DOUBLE PRECISION FUNCTION $\text{FUNCTN}(\text{NDIM}, Z)$
   INTEGER $\text{NDIM}$
   DOUBLE PRECISION $Z(\text{NDIM})$

1: $\text{NDIM}$ -- INTEGER, Input
   On entry: the number of dimensions of the integral, $n$.

2: $Z(\text{NDIM})$ -- DOUBLE PRECISION array, Input
   On entry: the co-ordinates of the point at which the integrand must be evaluated.

   $\text{FUNCTN}$ must be declared as EXTERNAL in the (sub)program from which D01FCF is called. Parameters denoted as Input must not be changed by this procedure.
7: EPS -- DOUBLE PRECISION
   Input
   On entry: the relative error acceptable to the user. When
   the solution is zero or very small relative accuracy may not
   be achievable but the user may still set EPS to a reasonable
   value and check for the error exit IFAIL = 2. Constraint:
   EPS > 0.0.

8: ACC -- DOUBLE PRECISION
   Output
   On exit: the estimated relative error in FINVAL.

9: LENWRK -- INTEGER
   Input
   On entry:
   the dimension of the array WRKSTR as declared in the
   (sub)program from which D01FCF is called.
   Suggested value: for maximum efficiency, LENWRK >=
   (NDIM+2)*(1+MAXPTS/(alpha)) (see parameter MAXPTS for
   (alpha)).

   If LENWRK is less than this, the routine will usually run
   less efficiently and may fail. Constraint: LENWRK>=2*NDIM+4.

10: WRKSTR(LENWRK) -- DOUBLE PRECISION array
    Workspace

11: FINVAL -- DOUBLE PRECISION
    Output
    On exit: the best estimate obtained for the integral.

12: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).

    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit. To suppress
    the output of an error message when soft failure occurs, set
    IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
   On entry NDIM < 2,
   or NDIM > 15,
or MAXPTS is too small,

or LENWRK<2*NDIM+4,

or EPS <= 0.0.

IFAIL= 2
MAXPTS was too small to obtain the required relative accuracy EPS. On soft failure, FINVAL and ACC contain estimates of the integral and the relative error, but ACC will be greater than EPS.

IFAIL= 3
LENWRK was too small. On soft failure, FINVAL and ACC contain estimates of the integral and the relative error, but ACC will be greater than EPS.

7. Accuracy
A relative error estimate is output through the parameter ACC.

8. Further Comments
Execution time will usually be dominated by the time taken to evaluate the integrand FUNCTN, and hence the maximum time that could be taken will be proportional to MAXPTS.

9. Example
This example program estimates the integral

\[
\int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \int_{0}^{1} \frac{4z \exp(2z z)}{1 + z + z^2} \, dz \, dz \, dz \, dz = 0.575364.
\]

The accuracy requested is one part in 10,000.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D01GAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D01GAF integrates a function which is specified numerically at four or more points, over the whole of its specified range, using third-order finite-difference formulae with error estimates, according to a method due to Gill and Miller.

2. Specification

SUBROUTINE D01GAF (X, Y, N, ANS, ER, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N), Y(N), ANS, ER

3. Description

This routine evaluates the definite integral

$$\int_{x_1}^{x_n} y(x) \, dx, \quad 1 \rightarrow n$$

where the function y is specified at the n-points x_1, x_2, ..., x_n, which should be all distinct, and in either ascending or descending order. The integral between successive points is calculated by a four-point finite-difference formula centred on the interval concerned, except in the case of the first and last intervals, where four-point forward and backward difference formulae respectively are employed. If n is less than 4, the routine fails. An approximation to the truncation error is integrated and added to the result. It is also returned separately to give an estimate of the uncertainty in the result. The method is due to Gill and Miller.

4. References

5. Parameters

1: \( X(N) \) -- DOUBLE PRECISION array  
   On entry: the values of the independent variable, i.e., the  
   \( x_1, x_2, \ldots, x_n \). Constraint: either \( x_1 < x_2 < \ldots < x_N \) or  
   \( x_1 > x_2 > \ldots > x_N \).

2: \( Y(N) \) -- DOUBLE PRECISION array  
   On entry: the values of the dependent variable \( y_i \) at the  
   points \( x_i \), for \( i=1,2,\ldots,n \).

3: \( N \) -- INTEGER  
   On entry: the number of points, \( n \). Constraint: \( N \geq 4 \).

4: \( ANS \) -- DOUBLE PRECISION  
   On exit: the estimate of the integral.

5: \( ER \) -- DOUBLE PRECISION  
   On exit: an estimate of the uncertainty in \( ANS \).

6: \( IFAIL \) -- INTEGER  
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.

   On exit: \( IFAIL = 0 \) unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

\( IFAIL = 1 \)

Indicates that fewer than four-points have been supplied to  
the routine.

\( IFAIL = 2 \)

Values of \( X \) are neither strictly increasing nor strictly  
decreasing.

\( IFAIL = 3 \)

Two points have the same \( X \)-value.

No error is reported arising from the relative magnitudes of \( ANS \)  
and \( ER \) on return, due to the difficulty when the true answer is  
zero.
7. Accuracy

No accuracy level is specified by the user before calling the routine but on return ABS(ER) is an approximation to, but not necessarily a bound for, |I-ANS|. If on exit IFAIL > 0, both ANS and ER are returned as zero.

8. Further Comments

The time taken by the routine depends on the number of points supplied, n.

In their paper, Gill and Miller [1] do not add the quantity ER to ANS before return. However, extensive tests have shown that a dramatic reduction in the error often results from such addition. In other cases, it does not make an improvement, but these tend to be cases of low accuracy in which the modified answer is not significantly inferior to the unmodified one. The user has the option of recovering the Gill-Miller answer by subtracting ER from ANS on return from the routine.

9. Example

The example program evaluates the integral

\[
\int_{0}^{\frac{\pi}{2}} \frac{1}{1+x} \, dx = (\pi)
\]

reading in the function values at 21 unequally-spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
D01GBF returns an approximation to the integral of a function over a hyper-rectangular region, using a Monte Carlo method. An approximate relative error estimate is also returned. This routine is suitable for low accuracy work.

2. Specification

```fortran
SUBROUTINE D01GBF (NDIM, A, B, MINCLS, MAXCLS, FUNCTN, EPS, ACC, LENWRK, WRKSTR, FINEST, IFAIL)
  INTEGER NDIM, MINCLS, MAXCLS, LENWRK, IFAIL
  DOUBLE PRECISION A(NDIM), B(NDIM), FUNCTN, EPS, ACC, WRKSTR
  EXTERNAL FUNCTN
```

3. Description

D01GBF uses an adaptive Monte Carlo method based on the algorithm described by Lautrup [1]. It is implemented for integrals of the form:

\[
\int_{a_1}^{b_1} \int_{a_2}^{b_2} \ldots \int_{a_n}^{b_n} f(x_1, x_2, \ldots, x_n) \, dx_1 \, dx_2 \, \ldots \, dx_n
\]

Upon entry, unless LENWRK has been set to the minimum value \(10 \times NDIM\), the routine subdivides the integration region into a number of equal volume subregions. Inside each subregion the integral and the variance are estimated by means of pseudo-random sampling. All contributions are added together to produce an estimate for the whole integral and total variance. The variance along each co-ordinate axis is determined and the routine uses this information to increase the density and change the widths of the sub-intervals along each axis, so as to reduce the total variance. The total number of subregions is then increased by a factor of two and the program recycles for another iteration. The program stops when a desired accuracy has been reached or too many integral evaluations are needed for the next cycle.

4. References


5. Parameters
1: NDIM -- INTEGER  
   Input  
   On entry: the number of dimensions of the integral, n.  
   Constraint: NDIM >= 1.

2: A(NDIM) -- DOUBLE PRECISION array  
   Input  
   On entry: the lower limits of integration, a_i, for i  
   1, 2, ..., n.

3: B(NDIM) -- DOUBLE PRECISION array  
   Input  
   On entry: the upper limits of integration, b_i, for i  
   1, 2, ..., n.

4: MINCLS -- INTEGER  
   Input/Output  
   On entry: MINCLS must be set:  
   either to the minimum number of integrand evaluations to be  
   allowed, in which case MINCLS >= 0;  
   or to a negative value. In this case the routine assumes  
   that a previous call had been made with the same parameters  
   NDIM, A and B and with either the same integrand (in which  
   case D01GBF continues calculation) or a similar integrand  
   (in which case D01GBF begins the calculation with the  
   subdivision used in the last iteration of the previous call)  
   . See also WRKSTR. On exit: MINCLS contains the number of  
   integrand evaluations actually used by D01GBF.

5: MAXCLS -- INTEGER  
   Input  
   On entry: the maximum number of integrand evaluations to be  
   allowed. In the continuation case this is the number of new  
   integrand evaluations to be allowed. These counts do not  
   include zero integrand values.  
   Constraints:  
   MAXCLS > MINCLS,  
   MAXCLS >= 4*(NDIM+1).

6: FUNCTN -- DOUBLE PRECISION FUNCTION, supplied by the user.  
   External Procedure  
   FUNCTN must return the value of the integrand f at a given  
   point.  
   Its specification is:

   DOUBLE PRECISION FUNCTION FUNCTN (NDIM, X)  
   INTEGER NDIM  
   DOUBLE PRECISION X(NDIM)
1: NDIM -- INTEGER
   On entry: the number of dimensions of the integral, n.

2: X(NDIM) -- DOUBLE PRECISION array
   On entry: the co-ordinates of the point at which the
   integrand must be evaluated.
   FUNCTN must be declared as EXTERNAL in the (sub)program
   from which D01GBF is called. Parameters denoted as
   Input must not be changed by this procedure.

7: EPS -- DOUBLE PRECISION
   On entry: the relative accuracy required. Constraint: EPS
   >= 0.0.

8: ACC -- DOUBLE PRECISION
   On exit: the estimated relative accuracy of FINEST.

9: LENWRK -- INTEGER
   On entry: the dimension of the array WRKSTR as declared in the
   (sub)program from which D01GBF is called.
   For maximum efficiency, LENWRK should be about
   1/NDIM
   3*NDIM*(MAXCLS/4) +7*NDIM.
   If LENWRK is given the value 10*NDIM then the subroutine
   uses only one iteration of a crude Monte Carlo method with
   MAXCLS sample points. Constraint: LENWRK >= 10*NDIM.

10: WRKSTR(LENWRK) -- DOUBLE PRECISION array
    On entry: if MINCLS<0, WRKSTR must be unchanged from the
    previous call of D01GBF - except that for a new integrand
    WRKSTR(LENWRK) must be set to 0.0. See also MINCLS. On
    exit: WRKSTR contains information about the current sub-
    interval structure which could be used in later calls of
    D01GBF. In particular, WRKSTR(j) gives the number of sub-
    intervals used along the jth co-ordinate axis.

11: FINEST -- DOUBLE PRECISION
    On exit: the best estimate obtained for the integral.

12: IFAIL -- INTEGER
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.

    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).

    For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit. To suppress the output of an error message when soft failure occurs, set IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
On entry NDIM < 1,
or  MINCLS >= MAXCLS,
or  LENWRK < 10*NDIM,
or  MAXCLS < 4*(NDIM+1),
or  EPS < 0.0.

IFAIL= 2
MAXCLS was too small for D01GBF to obtain the required relative accuracy EPS. In this case D01GBF returns a value of FINEST with estimated relative error ACC, but ACC will be greater than EPS. This error exit may be taken before MAXCLS non-zero integrand evaluations have actually occurred, if the routine calculates that the current estimates could not be improved before MAXCLS was exceeded.

7. Accuracy

A relative error estimate is output through the parameter ACC. The confidence factor is set so that the actual error should be less than ACC 90% of the time. If a user desires a higher confidence level then a smaller value of EPS should be used.

8. Further Comments

The running time for D01GBF will usually be dominated by the time used to evaluate the integrand FUNCTN, so the maximum time that could be used is approximately proportional to MAXCLS.

For some integrands, particularly those that are poorly behaved in a small part of the integration region, D01GBF may terminate with a value of ACC which is significantly smaller than the actual relative error. This should be suspected if the returned value of MINCLS is small relative to the expected difficulty of the integral. Where this occurs, D01GBF should be called again, but with a higher entry value of MINCLS (e.g. twice the returned
value) and the results compared with those from the previous call.

The exact values of FINEST and ACC on return will depend (within statistical limits) on the sequence of random numbers generated within D01GBF by calls to G05CAF. Separate runs will produce identical answers unless the part of the program executed prior to calling D01GBF also calls (directly or indirectly) routines from Chapter G05, and the series of such calls differs between runs. If desired, the user may ensure the identity or difference between runs of the results returned by D01GBF, by calling G05CBF or G05CCF respectively, immediately before calling D01GBF.

9. Example

This example program calculates the integral

\[
\int_0^1 \frac{4x}{x+1} \exp(2x-x) \, dx = \int_{x=0}^{x=1} \frac{4x}{x+1} \exp(2x-x) \, dx = 0.575364.
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

NagIntegrationPackage (NAGD01)
Exports:
d01ajf  d01akf  d01alf  d01amf  d01anf
d01apf  d01aqf  d01asf  d01bbf  d01fcf
d01gaf  d01gbf

— package NAGD01 NagIntegrationPackage —

)abbrev package NAGD01 NagIntegrationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:37 1994
++ Description:
++ This package uses the NAG Library to calculate the numerical value of
++ definite integrals in one or more dimensions and to evaluate
++ weights and abscissae of integration rules.

NagIntegrationPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==> with
d01ajf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Aspl(F))) -> Result
++ d01ajf(a,b,epsabs,epsrel,lw,liw,ifail,f)
++ is a general-purpose integrator which calculates an
++ approximation to the integral of a function f(x) over a finite
++ interval [a,b]:
++ See \downlink{Manual Page}{manpageXXd01ajf}.
d01akf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Aspl(F))) -> Result
++ d01akf(a,b,epsabs,epsrel,lw,liw,ifail,f)
++ is an adaptive integrator, especially suited to
++ oscillating, non-singular integrands, which calculates an
++ approximation to the integral of a function f(x) over a finite
++ interval [a,b]:
++ See \downlink{Manual Page}{manpageXXd01akf}.
d01alf : (DoubleFloat,DoubleFloat,Integer,Matrix DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Aspl(F))) -> Result
++ d01alf(a,b,npts,points,epsabs,epsrel,lw,liw,ifail,f)
++ is a general purpose integrator which calculates an
++ approximation to the integral of a function f(x) over a finite
++ interval [a,b]:
++ See \downlink{Manual Page}{manpageXXd01alf}.
d01amf : (DoubleFloat,Integer,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Aspl(F))) -> Result
++ d01amf(bound,inf,epsabs,epsrel,lw,liw,ifail,f)
++ calculates an approximation to the integral of a function
++ f(x) over an infinite or semi-infinite interval [a,b]:
++ See \downlink{Manual Page}{manpageXXd01amf}.
d01anf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer,Union(fn:FileName,fp:Asp1(G))) \to Result
++ d01anf(a,b,omega,key,epsabs,epsrel,\ell w,\ell iw,ifail,g)
++ calculates an approximation to the sine or the cosine
++ transform of a function g over \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01anf}.

d01apf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer,Union(fn:FileName,fp:Asp1(G))) \to Result
++ d01apf(a,b,alfa,beta,key,epsabs,epsrel,\ell w,\ell iw,ifail,g)
++ is an adaptive integrator which calculates an
++ approximation to the integral of a function g(x)w(x) over a
++ finite interval \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01apf}.

d01aqf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer,Union(fn:FileName,fp:Asp1(G))) \to Result
++ d01aqf(a,b,c,epsabs,epsrel,\ell w,\ell iw,ifail,g)
++ calculates an approximation to the Hilbert transform of a
++ function g(x) over \([a,b]\):
++ See \downlink{Manual Page}{manpageXXd01aqf}.

d01asf : (DoubleFloat,DoubleFloat,Integer,Integer,Union(fn:FileName,fp:Asp1(G))) \to Result
++ d01asf(a,omega,key,epsabs,\ell mlst,\ell w,\ell iw,ifail,g)
++ calculates an approximation to the sine or the cosine
++ transform of a function g over \([a,\infty)\):
++ See \downlink{Manual Page}{manpageXXd01asf}.

d01bbf : (DoubleFloat,DoubleFloat,Integer,Integer=Integer,Integer,Integer) \to Result
++ d01bbf(a,b,itype,n,gtype,ifail)
++ returns the weight appropriate to a
++ Gaussian quadrature.
++ The formulae provided are Gauss-Legendre, Gauss-Rational, Gauss-
++ Laguerre and Gauss-Hermite.
++ See \downlink{Manual Page}{manpageXXd01bbf}.

d01fcf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer,Integer,Union(fn:FileName,fp:Asp4(FUNCTN))) \to Result
++ d01fcf(ndim,a,b,maxpts,eps,lenwrk,\ell mpts,ifail,functn)
++ attempts to evaluate a multi-dimensional integral (up to
++ 15 dimensions), with constant and finite limits, to a specified
++ relative accuracy, using an adaptive subdivision strategy.
++ See \downlink{Manual Page}{manpageXXd01fcf}.

d01gaf : (Matrix DoubleFloat,Matrix DoubleFloat,Integer,Integer) \to Result
++ d01gaf(x,y,n,ifail)
++ integrates a function which is specified numerically at
++ four or more points, over the whole of its specified range, using
++ third-order finite-difference formulae with error estimates,
++ according to a method due to Gill and Miller.  
++ See \downlink{Manual Page}{manpageXXd01gaf}.

d01gbf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_
  DoubleFloat,Integer,Integer,Matrix DoubleFloat,Integer,_
  Union(fn:FileName,fp:Asp4(FUNCTN)) ) \rightarrow Result
++ d01gbf(ndim,a,b,maxcls,eps,lenwrk,mincls,wrkstr,ifail,functn)
++ returns an approximation to the integral of a function
++ over a hyper-rectangular region, using a Monte Carlo method. An
++ approximate relative error estimate is also returned. This
++ routine is suitable for low accuracy work.
++ See \downlink{Manual Page}{manpageXXd01gbf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName,fp:Asp1(F))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)

\[
d01ajf\(aArg\:DoubleFloat,bArg\:DoubleFloat,epsabsArg\:DoubleFloat,_
  epsrelArg\:DoubleFloat,lwArg\:Integer,liwArg\:Integer,_
  ifailArg\:Integer,fArg\:Union(fn:FileName,fp:Asp1(F))\): Result ==
\]
invocationNagman([\[fFilename\]$Lisp,_
  "d01ajf"$Lisp,_
    "w":S,"iw":S]$Lisp,_
    "result":S,"abserr":S,["w":S,"iw":S]$Lisp,"f":S]$Lisp_,
    "ifail":S]$Lisp_]
)\$Lisp_\]
\[
[\[aArg::Any,bArg::Any,epsabsArg::Any,epsrelArg::Any,lwArg::Any,_
  liwArg::Any,ifailArg::Any ]\]_
\[ \textbf{CHAPTER 15. CHAPTER N} \]

\begin{verbatim}
pretend List (Record(key:Symbol,entry:Any))
\)

\textbf{d01akf(}aArg:DoubleFloat,bArg:DoubleFloat,epsabsArg:DoubleFloat,_,
epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_,
ifailArg:Integer,fArg:Union(fn:FileName,fp:Asp1(F))\): Result ==
pushFortranOutputStack(ffilename := aspFilename "f")$FOP
if fArg case fn
then outputAsFortran(fArg.fn)
else outputAsFortran(fArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([ffilename]$Lisp,
"d01akf",
"w"::S,"iw"::S]$Lisp,
,"ifail"::S]$Lisp_]
]$Lisp_,
[(aArg::Any,bArg::Any,epsabsArg::Any,epsrelArg::Any,_,
liwArg::Any,ifailArg::Any )]
pretend List (Record(key:Symbol,entry:Any))
\)
\end{verbatim}
PACKAGE NAGD01 NAGINTEGRATIONPACKAGE

$d01amf$(boundArg:DoubleFloat, infArg:Integer, epsabsArg:DoubleFloat, epsrelArg:DoubleFloat, _
  lwArg:Integer, liwArg:Integer, ifailArg:Integer, fArg:Union(fn:FileName, fp:Asp1(F))): Result ==

$d01anf$(aArg:DoubleFloat, bArg:DoubleFloat, omegaArg:DoubleFloat, _
  keyArg:Integer, epsabsArg:DoubleFloat, epsrelArg:DoubleFloat, _
  lwArg:Integer, liwArg:Integer, ifailArg:Integer, _
  gArg:Union(fn:FileName, fp:Asp1(G))): Result ==
CHAPTER 15. CHAPTER N

"ifail"::S$Lisp_,
$List_,
 ["aArg::Any,bArg::Any,omegaArg::Any,keyArg::Any,epsabsArg::Any,_
  epsrelArg::Any,lwArg::Any,liwArg::Any,ifailArg::Any ]$List}_
@List Any]$Lisp)@
pretend List (Record(key:Symbol,entry:Any))$Result

pre01apf(aArg:DoubleFloat,bArg:DoubleFloat,alfaArg:DoubleFloat,_
  betaArg:DoubleFloat,keyArg:Integer,epsabsArg:DoubleFloat,_
  epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_
  ifailArg:Integer,gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
  if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
  [[invokeNagman([gFilename]$Lisp,_
"d01apf","["a"::S,"b"::S,"alfa"::S,"beta"::S,"key"::S_,
  "abserr"::S,"ifail"::S,"g"::S,"w"::S,"iw"::S$Lisp_,
 ["w"::S,"lw"::S$Lisp,"g"::S$Lisp_,
  ,"ifail"::S]$Lisp_}
]$Lisp_]

pre01aqf(aArg:DoubleFloat,bArg:DoubleFloat,alfaArg:DoubleFloat,_
  betaArg:DoubleFloat,keyArg:Integer,epsabsArg:DoubleFloat,_
  epsrelArg:DoubleFloat,lwArg:Integer,liwArg:Integer,_
  ifailArg:Integer,gArg:Union(fn:FileName,fp:Asp1(G))): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
  if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
  [[invokeNagman([gFilename]$Lisp,_
"d01aqf","["a"::S,"b"::S,"c"::S,"epsabs"::S,"epsrel"::S_,
  "g"::S,"w"::S,"iw"::S$Lisp_,
["result"::S,"abserr"::S,"w"::S,"iw"::S,"g"::S$Lisp_,
  ,"ifail"::S]$Lisp_]
}
PACKAGE NAGD01 NAGINTEGRATIONPACKAGE


pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
then outputAsFortran(gArg.fn)
else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([gFilename]$Lisp, 
"d01asf", 
"a":S,"omega":S,"key":S,"epsabs":S,"limlst":S,
)$Lisp_ 
["result":S,"abser":S,"iw":S,"ifail":S]$Lisp_,
[(aArg::Any,bArg::Any,cArg::Any,epsabsArg::Any, _ lwArg::Any,liwArg::Any,ifailArg::Any )]
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any)))$Result


[(invokeNagman(NIL$Lisp, 
"d01bbf", 
"a":S,"b":S,"itype":S,"n":S,"gtype":S_ 
["weight":S,"abscis":S]$Lisp_,
2280

CHAPTER 15. CHAPTER N

[["double":S,"a":S,"b":S,["weight":S,"n":S]$Lisp_,
 ["abscis":S,"n":S]$Lisp]$Lisp_,
 ["integer":S,"itype":S,"n":S,"gtype":S_,
 ,"ifail":S]$Lisp_,
 ]$Lisp_,
 ["weight":S,"abscis":S,"ifail":S]$Lisp_,
 [([aArg::Any,bArg::Any,itypeArg::Any,nArg::Any,_
 gtypeArg::Any,ifailArg::Any ])
 @List Any]$Lisp)$Lisp)_
 pretend List (Record(key:Symbol,entry:Any))$Result

d01fcf(ndimArg:Integer,aArg:Matrix DoubleFloat,bArg:Matrix DoubleFloat,_
 maxptsArg:Integer,epsArg:DoubleFloat,lenwrkArg:Integer,_
 minptsArg:Integer,ifailArg:Integer,_
 functnArg:Union(fn:FileName,fp:Asp4(FUNCTN))): Result ==
pushFortranOutputStack(functnFilename := aspFilename "functn")$FOP
if functnArg case fn
 then outputAsFortran(functnArg.fn)
 else outputAsFortran(functnArg.fp)
popFortranOutputStack()$FOP
 [(invokeNagman([functnFilename]$Lisp,_
 "d01fcf",_
 "b":S,"wrkstr":S]$Lisp_,
 "functn":S]$Lisp_,
 ,["integer":S,"ndim":S,"maxpts":S,"lenwrk":S_
 ,"minpts":S,"ifail":S]$Lisp_]
 )$Lisp_,
 pretend List (Record(key:Symbol,entry:Any))$Result

d01gaf(xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,nArg:Integer,_
 ifailArg:Integer): Result ==
 [(invokeNagman(NIL$Lisp,_
 "d01gaf",_
 ["ans":S,"er":S]$Lisp_,
 [["double":S,"x":S,"n":S]$Lisp,["y":S,"n":S]$Lisp_,
 ,"ans":S,"er":S]$Lisp_,
 ,["integer":S,"n":S,"ifail":S]$Lisp_]
 )$Lisp_,
 pretend List (Record(key:Symbol,entry:Any))$Result

([(nArg::Any,ifailArg::Any,xArg::Any,yArg::Any )]_
package NAGE01 NagInterpolationPackage

— NagInterpolationPackage.input —

)set break resume
\texttt{)sys rm -f NagInterpolationPackage.output}

\texttt{)spool NagInterpolationPackage.output}

\texttt{)set message test on}

\texttt{)set message auto off}

\texttt{)clear all}

\texttt{)show NagInterpolationPackage}

\texttt{--R NagInterpolationPackage is a package constructor}

\texttt{--R Abbreviation for NagInterpolationPackage is NAGE01}

\texttt{--R This constructor is exposed in this frame.}

\texttt{--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGE01}

\texttt{--R}

\texttt{--R------------------------------- Operations --------------------------------}

\texttt{--R e01baf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,Integer) -> Result}

\texttt{--R e01bef : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result}

\texttt{--R e01bff : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer) -> Result}

\texttt{--R e01bgf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer) -> Result}

\texttt{--R e01bhf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),DoubleFloat,DoubleFloat,Integer) -> Result}

\texttt{--R e01daf : (Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result}

\texttt{--R e01saf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result}

\texttt{--R e01sbf : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(Integer),Matrix(DoubleFloat),DoubleFloat,DoubleFloat,Integer) -> Result}

\texttt{--R e01sef : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Integer,DoubleFloat,DoubleFloat,Integer) -> Result}

\texttt{--R e01sff : (Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),DoubleFloat,Matrix(DoubleFloat),DoubleFloat,DoubleFloat,Integer) -> Result}

\texttt{)clear all}

\texttt{)showArrayValues true}

\texttt{--R}

\texttt{--R (1) true Type: Boolean}

\texttt{--E 2}

\texttt{)showScalarValues true}

\texttt{--R}

\texttt{--R (2) true Type: Boolean}

\texttt{--E 3}

\texttt{)m:=7}

\texttt{m:=7}
--R
--R (3) 7
--R
--E 4

--S 5 of 97
x:Matrix SF:= [[0.0 ,0.2 ,0.4 ,0.6 ,0.75 ,0.9 ,1.0 ]]  
--R
--R
--R (4) [0. 0.2 0.4 0.6 0.75 0.9 1.]
--R  Type: Matrix(DoubleFloat)
--E 5

--S 6 of 97
y:Matrix SF:= [[1.0000 ,1.2214 ,1.4918 ,1.8221 ,2.1170 ,2.4596 ,2.7183 ]]  
--R
--R
--R (5) [1. 1.2214 1.4918 1.8221 2.1170 2.4596 2.7183]
--R  Type: Matrix(DoubleFloat)
--E 6

--S 7 of 97
lck:=11
--R
--R
--R (6) 11
--R  Type: PositiveInteger
--E 7

--S 8 of 97
lwrk:=58
--R
--R
--R (7) 58
--R  Type: PositiveInteger
--E 8

--S 9 of 97
 result:=e01baf(m,x,y,lck,lwrk,-1)
--E 9
)
clear all

--S 10 of 97
showArrayValues true
--R
--R
--R (1) true
--R  Type: Boolean
n:=9

f:Matrix SF:= [[0.00000e+0, 0.27643e-4, 0.43750e-1, 0.16918, 0.46943, 0.94374, 0.99864, 0.99992, 0.99999, ...]]

result:=e01bef(n,x,f,-1)

)clear all
--S 17 of 97
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 17

--S 18 of 97
n:=9
--R
--R
--R (3) 9
--R Type: PositiveInteger
--E 18

--S 19 of 97
x:Matrix SF:= [[7.99,8.09,8.19,8.70,9.20,10.00,12.00,15.00,20.00]]
--R
--R
--R (4) [7.99,8.09,8.19,8.7,9.2,10.0,12.0,15.0,20.0]
--R Type: Matrix(DoubleFloat)
--E 19

--S 20 of 97
f:Matrix SF:= [[0.00000e+0,0.27643e-4,0.43750e-1,0.16918, _, 0.46943,0.94374,0.99864,0.99992,0.99999]]
--R
--R
--R (5)
--R [[0.,2.7643E-5,4.375E-2,0.16918,0.46943,0.94374,0.99864,0.99992,0.99999]]
--R Type: Matrix(DoubleFloat)
--E 20

--S 21 of 97
d:Matrix SF:= [[0.00000e+0,5.52510e-4,0.33587,0.34944, _, 0.59696,0.03260e-2,8.98335e-4,2.93954e-5,0.00000]]
--R
--R
--R (6)
--R [[0.,5.5251E-4,0.33587,0.34944,0.59696,6.0326E-2,8.98335E-4,2.93954E-5,0.0]]
--R Type: Matrix(DoubleFloat)
--E 21

--S 22 of 97
m:=11
--R
--R
--R (7) 11
px:Matrix SF:=
[[7.990000000000002 + (i-1) * 1.2010000000000001 for i in 1..11]]

result:=e01bff(n,x,f,d,m,px,-1)

n:=9

x:Matrix SF:= [[7.99 ,8.09 ,8.19 ,8.70 ,9.20 ,10.00 ,12.00 ,15.00 ,20.00 ]]

showArrayValues true

showScalarValues true
--R
--R (4) [7.99 8.09 8.19 8.7 9.2 10. 12. 15. 20.]
--R
--E 28

--S 29 of 97
f:Matrix SF:= [[0.00000e+0, 0.27643e-4, 0.43750e-1, 0.16918, 0.46943, _, 0.94374, 0.99864, 0.99992, 0.99999]]
--R
--R
--R (5)
--R [[0., 2.7643E-5, 4.375E-2, 0.16918, 0.46943, 0.94374, 0.99864, 0.99992, 0.99999]]
--R
--E 29

--S 30 of 97
d:Matrix SF:= [[0.00000e+0, 5.52510e-4, 0.33587, 0.34944, 0.59696, 6.03260e-2, 8.98335e-4, 2.93954e-5, 0.00000]]
--R
--R
--R (6)
--R [[0., 5.5251E-4, 0.33587, 0.34944, 0.59696, 6.0326E-2, 8.98335E-4, 2.93954E-5, 0.00000]]
--R
--E 30

--S 31 of 97
m:=11
--R
--R
--R (7) 11
--R
--E 31

--S 32 of 97
px:Matrix SF:=
[[7.9900000000000002 + (i-1) *1.2010000000000001 for i in 1..11]]
--R
--R
--R (8)
--R [
--R ]
--R
--E 32

--S 33 of 97
result:=e01bgf(n,x,f,d,m,px,-1)
--E 33
clear all

--S 34 of 97
showArrayValues true
--R
--R
--R (1) true
--R

Type: Boolean
--E 34

--S 35 of 97
showScalarValues true
--R
--R
--R (2) true
--R

Type: Boolean
--E 35

--S 36 of 97
n:=9
--R
--R
--R (3) 9
--R

Type: PositiveInteger
--E 36

--S 37 of 97
x:Matrix SF:= [[7.99 ,8.09 ,8.19 ,8.70 ,9.20 ,10.00 ,12.00 ,15.00 ,20.00 ]]
--R
--R
--R (4) [7.99 8.09 8.19 8.7 9.2 10. 12. 15. 20.]
--R

Type: Matrix(DoubleFloat)
--E 37

--S 38 of 97
f:Matrix SF:= [[0.00000e+0 ,0.27643e-4 ,0.43750e-1 ,0.16918 ,_ 0.46943 ,0.94374 ,0.99864 ,0.99992 ,0.99999 ]]
--R
--R
--R (5)
--R [ [0.,2.7643E-5,4.375E-2,0.16918,0.46943,0.94374,0.99864,0.99992,0.99999]]
--R

Type: Matrix(DoubleFloat)
--E 38

--S 39 of 97
d:Matrix SF:= [[0.00000e+0 ,5.52510e-4 ,0.33587 ,0.34944 ,_ 0.59696 ,6.03260e-2 ,8.98335e-4 ,2.93954e-5 ,0.00000 ]]
--R
--R
a := 7.99
b := 20.0
result := e01bhf(n, x, f, d, a, b, -1)
)}clear all

showArrayValues true

showScalarValues true

mx := 7

result := e01bhf(n, x, f, d, a, b, -1)
my:=6

\[ x: \text{Matrix SF:= \{[1.00, 1.10, 1.30, 1.50, 1.60, 1.80, 2.00]\}} \]

\[ y: \text{Matrix SF:= \{[0.00, 0.10, 0.40, 0.70, 0.90, 1.00]\}} \]

\[ f: \text{Matrix SF:= \{[1.00, 1.10, 1.40, 1.70, 1.90, 2.00, 1.21, 1.31, 1.61, 1.91, 2.11, 2.21, 1.69, 1.79, 2.09, 2.39, 2.59, 2.69, 2.25, 2.35, 2.65, 2.95, 3.15, 3.25, 2.56, 2.66, 2.96, 3.26, 3.46, 3.56, 3.24, 3.34, 3.64, 3.94, 4.14, 4.24, 4.00, 4.10, 4.40, 4.70, 4.90, 5.00]\}} \]

result:=e01daf(mx,my,x,y,f, -1)
clear all

--S 51 of 97
showArrayValues true
--R
--R
--R (1) true
--R
Type: Boolean
--E 51

--S 52 of 97
showScalarValues true
--R
--R
--R (2) true
--R
Type: Boolean
--E 52

--S 53 of 97
m:=30
--R
--R
--R (3) 30
--R
Type: PositiveInteger
--E 53

--S 54 of 97
x:Matrix SF:= [[11.16 ,12.85 ,19.85 ,19.72 ,15.91 , 0.00 ,20.87 , _
    3.45 ,14.26 ,17.43 ,22.80 , 7.58 ,25.00 , 0.00 , _
    9.66 , 5.22 ,17.25 ,25.00 ,12.13 ,22.23 ,11.52 , _
    15.20 , 7.54 ,17.32 , 2.14 , 0.51 ,22.69 , 5.47 ,21.67 ,3.31 ]]
--R
--R
--R (4)
--R
[  
--R [11.16, 12.85, 19.85, 19.72, 15.91, 0., 20.87, 3.45, 14.26, 17.43, 22.8,  
--R 17.32, 2.14, 0.51, 22.69, 5.47, 21.67, 3.31]  
--R ]  
--R
Type: Matrix(DoubleFloat)
--E 54

--S 55 of 97
y:Matrix SF:= [[1.24 ,3.06 ,10.72 ,1.39 ,7.74 ,20.00 ,20.00 ,12.78 , _
    17.87 ,3.46 ,12.39 ,1.98 ,11.87 ,0.00 ,20.00 ,14.66 , _
    19.57 ,3.87 ,10.79 ,6.21 ,8.53 ,0.0 ,10.69 ,13.78 , _
    15.03 ,8.37 ,19.63 ,17.13 ,14.36 ,0.33 ]]
--R
f: Matrix SF := [[22.15, 22.11, 7.97, 16.83, 15.30, 34.60, 5.74, 41.24, 10.74, 18.6, 5.47, 29.87, 4.4, 58.2, 4.73, 40.36, 6.43, 8.74, 13.71, 10.25, 15.74, 19.31, 12.11, 53.1, 49.43, 3.25, 28.63, 5.52, 44.08]]

px := 7.71

py := 3.45

resb := e01sbf(m, x, y, f, resa."triang", resa."grads", px, py, -1)
clear all

showArrayValues true

showScalarValues true

m:=30

x:Matrix SF:=

y:Matrix SF:=


```
[[1.24 ,3.06 ,10.72 ,1.39 ,7.74 ,20.00 ,20.00 ,12.78 ,-
17.87 ,3.46 ,12.39 ,1.98 ,11.87 ,0.00 ,20.00 ,14.66 ,-
19.57 ,3.87 ,10.79 ,6.21 ,8.53 ,0.0 ,10.69 ,13.78 ,-
15.03 ,8.37 ,19.63 ,17.13 ,14.36 ,0.33 ]]
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
result:=resb:=e01sbf(m,x,y,f,resa."triang",resa."grads",px,py,-1)

resb."pf"

)clear all

showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean

m:=30

(3) 30

Type: PositiveInteger

x:Matrix SF:=

\[ \begin{bmatrix} 12.78 & 17.87 & 3.46 & 12.39 & 1.98 & 11.87 & 0.00 \\ 20.00 & 14.66 & 19.57 & 3.87 & 10.79 & 6.21 & 8.53 \\ 0.0 & 10.69 & 13.78 & 15.03 & 8.37 & 19.63 & 17.13 \\ 14.36 & 0.33 \end{bmatrix} \]

\[ \begin{bmatrix} 22.15 & 22.11 & 7.97 & 16.83 & 15.30 & 34.60 & 5.74 \\ 41.24 & 10.74 & 18.60 & 5.47 & 29.87 & 4.40 & 58.20 \\ 4.73 & 40.36 & 6.43 & 8.74 & 13.71 & 10.25 & 15.74 \\ 21.60 & 19.31 & 12.11 & 53.10 & 49.43 & 3.25 & 28.63 \\ 5.52 & 44.08 \end{bmatrix} \]

\[ \begin{bmatrix} 22.15, 22.11, 7.97, 16.83, 15.30, 34.60, 5.74, 41.24, 10.74, 18.60, 5.47, \\ 29.87, 4.40, 58.20, 4.73, 40.36, 6.43, 8.74, 13.71, 10.25, 15.74, 21.60, \\ 19.31, 12.11, 53.10, 49.43, 3.25, 28.63, 5.52, 44.08 \end{bmatrix} \]
nq:=0
--R
--R
--R (8) 0
--R Type: NonNegativeInteger
--E 80

--S 81 of 97
rnw:=0.0
--R
--R
--R (9) 0.0
--R Type: Float
--E 81

--S 82 of 97
rnq:=0.0
--R
--R
--R (10) 0.0
--R Type: Float
--E 82

--S 83 of 97
result:=e01sef(m,x,y,f,nw,nq,rnw,rnq, -1)
--E 83

)clear all

--S 84 of 97
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 84

--S 85 of 97
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 85

--S 86 of 97
m:=30
--R
--R
\[
\begin{array}{cccccccc}
11.16 & 12.85 & 19.85 & 19.72 & 15.91 & 0.00 & 20.87 & - \\
3.45 & 14.26 & 17.43 & 22.80 & 7.58 & 25.00 & 0.00 & - \\
9.66 & 5.22 & 17.25 & 25.00 & 12.13 & 22.23 & 11.52 & - \\
15.20 & 7.54 & 17.32 & 2.14 & 0.51 & 22.69 & 5.47 & - \\
21.67 & 3.31 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
11.16, 12.85, 19.85, 19.72, 15.91, 0., 20.87, 3.45, 14.26, 17.43, 22.8, \\
17.32, 2.14, 0.51, 22.69, 5.47, 21.67, 3.31 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1.24 & 3.06 & 10.72 & 1.39 & 7.74 & 20.00 & 20.00 & - \\
12.78 & 17.87 & 3.46 & 12.39 & 1.98 & 11.87 & 0.00 & - \\
0.0 & 10.69 & 13.78 & 15.03 & 8.37 & 19.63 & 17.13 & - \\
14.36 & 0.33 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
1.24, 3.06, 10.72, 1.39, 7.74, 20., 20., 12.78, 17.87, 3.46, 12.39, 1.98, \\
11.87, 0., 20., 14.66, 19.57, 3.87, 10.79, 6.21, 8.53, 0., 10.69, 13.78, \\
15.03, 8.37, 19.63, 17.13, 14.36, 0.33 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
22.15 & 22.11 & 7.97 & 16.83 & 15.30 & 34.60 & 5.74 & - \\
41.24 & 10.74 & 18.60 & 5.47 & 29.87 & 4.40 & 58.20 & - \\
4.73 & 40.36 & 6.43 & 8.74 & 13.71 & 10.25 & 15.74 & - \\
21.60 & 19.31 & 12.11 & 53.10 & 49.43 & 3.25 & 28.63 & - \\
5.52 & 44.08 & \\
\end{array}
\]

\[
\begin{array}{cccccccc}
22.15, 22.11, 7.97, 16.83, 15.30, 34.60, 5.74, \\
41.24, 10.74, 18.60, 5.47, 29.87, 4.40, 58.20, \\
4.73, 40.36, 6.43, 8.74, 13.71, 10.25, 15.74, \\
21.60, 19.31, 12.11, 53.10, 49.43, 3.25, 28.63, \\
5.52, 44.08 & \\
\end{array}
\]
```
--R [22.15, 22.11, 7.97, 16.83, 15.3, 34.6, 5.74, 41.24, 10.74, 18.6, 5.47,
--R 29.87, 4.4, 58.2, 4.73, 40.36, 6.43, 8.74, 13.71, 10.25, 15.74, 21.6,
--R 19.31, 12.11, 53.1, 49.43, 3.25, 28.63, 5.52, 44.08]
--R ]
--R Type: Matrix(DoubleFloat)
--E 89

--S 90 of 97
nw:=0
--R
--R
--R (7) 0
--R Type: NonNegativeInteger
--E 90

--S 91 of 97
nq:=0
--R
--R
--R (8) 0
--R Type: NonNegativeInteger
--E 91

--S 92 of 97
rnw:=0.0
--R
--R
--R (9) 0.0
--R Type: Float
--E 92

--S 93 of 97
rnq:=0.0
--R
--R
--R (10) 0.0
--R Type: Float
--E 93

--S 94 of 97
-- resa:=e01sef(m,x,y,f,nw,nq,rnw,rnq, -1)
-- E 94

--S 95 of 97
px:=3.0
--R
--R
--R (11) 3.0
--R Type: Float
```
---E 95

---S 96 of 97
py:=2.0
---R
---R
---R (12) 2.0
---R
---E 96

---S 97 of 97
--- result:=e01sff(m,x,y,f,resa."rnw",resa."fnodes",px ,py,-1)
---E 97

)spool
)lisp (bye)

--- NagInterpolationPackage.help ---

This package uses the NAG Library to calculate the interpolation of a function of one or two variables. When provided with the value of the function (and possibly one or more of its lowest-order derivatives) at each of a number of values of the variable(s), the routines provide either an interpolating function or an interpolated value. For some of the interpolating functions, there are supporting routines to evaluate, differentiate or integrate them.

E01 -- Interpolation

Introduction -- E01

Chapter E01

Interpolation

1. Scope of the Chapter

This chapter is concerned with the interpolation of a function of one or two variables. When provided with the value of the function (and possibly one or more of its lowest-order derivatives) at each of a number of values of the variable(s), the routines provide either an interpolating function or an interpolated value. For some of the interpolating functions, there are supporting routines to evaluate, differentiate or integrate them.

2. Background to the Problems

In motivation and in some of its numerical processes, this chapter has much in common with Chapter E02 (Curve and Surface
Fitting). For this reason, we shall adopt the same terminology and refer to dependent variable and independent variable(s) instead of function and variable(s). Where there is only one independent variable, we shall denote it by \( x \) and the dependent variable by \( y \). Thus, in the basic problem considered in this chapter, we are given a set of distinct values \( x_1, x_2, \ldots, x_m \) and a corresponding set of values \( y_1, y_2, \ldots, y_m \) of \( y \), and we shall describe the problem as being one of interpolating the data points \( (x_r, y_r) \), rather than interpolating a function. In modern usage, however, interpolation can have either of two rather different meanings, both relevant to routines in this chapter. They are

(a) the determination of a function of \( x \) which takes the value \( y_r \) at \( x=x_r \), for \( r=1,2,\ldots,m \) (an interpolating function or interpolant),

(b) the determination of the value (interpolated value or interpolate) of an interpolating function at any given value, say \( x \), of \( x \) within the range of the \( x \) (so as to estimate the value at \( x \) of the function underlying the data).

The latter is the older meaning, associated particularly with the use of mathematical tables. The term 'function underlying the data', like the other terminology described above, is used so as to cover situations additional to those in which the data points have been computed from a known function, as with a mathematical table. In some contexts, the function may be unknown, perhaps representing the dependency of one physical variable on another, say temperature upon time.

Whether the underlying function is known or unknown, the object of interpolation will usually be to approximate it to acceptable accuracy by a function which is easy to evaluate anywhere in some range of interest. Piecewise polynomials such as cubic splines (see Section 2.2 of the E02 Chapter Introduction for definitions of terms in this case), being easy to evaluate and also capable of approximating a wide variety of functions, are the types of function mostly used in this chapter as interpolating functions.

Piecewise polynomials also, to a large extent, avoid the well-known problem of large unwanted fluctuations which can arise when
interpolating a data set with a simple polynomial. Fluctuations can still arise but much less frequently and much less severely than with simple polynomials. Unwanted fluctuations are avoided altogether by a routine using piecewise cubic polynomials having only first derivative continuity. It is designed especially for monotonic data, but for other data still provides an interpolant which increases, or decreases, over the same intervals as the data.

The concept of interpolation can be generalised in a number of ways. For example, we may be required to estimate the value of the underlying function at a value \( x \) outside the range of the data. This is the process of extrapolation. In general, it is a good deal less accurate than interpolation and is to be avoided whenever possible.

Interpolation can also be extended to the case of two independent variables. We shall denote these by \( x \) and \( y \), and the dependent variable by \( f \). Methods used depend markedly on whether or not the data values of \( f \) are given at the intersections of a rectangular mesh in the \((x,y)\)-plane. If they are, bicubic splines (see Section 2.3.2 of the E02 Chapter Introduction) are very suitable and usually very effective for the problem. For other cases, perhaps where the \( f \) values are quite arbitrarily scattered in the \((x,y)\)-plane, polynomials and splines are not at all appropriate and special forms of interpolating function have to be employed. Many such forms have been devised and two of the most successful are in routines in this chapter. They both have continuity in first, but not higher, derivatives.

2.1. References


3. Recommendations on Choice and Use of Routines

3.1. General

Before undertaking interpolation, in other than the simplest cases, the user should seriously consider the alternative of using a routine from Chapter E02 to approximate the data by a polynomial or spline containing significantly fewer coefficients than the corresponding interpolating function. This approach is much less liable to produce unwanted fluctuations and so can often provide a better approximation to the function underlying
When interpolation is employed to approximate either an underlying function or its values, the user will need to be satisfied that the accuracy of approximation achieved is adequate. There may be a means for doing this which is particular to the application, or the routine used may itself provide a means. In other cases, one possibility is to repeat the interpolation using one or more extra data points, if they are available, or otherwise one or more fewer, and to compare the results. Other possibilities, if it is an interpolating function which is determined, are to examine the function graphically, if that gives sufficient accuracy, or to observe the behaviour of the differences in a finite-difference table, formed from evaluations of the interpolating function at equally-spaced values of $x$ over the range of interest. The spacing should be small enough to cause the typical size of the differences to decrease as the order of difference increases.

3.2. One Independent Variable

E01BAF computes an interpolating cubic spline, using a particular choice for the set of knots which has proved generally satisfactory in practice. If the user wishes to choose a different set, a cubic spline routine from Chapter E02, namely E02BAF, may be used in its interpolating mode, setting $\text{NCAP7} = M+4$ and all elements of the parameter $W$ to unity. These routines provide the interpolating function in B-spline form (see Section 2.2.2 in the E02 Chapter Introduction). Routines for evaluating, differentiating and integrating this form are discussed in Section 3.7 of the E02 Chapter Introduction.

The cubic spline does not always avoid unwanted fluctuations, especially when the data show a steep slope close to a region of small slope, or when the data inadequately represent the underlying curve. In such cases, E01BEF can be very useful. It derives a piecewise cubic polynomial (with first derivative continuity) which, between any adjacent pair of data points, either increases all the way, or decreases all the way (or stays constant). It is especially suited to data which are monotonic over their whole range.

In this routine, the interpolating function is represented simply by its value and first derivative at the data points. Supporting routines compute its value and first derivative elsewhere, as well as its definite integral over an arbitrary interval.

3.3. Two Independent Variables

3.3.1. Data on a rectangular mesh
Given the value \( f \) of the dependent variable \( f \) at the point \( (x_q, y_r) \) in the plane of the independent variables \( x \) and \( y \), for each \( q=1,2,\ldots,m \) and \( r=1,2,\ldots,n \) (so that the points \( (x_q, y_r) \) lie at the \( m \times n \) intersections of a rectangular mesh), E01DAF computes an interpolating bicubic spline, using a particular choice for each of the spline's knot-set. This choice, the same as in E01BAF, has proved generally satisfactory in practice. If, instead, the user wishes to specify his own knots, a routine from Chapter E02, namely E02DAF, may be adapted (it is more cumbersome for the purpose, however, and much slower for larger problems). Using \( m \) and \( n \) in the above sense, the parameter \( M \) must be set to \( m \times n \), \( PX \) and \( PY \) must be set to \( m+4 \) and \( n+4 \) respectively and all elements of \( W \) should be set to unity. The recommended value for \( EPS \) is zero.

3.3.2. Arbitrary data

As remarked at the end of Section 2, special types of interpolating are required for this problem, which can often be difficult to solve satisfactorily. Two of the most successful are employed in E01SAF and E01SEF, the two routines which (with their respective evaluation routines E01SBF and E01SFF) are provided for the problem. Definitions can be found in the routine documents. Both interpolants have first derivative continuity and are 'local', in that their value at any point depends only on data in the immediate neighbourhood of the point. This latter feature is necessary for large sets of data to avoid prohibitive computing time.

The relative merits of the two methods vary with the data and it is not possible to predict which will be the better in any particular case.

3.4. Index

Derivative, of interpolant from E01BEF E01BGF
Evaluation, of interpolant
  from E01BEF E01BFF
  from E01SAF E01SBF
  from E01SEF E01SFF
Extrapolation, one variable E01BEF
Integration (definite) of interpolant from E01BEF E01BHF
Interpolated values, one variable, from interpolant from E01BEF E01BFF
  E01BGF
Interpolated values, two variables, from interpolant from E01SAF E01SBF
from interpolant from E01SEF E01SFF
Interpolating function, one variable, cubic spline E01BAF
other piecewise polynomial E01BEF
Interpolating function, two variables bicubic spline E01DAF
other piecewise polynomial E01SAF
modified Shepard method E01SEF

E01 -- Interpolation Contents -- E01
Chapter E01

Interpolation
E01BAF Interpolating functions, cubic spline interpolant, one variable
E01BEF Interpolating functions, monotonicity-preserving, piecewise cubic Hermite, one variable
E01BFF Interpolated values, interpolant computed by E01BEF, function only, one variable,
E01BGF Interpolated values, interpolant computed by E01BEF, function and 1st derivative, one variable
E01BHF Interpolated values, interpolant computed by E01BEF, definite integral, one variable
E01DAF Interpolating functions, fitting bicubic spline, data on rectangular grid
E01SAF Interpolating functions, method of Renka and Cline, two variables
E01SBF Interpolated values, evaluate interpolant computed by E01SAF, two variables
E01SEF Interpolating functions, modified Shepard’s method, two variables
E01SFF Interpolated values, evaluate interpolant computed by E01SEF, two variables

E01 -- Interpolation E01BAF
EO1BAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

EO1BAF determines a cubic spline interpolant to a given set of data.

2. Specification

```fortran
SUBROUTINE EO1BAF (M, X, Y, LAMDA, C, LCK, WRK, LWRK,
  1                 IFAIL)
  INTEGER M, LCK, LWRK, IFAIL
  DOUBLE PRECISION X(M), Y(M), LAMDA(LCK), C(LCK), WRK(LWRK)
```

3. Description

This routine determines a cubic spline \( s(x) \), defined in the range \( x_{\text{min}} \leq x \leq x_{\text{max}} \), which interpolates (passes exactly through) the set of \( m \) data points \((x_i, y_i)\), for \( i=1,2,...,m\), where \( m \geq 4 \) and \( x_{\text{min}} < x_1 < ... < x_m \). The end conditions are not imposed. The spline interpolant chosen has \( m-4 \) interior knots \((\lambda_5, \lambda_6, ..., \lambda_m)\), which are set to the values of \( x_3, x_4, ..., x_{m-2} \) respectively. This spline is represented in its B-spline form (see Cox [1]):

\[
 s(x) = \sum_{i=1}^{m} c_i N_i(x),
\]

where \( N_i(x) \) denotes the normalised B-Spline of degree 3, defined upon the knots \((\lambda_1, \lambda_2, ..., \lambda_m)\), and \( c_1, c_2, ..., c_{m-4} \) denotes its coefficient, whose value is to be determined by the routine.

The use of B-splines requires eight additional knots \((\lambda_1, \lambda_2, ..., \lambda_m)\), which

1. \( \lambda_1 \)
2. \( \lambda_2 \)
3. \( \lambda_3 \)
4. \( \lambda_4 \)
5. \( \lambda_{m+1} \)
6. \( \lambda_{m+2} \)
The algorithm for determining the coefficients is as described in [1] except that QR factorization is used instead of LU decomposition. The implementation of the algorithm involves setting up appropriate information for the related routine E02BAF followed by a call of that routine. (For further details of E02BAF, see the routine document.)

Values of the spline interpolant, or of its derivatives or definite integral, can subsequently be computed as detailed in Section 8.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: m, the number of data points. Constraint: M >= 4.

2: X(M) -- DOUBLE PRECISION array Input
   On entry: X(i) must be set to x , the ith data value of the
   independent variable x, for i=1,2,...,m. Constraint: X(i) <
   X(i+1), for i=1,2,...,M-1.

3: Y(M) -- DOUBLE PRECISION array Input
   On entry: Y(i) must be set to y , the ith data value of the
   dependent variable y, for i=1,2,...,m.

4: LAMDA(LCK) -- DOUBLE PRECISION array Output
   On exit: the value of (lambda) , the ith knot, for
   i=1,2,...,m+4.

5: C(LCK) -- DOUBLE PRECISION array Output
   On exit: the coefficient c of the B-spline N (x), for
   i=1,2,...,m. The remaining elements of the array are not...
used.

6: LCK -- INTEGER Input
On entry: the dimension of the arrays LAMDA and C as declared in the
(sub)program from which E01BAF is called.
Constraint: LCK >= M + 4.

7: WRK(LWRK) -- DOUBLE PRECISION array Workspace

8: LWRK -- INTEGER Input
On entry: the dimension of the array WRK as declared in the
(sub)program from which E01BAF is called.
Constraint: LWRK>=6*M+16.

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
On entry M < 4,
or LCK<M+4,
or LWRK<6*M+16.

IFAIL= 2
The X-values fail to satisfy the condition
X(1) < X(2) < X(3) <... < X(M).

7. Accuracy

The rounding errors incurred are such that the computed spline is
an exact interpolant for a slightly perturbed set of ordinates
y + (delta)y . The ratio of the root-mean-square value of the
i
i
(delta)y to that of the y is no greater than a small multiple
i
i
of the relative machine precision.
8. Further Comments

The time taken by the routine is approximately proportional to m.

All the x are used as knot positions except x and x. This  
\[ x_i, x_{i+1}, \ldots, x_{m-1} \]
choice of knots (see Cox [2]) means that s(x) is composed of m-3  
cubic arcs as follows. If m=4, there is just a single arc spanning the whole interval x to x. If m>=5, the first and last  
arc spans the intervals x to x and x to x respectively.  
\[ x_1, x_2, \ldots, x_{m-2}, x_m \]
Additionally if m>=6, the ith arc, for i=2,3,\ldots,m-4 spans the  
interval x to x.  
\[ x_{i+1}, x_{i+2}, \ldots, x_{i+m-2} \]

After the call

\[ \text{CALL E01BAF} \ (M, X, Y, LAMDA, C, LCK, WRK, LWRK, IFAIL) \]

the following operations may be carried out on the interpolant  
s(x).

The value of s(x) at x = XVAL can be provided in the real  
variable SVAL by the call

\[ \text{CALL E02BBF} \ (M+4, LAMDA, C, XVAL, SVAL, IFAIL) \]

The values of s(x) and its first three derivatives at x = XVAL  
can be provided in the real array SDIF of dimension 4, by the  
call

\[ \text{CALL E02BCF} \ (M+4, LAMDA, C, XVAL, LEFT, SDIF, IFAIL) \]

Here LEFT must specify whether the left- or right-hand value of  
the third derivative is required (see E02BCF for details).

The value of the integral of s(x) over the range x to x can be  
\[ x_1, x_m \]
provided in the real variable SINT by

\[ \text{CALL E02BDF} \ (M+4, LAMDA, C, SINT, IFAIL) \]

9. Example

The example program sets up data from 7 values of the exponential  
function in the interval 0 to 1. E01BAF is then called to compute  
a spline interpolant to these data.

The spline is evaluated by E02BBF, at the data points and at
points halfway between each adjacent pair of data points, and the 
\(x\) spline values and the values of \(e\) are printed out.

The example program is not reproduced here. The source code for 
all example programs is distributed with the NAG Foundation 
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E01 -- Interpolation
E01BEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for 
your implementation to check implementation-dependent details. 
The symbol (*) after a NAG routine name denotes a routine that is 
not included in the Foundation Library.

1. Purpose

E01BEF computes a monotonicity-preserving piecewise cubic Hermite 
interpolant to a set of data points.

2. Specification

    SUBROUTINE E01BEF (N, X, F, D, IFAIL)
    INTEGER N, IFAIL
    DOUBLE PRECISION X(N), F(N), D(N)

3. Description

This routine estimates first derivatives at the set of data 
points \((x_r, f_r)\), for \(r=1,2,\ldots,n\), which determine a piecewise 
\(r\) cubic Hermite interpolant to the data, that preserves 
monotonicity over ranges where the data points are monotonic. If 
the data points are only piecewise monotonic, the interpolant 
will have an extremum at each point where monotonicity switches 
direction. The estimates of the derivatives are computed by a 
formula due to Brodlie, which is described in Fritsch and Butland 
[1], with suitable changes at the boundary points.

The routine is derived from routine PCHIM in Fritsch [2].

Values of the computed interpolant, and of its first derivative 
and definite integral, can subsequently be computed by calling 
E01BFF, E01BGF and E01BHF, as described in Section 8.

4. References
5. Parameters

1: N -- INTEGER Input
   On entry: n, the number of data points. Constraint: N >= 2.

2: X(N) -- DOUBLE PRECISION array Input
   On entry: X(r) must be set to x, the rth value of the
   r independent variable (abscissa), for r=1,2,...,n.
   Constraint: X(r) < X(r+1).

3: F(N) -- DOUBLE PRECISION array Input
   On entry: F(r) must be set to f, the rth value of the
   r dependent variable (ordinate), for r=1,2,...,n.

4: D(N) -- DOUBLE PRECISION array Output
   On exit: estimates of derivatives at the data points. D(r)
   contains the derivative at X(r).

5: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry N < 2.

IFAIL= 2
   The values of X(r), for r=1,2,...,N, are not in strictly
   increasing order.

7. Accuracy
The computational errors in the array D should be negligible in most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to n.

The values of the computed interpolant at the points PX(i), for i=1,2,...,M, may be obtained in the real array PF, of length at least M, by the call:

`CALL E01BFF(N,X,F,D,M,PX,PF,IFAIL)`

where N, X and F are the input parameters to E01BEF and D is the output parameter from E01BEF.

The values of the computed interpolant at the points PX(i), for i = 1,2,...,M, together with its first derivatives, may be obtained in the real arrays PF and PD, both of length at least M, by the call:

`CALL E01BGF(N,X,F,D,M,PX,PF,PD,IFAIL)`

where N, X, F and D are as described above.

The value of the definite integral of the interpolant over the interval A to B can be obtained in the real variable PINT by the call:

`CALL E01BHF(N,X,F,D,A,B,PINT,IFAIL)`

where N, X, F and D are as described above.

9. Example

This example program reads in a set of data points, calls E01BEF to compute a piecewise monotonic interpolant, and then calls E01BFF to evaluate the interpolant at equally spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01BFF evaluates a piecewise cubic Hermite interpolant at a set of points.

2. Specification

```fortran
SUBROUTINE E01BFF (N, X, F, D, M, PX, PF, IFAIL)
INTEGER N, M, IFAIL
DOUBLE PRECISION X(N), F(N), D(N), PX(M), PF(M)
```

3. Description

This routine evaluates a piecewise cubic Hermite interpolant, as computed by E01BEF, at the points PX(i), for i=1,2,...,m. If any point lies outside the interval from X(1) to X(N), a value is extrapolated from the nearest extreme cubic, and a warning is returned.

The routine is derived from routine PCHFE in Fritsch [1].

4. References


5. Parameters

1: N -- INTEGER  
   Input

2: X(N) -- DOUBLE PRECISION array  
   Input

3: F(N) -- DOUBLE PRECISION array  
   Input

4: D(N) -- DOUBLE PRECISION array  
   Input
   On entry: N, X, F and D must be unchanged from the previous call of E01BEF.

5: M -- INTEGER  
   Input
   On entry: m, the number of points at which the interpolant is to be evaluated. Constraint: M >= 1.

6: PX(M) -- DOUBLE PRECISION array  
   Input
   On entry: the m values of x at which the interpolant is to be evaluated.
7: PF(M) -- DOUBLE PRECISION array 
On exit: PF(i) contains the value of the interpolant 
evaluated at the point PX(i), for i=1,2,...,m.

8: IFAIL -- INTEGER 
Input/Output 
On entry: IFAIL must be set to 0, -1 or 1. For users not 
familiar with this parameter (described in the Essential 
Introduction) the recommended value is 0. 
On exit: IFAIL = 0 unless the routine detects an error (see 
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are 
output on the current error message unit (as defined by X04AAF).

IFAIL= 1 
On entry N < 2.

IFAIL= 2 
The values of X(r), for r = 1,2,...,N, are not in strictly 
increasing order.

IFAIL= 3 
On entry M < 1.

IFAIL= 4 
At least one of the points PX(i), for i = 1,2,...,M, lies 
outside the interval [X(1),X(N)], and extrapolation was 
performed at all such points. Values computed at such points 
may be very unreliable.

7. Accuracy

The computational errors in the array PF should be negligible in 
most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to 
the number of evaluation points, m. The evaluation will be most 
efficient if the elements of PX are in non-decreasing order (or, 
more generally, if they are grouped in increasing order of the 
intervals [X(r-1),X(r)]). A single call of E01BFF with m>1 is 
more efficient than several calls with m=1.
9. Example

This example program reads in values of N, X, F and D, and then calls E01BFF to evaluate the interpolant at equally spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E01 -- Interpolation
E01BGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E01BGF evaluates a piecewise cubic Hermite interpolant and its first derivative at a set of points.

2. Specification

```fortran
SUBROUTINE E01BGF (N, X, F, D, M, PX, PF, PD, IFAIL)
INTEGER N, M, IFAIL
DOUBLE PRECISION X(N), F(N), D(N), PX(M), PF(M), PD(M)
```

3. Description

This routine evaluates a piecewise cubic Hermite interpolant, as computed by E01BEF, at the points PX(i), for i=1,2,...,m. The first derivatives at the points are also computed. If any point lies outside the interval from X(1) to X(N), values of the interpolant and its derivative are extrapolated from the nearest extreme cubic, and a warning is returned.

If values of the interpolant only, and not of its derivative, are required, E01BFF should be used.

The routine is derived from routine PCHIP in Fritsch [1].

4. References

5. Parameters

1: \( N \) -- INTEGER
   Input

2: \( X(N) \) -- DOUBLE PRECISION array
   Input

3: \( F(N) \) -- DOUBLE PRECISION array
   Input

4: \( D(N) \) -- DOUBLE PRECISION array
   Input
   On entry: \( N, X, F \) and \( D \) must be unchanged from the previous
   call of E01BEF.

5: \( M \) -- INTEGER
   Input
   On entry: \( m \), the number of points at which the interpolant
   is to be evaluated. Constraint: \( M \geq 1 \).

6: \( PX(M) \) -- DOUBLE PRECISION array
   Input
   On entry: the \( m \) values of \( x \) at which the interpolant is to
   be evaluated.

7: \( PF(M) \) -- DOUBLE PRECISION array
   Output
   On exit: \( PF(i) \) contains the value of the interpolant
   evaluated at the point \( PX(i) \), for \( i=1,2,\ldots,m \).

8: \( PD(M) \) -- DOUBLE PRECISION array
   Output
   On exit: \( PD(i) \) contains the first derivative of the
   interpolant evaluated at the point \( PX(i) \), for \( i=1,2,\ldots,m \).

9: \( IFAIL \) -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry \( N < 2 \).

IFAIL= 2
   The values of \( X(r) \), for \( r = 1,2,\ldots,N \), are not in strictly
   increasing order.
IFAIL = 3
On entry M < 1.

IFAIL = 4
At least one of the points PX(i), for i = 1, 2, ..., M, lies outside the interval [X(1), X(N)], and extrapolation was performed at all such points. Values computed at these points may be very unreliable.

7. Accuracy
The computational errors in the arrays PF and PD should be negligible in most practical situations.

8. Further Comments
The time taken by the routine is approximately proportional to the number of evaluation points, m. The evaluation will be most efficient if the elements of PX are in non-decreasing order (or, more generally, if they are grouped in increasing order of the intervals [X(r-1), X(r)]). A single call of E01BGF with m>1 is more efficient than several calls with m=1.

9. Example
This example program reads in values of N, X, F and D, and calls E01BGF to compute the values of the interpolant and its derivative at equally spaced points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E01 -- Interpolation
E01BHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
E01BHF evaluates the definite integral of a piecewise cubic Hermite interpolant over the interval [a, b].

2. Specification
SUBROUTINE E01BHF (N, X, F, D, A, B, PINT, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N), F(N), D(N), A, B, PINT

3. Description

This routine evaluates the definite integral of a piecewise cubic Hermite interpolant, as computed by E01BEF, over the interval [a,b].

If either a or b lies outside the interval from X(1) to X(N) computation of the integral involves extrapolation and a warning is returned.

The routine is derived from routine PCHIA in Fritsch [1].

4. References


5. Parameters

1: N -- INTEGER Input
2: X(N) -- DOUBLE PRECISION array Input
3: F(N) -- DOUBLE PRECISION array Input
4: D(N) -- DOUBLE PRECISION array Input
   On entry: N, X, F and D must be unchanged from the previous call of E01BEF.
5: A -- DOUBLE PRECISION Input
6: B -- DOUBLE PRECISION Input
   On entry: the interval [a,b] over which integration is to be performed.
7: PINT -- DOUBLE PRECISION Output
   On exit: the value of the definite integral of the interpolant over the interval [a,b].
8: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).
6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry N < 2.

IFAIL = 2
The values of X(r), for r = 1,2,...,N, are not in strictly increasing order.

IFAIL = 3
On entry at least one of A or B lies outside the interval [X(1),X(N)], and extrapolation was performed to compute the integral. The value returned is therefore unreliable.

7. Accuracy

The computational error in the value returned for PINT should be negligible in most practical situations.

8. Further Comments

The time taken by the routine is approximately proportional to the number of data points included within the interval [a,b].

9. Example

This example program reads in values of N, X, F and D. It then reads in pairs of values for A and B, and evaluates the definite integral of the interpolant over the interval [A,B] until end-of-file is reached.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

E01DAF computes a bicubic spline interpolating surface through a set of data values, given on a rectangular grid in the x-y plane.

2. Specification

```plaintext
SUBROUTINE E01DAF (MX, MY, X, Y, F, PX, PY, LAMDA, MU, C,
1 WRK, IFAIL)
INTEGER MX, MY, PX, PY, IFAIL
DOUBLE PRECISION X(MX), Y(MY), F(MX*MY), LAMDA(MX+4), MU(MX
1 +4), C(MX*MY), WRK((MX+6)*(MY+6))
```

3. Description

This routine determines a bicubic spline interpolant to the set of data points \((x_{qr}, y_{qr}, f_{qr})\), for \(q=1,2,...,m\); \(r=1,2,...,m\). The spline is given in the B-spline representation

\[
s(x, y) = \sum_{i=1}^{m} \sum_{j=1}^{m} c_{ij} M_i(x) N_j(y),
\]

such that

\[
s(x_{qr}, y_{qr}) = f_{qr},
\]

where \(M_i(x)\) and \(N_j(y)\) denote normalised cubic B-splines, the former defined on the knots \((\lambda_i)\) to \((\lambda_{i+4})\) and the latter on the knots \((\mu_j)\) to \((\mu_{j+4})\), and the \(c_{ij}\) are the spline coefficients. These knots, as well as the coefficients, are determined by the routine, which is derived from the routine B2IRE in Anthony et al[1]. The method used is described in Section 8.2.


Values of the computed spline can subsequently be obtained by calling E02DEF or E02DFF as described in Section 8.3.
4. References


5. Parameters

1: MX -- INTEGER Input  
2: MY -- INTEGER Input  
On entry: MX and MY must specify m and m respectively, the number of points along the x and y axis that define the rectangular grid. Constraint: MX >= 4 and MY >= 4.

3: X(MX) -- DOUBLE PRECISION array Input  
4: Y(MY) -- DOUBLE PRECISION array Input  
On entry: X(q) and Y(r) must contain x , for q=1,2,...,m , and y , for r=1,2,...,m , respectively. Constraints:  
X(q) < X(q+1), for q=1,2,...,m -1, 
Y(r) < Y(r+1), for r=1,2,...,m -1.

5: F(MX*MY) -- DOUBLE PRECISION array Input  
On entry: F(m *(q-1)+r) must contain f , for q=1,2,...,m ; 
r=1,2,...,m .

6: PX -- INTEGER Output  
7: PY -- INTEGER Output  
On exit: PX and PY contain m +4 and m +4, the total number
of knots of the computed spline with respect to the x and y variables, respectively.

8: LAMDA(MX+4) -- DOUBLE PRECISION array Output

9: MU(MY+4) -- DOUBLE PRECISION array Output

On exit: LAMDA contains the complete set of knots (lambda) associated with the x variable, i.e., the interior knots LAMDA(5), LAMDA(6), ..., LAMDA(PX-4), as well as the additional knots LAMDA(1) = LAMDA(2) = LAMDA(3) = LAMDA(4) = X(1) and LAMDA(PX-3) = LAMDA(PX-2) = LAMDA(PX-1) = LAMDA(PX) = X(MX) needed for the B-spline representation. MU contains the corresponding complete set of knots (mu) associated with the y variable.

10: C(MX*MY) -- DOUBLE PRECISION array Output

On exit: the coefficients of the spline interpolant. C(m *(i-1)+j) contains the coefficient c described in Section 3.

11: WRK((MX+6)*(MY+6)) -- DOUBLE PRECISION array Workspace

12: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1

On entry MX < 4,

or MY < 4.

IFAIL= 2

On entry either the values in the X array or the values in the Y array are not in increasing order.

IFAIL= 3
A system of linear equations defining the B-spline coefficients was singular; the problem is too ill-conditioned to permit solution.

7. Accuracy

The main sources of rounding errors are in steps (2), (3), (6) and (7) of the algorithm described in Section 8.2. It can be shown (Cox [2]) that the matrix $A$ formed in step (2) has

$$x$$

elements differing relatively from their true values by at most a small multiple of $3(\epsilon)$, where $\epsilon$ is the machine precision. $A$ is 'totally positive', and a linear system with

$$x$$
such a coefficient matrix can be solved quite safely by elimination without pivoting. Similar comments apply to steps (6) and (7). Thus the complete process is numerically stable.

8. Further Comments

8.1. Timing

The time taken by this routine is approximately proportional to $m m$.

$$x y$$

8.2. Outline of method used

The process of computing the spline consists of the following steps:

(1) choice of the interior $x$-knots $(\lambda_5, \lambda_6, \ldots, \lambda_m)$ with $\lambda_i = x_{i-2}$ for $i = 5, 6, \ldots, m$.

(2) formation of the system

$$A E = F,$$

where $A$ is a band matrix of order $m$ and bandwidth 4, containing in its $q$th row the values at $x$ of the $B$-splines in $x$, $F$ is the $m$ by $m$ rectangular matrix of values $f_{q,r}$, and $E$ denotes an $m$ by $m$ rectangular matrix of intermediate coefficients,
(3) use of Gaussian elimination to reduce this system to band triangular form,

(4) solution of this triangular system for \( E \),

(5) choice of the interior \( y \) knots \((\mu)_j, (\mu)_6, \ldots, (\mu)_m\) as

\[
(\mu)_i = y_{i-2}, \text{ for } i = 5, 6, \ldots, m,
\]

(6) formation of the system

\[
 \begin{align*}
 T & \quad T \\
 A & \quad C = E, \\
 y & \quad y
\end{align*}
\]

where \( A \) is the counterpart of \( A \) for the \( y \) variable, and \( C \) denotes the \( m \times m \) rectangular matrix of values of \( c_{ij} \),

(7) use of Gaussian elimination to reduce this system to band triangular form,

(8) solution of this triangular system for \( C \) and hence \( C \).

For computational convenience, steps (2) and (3), and likewise steps (6) and (7), are combined so that the formation of \( A \) and \( C \) and the reductions to triangular form are carried out one row \( y \) at a time.

8.3. Evaluation of Computed Spline

The values of the computed spline at the points \((TX(r), TY(r))\), for \( r = 1, 2, \ldots, N \), may be obtained in the double precision array \( FF \), of length at least \( N \), by the following call:

\[
\begin{align*}
 IFAIL &= 0 \\
 CALL \ E02DEF(N, PX, PY, TX, TY, LAMDA, MU, C, FF, WRK, IWRK, IFAIL)
\end{align*}
\]

where \( PX, PY, LAMDA, MU \) and \( C \) are the output parameters of \( E01DAF \), \( WRK \) is a double precision workspace array of length at least \( PY-4 \), and \( IWRK \) is an integer workspace array of length at least \( PY-4 \).

To evaluate the computed spline on an \( NX \) by \( NY \) rectangular grid of points in the \( x-y \) plane, which is defined by the \( x \) co-
ordinates stored in TX(q), for q = 1,2,...,NX, and the y co-
ordinates stored in TY(r), for r = 1,2,...,NY, returning the
results in the double precision array FG which is of length at
least NX*NY, the following call may be used:

IFAIL = 0
CALL E02DFF(NX,NY,PX,PY,TX,TY,LAMDA,MU,C,FG,WRK,LWRK,
          * IWRK,LIWRK,IFAIL)

where PX, PY, LAMDA, MU and C are the output parameters of E01DAF
, WRK is a double precision workspace array of length at least
LWRK = min(NWRK1,NWRK2), NWRK1 = NX*4+PX, NWRK2 = NY*4+PY, and
IWRK is an integer workspace array of length at least LIWRK = NY
+ PY - 4 if NWRK1 > NWRK2, or NX + PX - 4 otherwise. The result
of the spline evaluated at grid point (q,r) is returned in
element (NY*(q-1)+r) of the array FG.

9. Example

This program reads in values of m, x for q=1,2,...,m, and
x y
y for r=1,2,...,m, followed by values of the ordinates f
r x q y
defined at the grid points (x,y). It then calls E01DAF to
q r
compute a bicubic spline interpolant of the data values, and
prints the values of the knots and B-spline coefficients. Finally
it evaluates the spline at a small sample of points on a
rectangular grid.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
2. Specification

SUBROUTINE E01SAF (M, X, Y, F, TRIANG, GRADS, IFAIL)
INTEGER M, TRIANG(7*M), IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), GRADS(2,M)

3. Description

This routine constructs an interpolating surface \( F(x,y) \) through a set of \( m \) scattered data points \((x_r, y_r, f_r)\), for \( r=1,2,...,m \), using a method due to Renka and Cline. In the \((x,y)\) plane, the data points must be distinct. The constructed surface is continuous and has continuous first derivatives.

The method involves firstly creating a triangulation with all the \((x,y)\) data points as nodes, the triangulation being as nearly equiangular as possible (see Cline and Renka [1]). Then gradients in the \( x \) and \( y \)-directions are estimated at node \( r \), for \( r=1,2,...,m \), as the partial derivatives of a quadratic function of \( x \) and \( y \) which interpolates the data value \( f_r \), and which fits the data values at nearby nodes (those within a certain distance chosen by the algorithm) in a weighted least-squares sense. The weights are chosen such that closer nodes have more influence than more distant nodes on derivative estimates at node \( r \). The computed partial derivatives, with the \( f \) values, at the three nodes of each triangle define a piecewise polynomial surface of a certain form which is the interpolant on that triangle. See Renka and Cline [4] for more detailed information on the algorithm, a development of that by Lawson [2]. The code is derived from Renka [3].

The interpolant \( F(x,y) \) can subsequently be evaluated at any point \((x,y)\) inside or outside the domain of the data by a call to E01SBF. Points outside the domain are evaluated by extrapolation.

4. References


5. Parameters

1:  M -- INTEGER Input
    On entry: m, the number of data points. Constraint: M >= 3.

2:  X(M) -- DOUBLE PRECISION array Input

3:  Y(M) -- DOUBLE PRECISION array Input

4:  F(M) -- DOUBLE PRECISION array Input
    On entry: the co-ordinates of the rth data point, for r=1,2,...,m. The data points are accepted in any order, but see Section 8. Constraint: The (x,y) nodes must not all be collinear, and each node must be unique.

5:  TRIANG(7*M) -- INTEGER array Output
    On exit: a data structure defining the computed triangulation, in a form suitable for passing to E01SBF.

6:  GRADS(2,M) -- DOUBLE PRECISION array Output
    On exit: the estimated partial derivatives at the nodes, in a form suitable for passing to E01SBF. The derivatives at node r with respect to x and y are contained in GRADS(1,r) and GRADS(2,r) respectively, for r=1,2,...,m.

7:  IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
    On entry M < 3.
CHAPTER 15. CHAPTER N

IFAIL = 2
On entry all the (X,Y) pairs are collinear.

IFAIL = 3
On entry (X(i),Y(i)) = (X(j),Y(j)) for some i ≠ j.

7. Accuracy

On successful exit, the computational errors should be negligible in most situations but the user should always check the computed surface for acceptability, by drawing contours for instance. The surface always interpolates the input data exactly.

8. Further Comments

The time taken for a call of E01SAF is approximately proportional to the number of data points, m. The routine is more efficient if, before entry, the values in X, Y, F are arranged so that the X array is in ascending order.

9. Example

This program reads in a set of 30 data points and calls E01SAF to construct an interpolating surface. It then calls E01SBF to evaluate the interpolant at a sample of points on a rectangular grid.

Note that this example is not typical of a realistic problem: the number of data points would normally be larger, and the interpolant would need to be evaluated on a finer grid to obtain an accurate plot, say.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
2. Specification

SUBROUTINE E01SBF (M, X, Y, F, TRIANG, GRADS, PX, PY, PF, IFAIL)
INTEGER M, TRIANG(7*M), IFAIL
DOUBLE PRECISION X(M), Y(M), F(M), GRADS(2,M), PX, PY, PF

3. Description

This routine takes as input the parameters defining the interpolant \( F(x,y) \) of a set of scattered data points \( (x_r, y_r, f_r) \), for \( r=1,2,...,m \), as computed by E01SAF, and evaluates the interpolant at the point \( (px,py) \).

If \( (px,py) \) is equal to \( (x_r, y_r) \) for some value of \( r \), the returned value will be equal to \( f_r \).

If \( (px,py) \) is not equal to \( (x_r, y_r) \) for any \( r \), the derivatives in \( GRADS \) will be used to compute the interpolant. A triangle is sought which contains the point \( (px,py) \), and the vertices of the triangle along with the partial derivatives and \( f \) values at the vertices are used to compute the value \( F(px,py) \). If the point \( (px,py) \) lies outside the triangulation defined by the input parameters, the returned value is obtained by extrapolation. In this case, the interpolating function \( F \) is extended linearly beyond the triangulation boundary. The method is described in more detail in Renka and Cline [2] and the code is derived from Renka [1].

E01SBF must only be called after a call to E01SAF.

4. References


5. Parameters

1: M -- INTEGER
   Input
2: X(M) -- DOUBLE PRECISION array  
   Input
3: Y(M) -- DOUBLE PRECISION array  
   Input
4: F(M) -- DOUBLE PRECISION array  
   Input
5: TRIANG(7*M) -- INTEGER array  
   Input
   On entry: M, X, Y, F, TRIANG and GRADS must be unchanged from the previous call of E01SAF.
6: GRADS(2,M) -- DOUBLE PRECISION array  
   Input
   On entry: the point (px,py) at which the interpolant is to be evaluated.
7: PX -- DOUBLE PRECISION  
   Input
8: PY -- DOUBLE PRECISION  
   Input
9: PF -- DOUBLE PRECISION  
   Output
   On exit: the value of the interpolant evaluated at the point (px,py).
10: IFAIL -- INTEGER  
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1  
On entry M < 3.

IFAIL= 2  
On entry the triangulation information held in the array TRIANG does not specify a valid triangulation of the data points. TRIANG may have been corrupted since the call to E01SAF.

IFAIL= 3  
The evaluation point (PX,PY) lies outside the nodal triangulation, and the value returned in PF is computed by
extrapolation.

7. Accuracy

Computational errors should be negligible in most practical situations.

8. Further Comments

The time taken for a call of E01SBF is approximately proportional to the number of data points, m.

The results returned by this routine are particularly suitable for applications such as graph plotting, producing a smooth surface from a number of scattered points.

9. Example

See the example for E01SAF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
set of \( m \) scattered data points \((x_i, y_i, f_i)\), for \( r=1,2,\ldots,m\), using a modification of Shepard’s method. The surface is continuous and has continuous first derivatives.

The basic Shepard method, described in [2], interpolates the input data with the weighted mean

\[
F(x,y) = \frac{\sum_{r=1}^{m} w(x,y) f_r}{\sum_{r=1}^{m} w(x,y)},
\]

where \( w(x,y) = \frac{1}{d^{2}} \) and \( d = (x-x_r)^2 + (y-y_r)^2 \).

The basic method is global in that the interpolated value at any point depends on all the data, but this routine uses a modification due to Franke and Nielson described in [1], whereby the method becomes local by adjusting each \( w(x,y) \) to be zero outside a circle with centre \((x_r, y_r)\) and some radius \( R \). Also, to improve the performance of the basic method, each \( f_r \) above is replaced by a function \( f(x,y) \), which is a quadratic fitted by weighted least-squares to data local to \((x_r, y_r)\) and forced to interpolate \((x_r, y_r, f_r)\). In this context, a point \((x_r, y_r)\) is defined to be local to another point if it lies within some distance \( R \) of it. Computation of these quadratics constitutes the main work done by this routine. If there are less than 5 other points within distance \( R \) from \((x_r, y_r)\), the quadratic is replaced by a linear function. In cases of rank-deficiency, the minimum norm solution is computed.
The user may specify values for $R_w$ and $R_q$, but it is usually easier to choose instead two integers $N_w$ and $N_q$, from which the routine will compute $R_w$ and $R_q$. These integers can be thought of as the average numbers of data points lying within distances $R_w$ and $R_q$ respectively from each node. Default values are provided, and advice on alternatives is given in Section 8.2.

The interpolant $F(x,y)$ generated by this routine can subsequently be evaluated for any point $(x,y)$ in the domain of the data by a call to E01SFF.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: $m$, the number of data points. Constraint: $M \geq 3$.

2: X(M) -- DOUBLE PRECISION array
3: Y(M) -- DOUBLE PRECISION array
4: F(M) -- DOUBLE PRECISION array
   On entry: the co-ordinates of the $r$th data point, for $r=1,2,\ldots,m$. The order of the data points is immaterial. Constraint: each of the $(X(r),Y(r))$ pairs must be unique.

5: RNW -- DOUBLE PRECISION Input/Output
   On entry: suitable values for the radii $R_w$ and $R_q$, described in Section 3. Constraint: $RNQ \leq 0$ or $0 < RNW \leq RNQ$. On exit: if $RNQ$ is set less than or equal to zero on entry, then default values for both of them will be computed from the parameters NW and NQ, and $RNW$ and $RNQ$ will contain these values on exit.
7: NW -- INTEGER
   Input

8: NQ -- INTEGER
   Input
   On entry: if RNQ > 0.0 and RNW > 0.0 then NW and NQ are not
   referenced by the routine. Otherwise, NW and NQ must specify
   suitable values for the integers N and N described in
   Section 3.
   If NQ is less than or equal to zero on entry, then default
   values for both of them, namely NW = 9 and NQ = 18, will be
   used. Constraint: NQ <= 0 or 0 < NW <= NQ.

9: FNODES(5*M) -- DOUBLE PRECISION array
   Output
   On exit: the coefficients of the constructed quadratic
   nodal functions. These are in a form suitable for passing to
   E01SFF.

10: MINNQ -- INTEGER
    Output
    On exit: the minimum number of data points that lie within
    radius RNQ of any node, and thus define a nodal function. If
    MINNQ is very small (say, less than 5), then the interpolant
    may be unsatisfactory in regions where the data points are
    sparse.

11: WRK(6*M) -- DOUBLE PRECISION array
    Workspace

12: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   On entry M < 3.

IFAIL = 2
   On entry RNQ > 0 and either RNW > RNQ or RNW <= 0.

IFAIL = 3
On entry $NQ > 0$ and either $NW > NQ$ or $NW <= 0$.

IFAIL= 4
On entry $(X(i),Y(i))$ is equal to $(X(j),Y(j))$ for some $i /= j$.

7. Accuracy
On successful exit, the computational errors should be negligible in most situations but the user should always check the computed surface for acceptability, by drawing contours for instance. The surface always interpolates the input data exactly.

8. Further Comments

8.1. Timing
The time taken for a call of E01SEF is approximately proportional to the number of data points, $m$, provided that $N$ is of the same order as its default value (18). However if $N$ is increased so that the method becomes more global, the time taken becomes approximately proportional to $m^2$.

8.2. Choice of $\{w\}$
Note first that the radii $R$ and $R_w$, described in Section 3, are

\[
\begin{array}{ll}
D_w & \text{computed as } \frac{\sqrt{m}}{2} \\
D_q & \text{computed as } \frac{\sqrt{m}}{2}
\end{array}
\]

where $D$ is the maximum distance between any pair of data points.
Default values $N = 9$ and $N = 18$ work quite well when the data points are fairly uniformly distributed. However, for data having some regions with relatively few points or for small data sets ($m < 25$), a larger value of $N$ may be needed. This is to ensure a reasonable number of data points within a distance $R$ of each node, and to avoid some regions in the data area being left outside all the discs of radius $R$ on which the weights $w(x,y)$ are non-zero. Maintaining $N$ approximately equal to $2N$ is usually an advantage.
Note however that increasing \( N \) and \( N \) does not improve the \\
\( w \) \( q \) quality of the interpolant in all cases. It does increase the \\
computational cost and makes the method less local.

9. Example

This program reads in a set of 30 data points and calls E01SEF to 
construct an interpolating surface. It then calls E01SFF to 
evaluate the interpolant at a sample of points on a rectangular 
grid.

Note that this example is not typical of a realistic problem: the 
number of data points would normally be larger, and the 
interpolant would need to be evaluated on a finer grid to obtain 
an accurate plot, say.

The example program is not reproduced here. The source code for 
all example programs is distributed with the NAG Foundation 
Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E01 -- Interpolation
E01SFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for 
your implementation to check implementation-dependent details. 
The symbol (*) after a NAG routine name denotes a routine that is 
not included in the Foundation Library.

1. Purpose

E01SFF evaluates at a given point the two-dimensional 
interpolating function computed by E01SEF.

2. Specification

```fortran
SUBROUTINE E01SFF (M, X, Y, F, RNW, FNODES, PX, PY, PF, 
  IFAIL)
  INTEGER M, IFAIL
  DOUBLE PRECISION X(M), Y(M), F(M), RNW, FNODES(5*M), PX, 
    PY, PF
```

3. Description

This routine takes as input the interpolant \( F(x,y) \) of a set of 
scattered data points \( (x_r,y_r,f_r) \), for \( r=1,2,...,m \), as computed by 
\( E01SEF \), and evaluates the interpolant at the point \( (px,py) \).
If \((px, py)\) is equal to \((x, y)\) for some value of \(r\), the returned \(r\) value will be equal to \(f\) \(r\).

If \((px, py)\) is not equal to \((x, y)\) for any \(r\), all points that are \(r\) \(r\) within distance \(RNW\) of \((px, py)\), along with the corresponding nodal functions given by \(FNODES\), will be used to compute a value of the interpolant.

\(E01SFF\) must only be called after a call to \(E01SEF\).

4. References


5. Parameters

\(1: M \quad -- \quad \text{INTEGER} \quad \text{Input}\)

\(2: X(M) \quad -- \quad \text{DOUBLE PRECISION} \quad \text{array} \quad \text{Input}\)

\(3: Y(M) \quad -- \quad \text{DOUBLE PRECISION} \quad \text{array} \quad \text{Input}\)

\(4: F(M) \quad -- \quad \text{DOUBLE PRECISION} \quad \text{array} \quad \text{Input}\)

\(5: RNW \quad -- \quad \text{DOUBLE PRECISION} \quad \text{Input}\)

\(6: FNODES(5*M) \quad -- \quad \text{DOUBLE PRECISION} \quad \text{array} \quad \text{Input}\)

On entry: \(M, X, Y, F, RNW\) and \(FNODES\) must be unchanged from the previous call of \(E01SEF\).

\(7: PX \quad -- \quad \text{DOUBLE PRECISION} \quad \text{Input}\)

\(8: PY \quad -- \quad \text{DOUBLE PRECISION} \quad \text{Input}\)

On entry: the point \((px, py)\) at which the interpolant is to be evaluated.

\(9: PF \quad -- \quad \text{DOUBLE PRECISION} \quad \text{Output}\)

On exit: the value of the interpolant evaluated at the point \((px, py)\).
10: IFAIL -- INTEGER
     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.

     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
     On entry M < 3.

IFAIL= 2
     The interpolant cannot be evaluated because the evaluation
     point (PX,PY) lies outside the support region of the data
     supplied in X, Y and F. This error exit will occur if
     (PX,PY) lies at a distance greater than or equal to RNW from
     every point given by arrays X and Y.

     The value 0.0 is returned in PF. This value will not provide
     continuity with values obtained at other points (PX,PY),
     i.e., values obtained when IFAIL = 0 on exit.

7. Accuracy

Computational errors should be negligible in most practical
situations.

8. Further Comments

The time taken for a call of E01SFF is approximately proportional
to the number of data points, m.

The results returned by this routine are particularly suitable
for applications such as graph plotting, producing a smooth
surface from a number of scattered points.

9. Example

See the example for E01SEF.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
NagInterpolationPackage (NAGE01)

Exports:
e01baf  e01bef  e01bff  e01bgf  e01bhf  
e01daf  e01saf  e01sbf  e01sef  e01sff

--- package NAGE01 NagInterpolationPackage ---

NagInterpolationPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
  e01baf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_,
    Integer,Integer) -> Result
  ++ e01baf(m,x,y,lck,lwrk,ifail)
  ++ determines a cubic spline to a given set of
++ data.
++ See \downlink{Manual Page}{manpageXXe01bef}.
e01bef : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Integer) \rightarrow Result
++ e01bef(n,x,f,ifail)
++ computes a monotonicity-preserving piecewise cubic Hermite
++ interpolant to a set of data points.
++ See \downlink{Manual Page}{manpageXXe01bef}.
e01bff : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
Integer, Matrix DoubleFloat, Integer) \rightarrow Result
++ e01bff(n,x,f,d,m,px,ifail)
++ evaluates a piecewise cubic Hermite interpolant at a set
++ of points.
++ See \downlink{Manual Page}{manpageXXe01bff}.
e01bgf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
Integer, Matrix DoubleFloat, Integer) \rightarrow Result
++ e01bgf(n,x,f,d,m,px,ifail)
++ evaluates a piecewise cubic Hermite interpolant and its
++ first derivative at a set of points.
++ See \downlink{Manual Page}{manpageXXe01bgf}.
e01bhf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
DoubleFloat, DoubleFloat, Integer) \rightarrow Result
++ e01bhf(n,x,f,d,a,b,ifail)
++ evaluates the definite integral of a piecewise cubic
++ Hermite interpolant over the interval [a,b].
++ See \downlink{Manual Page}{manpageXXe01bhf}.
e01daf : (Integer, Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
Integer) \rightarrow Result
++ e01daf(mx,my,x,y,f,ifail)
++ computes a bicubic spline interpolating surface through a
++ set of data values, given on a rectangular grid in the x-y plane.
++ See \downlink{Manual Page}{manpageXXe01daf}.
e01saf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
Integer) \rightarrow Result
++ e01saf(m,x,y,f,ifail)
++ generates a two-dimensional surface interpolating a set of
++ scattered data points, using the method of Renka and Cline.
++ See \downlink{Manual Page}{manpageXXe01saf}.
e01sbf : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
Matrix Integer, Matrix DoubleFloat, DoubleFloat, DoubleFloat, Integer) \rightarrow Result
++ e01sbf(m,x,y,f,triang,grads,px,py,ifail)
++ evaluates at a given point the two-dimensional interpolant
++ function computed by E01SAF.
++ See \downlink{Manual Page}{manpageXXe01sbf}.
e01sef : (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat,
Integer, Integer, DoubleFloat, DoubleFloat, Integer) \rightarrow Result
++ e01sef(m,x,y,f,nw,nq,rnw,rnq,ifail)
++ generates a two-dimensional surface interpolating a set of
++ scattered data points, using a modified Shepard method.
++ See \downlink{Manual Page}{manpageXXe01sef}.
e01sff: (Integer, Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, 
DoubleFloat, Matrix DoubleFloat, DoubleFloat, DoubleFloat, 
Integer) -> Result
++ e01sff(m,x,y,f,fnodes,px,py,ifail)
++ evaluates at a given point the two-dimensional
++ interpolating function computed by E01SEF.
++ See \downlink{Manual Page}{manpageXXe01sff}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Integer)
import AnyFunctions1(DoubleFloat)

e01baf(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,
[(invokeNagman(NIL$Lisp,_
"e01baf",_
"lamba"::S,"c"::S,"wrk"::S_]
)$Lisp,_
["lamba"::S,"c"::S,"wrk"::S]$Lisp,_
[["double"::S,"x"::S,"m"::S]$Lisp,["y"::S,"m"::S]$Lisp_,
["wrk"::S,"lwrk"::S]$Lisp]$Lisp_]
]$Lisp_]
)$Lisp_]

pretend List (Record(key:Symbol,entry:Any))$Result

e01bef(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"e01bef",_
["d"::S]$Lisp,_
e01bff(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat, 
dArg:Matrix DoubleFloat,mArg:Integer,pxArg:Matrix DoubleFloat, 
ifailArg:Integer): Result == 
  [[invokeNagman(NIL$Lisp, 
"e01bff", 
,"pd":S]$Lisp, 
["double":S,["x":S,"n":S]$Lisp,["f":S,"n":S]$Lisp_ 
,"d":S,"n":S]$Lisp,["px":S,"m":S]$Lisp, 
["integer":S,"n":S,"m":S,"ifail":S]$Lisp_ 
]$Lisp, 
["pf":S,"pd":S]$Lisp_ 
[[[nArg::Any,ifailArg::Any,xArg::Any,fArg::Any, 
dArg::Any,pxArg::Any ]]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result 

e01bgf(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat, 
dArg:Matrix DoubleFloat,mArg:Integer,pxArg:Matrix DoubleFloat, 
ifailArg:Integer): Result == 
  [[invokeNagman(NIL$Lisp, 
"e01bgf", 
,"pd":S]$Lisp, 
["double":S,["x":S,"n":S]$Lisp,["f":S,"n":S]$Lisp_ 
,"d":S,"n":S]$Lisp,["px":S,"m":S]$Lisp, 
["integer":S,"n":S,"m":S,"ifail":S]$Lisp_ 
]$Lisp, 
["pf":S,"pd":S]$Lisp_ 
[[[nArg::Any,mArg::Any,ifailArg::Any,xArg::Any,fArg::Any, 
dArg::Any,pxArg::Any ]]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result 

e01bhf(nArg:Integer,xArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat, 
dArg:Matrix DoubleFloat,aArg:DoubleFloat,bArg:DoubleFloat, 
ifailArg:Integer): Result == 
  [[invokeNagman(NIL$Lisp, 
"e01bhf", 
"["d":S,"n":S]$Lisp_ 
["integer":S,"n":S,"ifail":S]$Lisp_ 
[[[nArg::Any,aArg::Any,bArg::Any,ifailArg::Any, 
fArg::Any,bArg::Any]]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result
e01bhf
"x"::S,"f"::S,"d"::S]$Lisp,
[$"pint"::S]$Lisp,
[["double"::S,["x"::S,"n"::S]$Lisp,[["f"::S,"n"::S]$Lisp_,
$Lisp_,
[$"pint"::S,"ifail"::S]$Lisp_,
[([nArg::Any,aArg::Any,bArg::Any,ifailArg::Any,xArg::Any_,
   fArg::Any,dArg::Any ])
   @(List Any)$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

e01daf(mxArg::Integer,myArg::Integer,xArg:Matrix DoubleFloat,_
yArg:Matrix DoubleFloat,fArg:Matrix DoubleFloat,_
ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_
   "e01daf",_nArg::Any,aArg::Any,bArg::Any,ifailArg::Any,xArg::Any_,
   fArg::Any,dArg::Any ])
   @(List Any)$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

e01saf(mArg::Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
fArg:Matrix DoubleFloat,ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_
   "e01saf",_mArg::Any,yArg::Any,ifailArg::Any,xArg::Any_,
   fArg::Any,ifailArg::Any ])
   @(List Any)$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
e01sbf(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
fArg:Matrix DoubleFloat,triangArg:Matrix Integer,_
gradsArg:Matrix DoubleFloat,_
pxArg:Matrix DoubleFloat,pyArg:DoubleFloat,ifailArg:Integer): Result ==
((invokeNagman(NIL$Lisp,_
"e01sbf",_"m":S,"px":S,"py":S,"pf":S,"ifail":S,_
"x":S,"y":S,"f":S,"triang":S,"grads":S_)
)@$Result
pretend List (Record(key:Symbol,entry:Any))$Result

e01sef(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
fArg:Matrix DoubleFloat,nwArg:Integer,nqArg:Integer,_
rnwArg:DoubleFloat,rnqArg:DoubleFloat,ifailArg:Integer): Result ==
((invokeNagman(NIL$Lisp,_
)@$Result
pretend List (Record(key:Symbol,entry:Any))$Result
package NAGF07 NagLapack

--- NagLapack.input ---

)set break resume
)sys rm -f NagLapack.output
)spool NagLapack.output
)set message test on
)set message auto off
)clear all

-- S 1 of 36
)show NagLapack

--- NAGE01.dotabb ---

"NAGE01" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGE01"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGE01" -> "ALIST"

--- "PACKAGE NAGF07 NAGLAPACK" 2345 ---

e01sff(mArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
  fArg:Matrix DoubleFloat,rnwArg:DoubleFloat,_
  pxArg:DoubleFloat,pyArg:DoubleFloat,ifailArg:Integer): Result ==
  [((invokeNagman(NIL$Lisp,_
    "e01sff",_
    ["m":S,"rnw":S,"px":S,"py":S,"pf":S,_
    ["pf":S]$_Lisp,_,
    [["double":S,"x":S,"m":S]$_Lisp,["y":S,"m":S]$_Lisp,_,
      ["f":S,"m":S]$_Lisp,"rnw":S,["fnodes":S,_
      ["integer":S,"m":S,"ifail":S]$_Lisp,_
    ]$_Lisp,_,
    ["pf":S,"ifail":S]$_Lisp,_,
    [[mArg:Any,rnwArg:Any,pxArg:Any,pyArg:Any,_
    @List Any]$_Lisp)$_Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result
NagLapack is a package constructor
Abbreviation for NagLapack is NAGF07
This constructor is exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGF07

---

Operations
---

f07adf : (Integer,Integer,Integer,Matrix(DoubleFloat)) -> Result
f07aef : (String,Integer,Integer,Matrix(DoubleFloat),Integer,Matrix(Integer),Integer,Matrix(DoubleFloat)) -> Result
f07fdf : (String,Integer,Integer,Matrix(DoubleFloat)) -> Result
f07fef : (String,Integer,Integer,Matrix(DoubleFloat),Integer,Integer,Matrix(DoubleFloat)) -> Result

---

)clear all

---

showArrayValues true

---

(1) true

---

Type: Boolean

---

showScalarValues true

---

(2) true

---

Type: Boolean

---

m:=4

---

(3) 4

---

Type: PositiveInteger

---

n:=4

---

(4) 4

---

Type: PositiveInteger

---

lda:=4
```plaintext
---R
---R
---R (5) 4
---R
---E 6

---S 7 of 36
a:Matrix SF:=
[[1.8 ,2.88 ,2.05 ,-0.89 ],
 [5.25 ,-2.95 ,-0.95 ,-3.8 ],
 [1.58 ,-2.69 ,-2.9 ,-1.04 ],
 [-1.11 ,-0.66 ,-0.59 ,0.8 ]]
---R
---R
---R (6)
---R [1.7999999999999998, 2.8799999999999999, 2.0499999999999998, -0.8899999999999999
 , 5.25,- 2.9499999999999997,- 0.94999999999999996,- 3.7999999999999998],
 ---R [1.5800000000000001, - 2.6900000000000004, - 2.9000000000000004,
 , - 1.1099999999999999, - 0.65999999999999992, - 0.59000000000000008,
 , 0.8000000000000004]
---R
---R
---E 7

---S 8 of 36
--- result:=f07adf(m,n,lda,a)
---E 8

)clear all
---S 9 of 36
showArrayValues true
---R
---R
---R (1) true
---R
---E 9

---S 10 of 36
showScalarValues true
---R
---R
---R (2) true
```
trans:="N"

n:=4

nrhs:=2

a :=
[[5.25, -2.95, -0.95, -3.8 ],
 [0.34, 3.89, 2.38, 0.41 ],
 [0.3, -0.46, -1.51, 0.29 ],
 [-0.21, -0.33, 0.01, 1.13 ]]
 lda:=4

 ipiv:Matrix Integer:= [[2, 2, 3, 4]]

 ldb:=4

 b:Matrix SF:=

 result:=f07aef(trans,n,nrhs,a,lda,ipiv,ldb,b)

 )clear all
```plaintext
showArrayValues true
(1) true
Type: Boolean

showScalarValues true
(2) true
Type: Boolean

uplo:="L"
(3) "L"
Type: String

n:=4
(4) 4
Type: PositiveInteger

lda:=4
(5) 4
Type: PositiveInteger

a:Matrix SF:=
[[4.16 ,0.0 ,0.0 ,0.0 ],
 [-3.12 ,5.03 ,0.0 ,0.0 ],
 [0.56 ,-0.83 ,0.76 ,0.0 ],
 [-0.1 ,1.18 ,0.34 ,1.18 ]]
--R [- 3.1199999999999997, 5.0299999999999994, 0., 0.],
--R [0.56000000000000005, -0.83000000000000007, 0.76000000000000001, 0.],
--R
--R [- 9.9999999999999992E-2, 1.1799999999999999, 0.33999999999999997,
--R 1.1799999999999999]  
--R ]
--R Type: Matrix(DoubleFloat)
--E 25

--S 26 of 36
-- result:=f07fdf(uplo,n,lda,a)
--E 26

)clear all

--S 27 of 36
showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 27

--S 28 of 36
showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 28

--S 29 of 36
uplo:="L"
--R
--R (3) "L"
--R Type: String
--E 29

--S 30 of 36
n:=4
--R
--R (4) 4
--R Type: PositiveInteger
--E 30

--S 31 of 36
nrhs:=2
--R
\[
\begin{bmatrix}
2.04 & 0.0 & 0.0 & 0.0 \\
-1.53 & 1.64 & 0.0 & 0.0 \\
0.28 & -0.25 & 0.79 & 0.0 \\
-0.05 & 0.67 & 0.66 & 0.54
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
\begin{bmatrix}
2.04 & 0.0 & 0.0 & 0.0 \\
-1.53 & 1.63 & 0.0 & 0.0 \\
0.28 & -0.25 & 0.79 & 0.0 \\
-0.05 & 0.67 & 0.66 & 0.54
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
\begin{bmatrix}
8.7 & 8.3 \\
-13.35 & 2.13 \\
1.89 & 1.61 \\
-4.14 & 5
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)
This package uses the NAG Library to compute matrix factorizations, and to solve systems of linear equations following the matrix factorizations.

**F07 -- Linear Equations (LAPACK)**

Chapter F07

**Linear Equations (LAPACK)**

1. **Scope of the Chapter**

This chapter provides four routines concerned with matrix factorization, and the solution of systems of linear equations following the matrix factorizations.

2. **Background to the Problems**

Background material, together with pointers to the routines in this chapter, are to be found in the F01 and F04 Chapter Introductions.

3. **Recommendations on Choice and Use of Routines**

The routines in this chapter are derived from the LAPACK project and may also be called using the LAPACK name, which is given in brackets following the F07 name in the following descriptions.

Routine F07ADF (DGETRF) performs an LU factorization of a real m by n matrix A. Following the use of this routine, F07AEF (DGETRS) may be used to solve a system of n non-singular linear equations, with one or more right-hand sides.

Routine F07FDF (DPOTRF) performs the Cholesky factorization of a real symmetric positive-definite matrix A. Following the use of this routine, F07FEF (DPOTRS) may be used to solve a system of symmetric positive-definite linear equations, with one or more
right-hand sides.

F07 -- Linear Equations (LAPACK)  
Chapter F07

Linear Equations (LAPACK)

F07ADF (DGETRF) LU factorization of real m by n matrix

F07AEF (DGETRS) Solution of real system of linear equations, multiple right-hand sides, matrix already factorized by F07ADF

F07FDF (DPOTRF) Cholesky factorization of real symmetric positive-definite matrix

F07FEF (DPOTRS) Solution of real symmetric positive-definite system of linear equations, multiple right-hand sides, matrix already factorized by F07FDF

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F07 -- Linear Equations (LAPACK)  
F07ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F07ADF (DGETRF) computes the LU factorization of a real m by n matrix.

2. Specification

```
SUBROUTINE F07ADF (M, N, A, LDA, IPIV, INFO)
ENTRY M, N, A, LDA, IPIV, INFO
INTEGER M, N, LDA, IPIV(*), INFO
DOUBLE PRECISION A(LDA,*
```

The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

This routine forms the LU factorization of a real m by n matrix A as A=PLU, where P is a permutation matrix, L is lower triangular
with unit diagonal elements (lower trapezoidal if m>n) and U is upper triangular (upper trapezoidal if m<n). Usually A is square (m=n), and both L and U are triangular. The routine uses partial pivoting, with row interchanges.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: m, the number of rows of the matrix A.
   Constraint: M >= 0.

2: N -- INTEGER Input
   On entry: n, the number of columns of the matrix A.
   Constraint: N >= 0.

3: A(LDA,*) -- DOUBLE PRECISION array Input/Output
   Note: the second dimension of the array A must be at least max(1,N).
   On entry: the m by n matrix A. On exit: A is overwritten by the factors L and U; the unit diagonal elements of L are not stored.

4: LDA -- INTEGER Input
   On entry:
   the first dimension of the array A as declared in the (sub)program from which F07ADF is called.
   Constraint: LDA >= max(1,M).

5: IPIV(*) -- INTEGER array Output
   Note: the dimension of the array IPIV must be at least max(1,min(M,N)).
   On exit: the pivot indices. Row i of the matrix A was interchanged with row IPIV(i) for i=1,2,...,min(m,n).

6: INFO -- INTEGER Output
   On exit: INFO = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

INFO < 0
If INFO = -i, the ith parameter has an illegal value. An explanatory message is output, and execution of the program is terminated.
INFO > 0
If INFO = i, u is exactly zero. The factorization has been completed but the factor U is exactly singular, and division by zero will occur if it is subsequently used to solve a system of linear equations or to compute $A^{-1}$.

7. Accuracy
The computed factors L and U are the exact factors of a perturbed matrix $A+E$, where

$$|E| \leq c\left(\min(m,n)\right)(\text{epsilon})P|L||U|,$$

$c(n)$ is a modest linear function of $n$, and $(\text{epsilon})$ is the machine precision.

8. Further Comments
The total number of floating-point operations is approximately

$$2^3 1 2 1 2$$

$-n$ if $m=n$ (the usual case), $-n (3m-n)$ if $m>n$ and $-m (3n-m)$ if $m<n$.

A call to this routine with $m=n$ may be followed by calls to the routines:

F07AEF (DGETRS) to solve $AX=B$ or $A^{-1}X=B$;
F07AGF (DGECON) (*) to estimate the condition number of $A$;
F07AJF (DGETRI) (*) to compute the inverse of $A$.

The complex analogue of this routine is F07ARF (ZGETRF) (*).

9. Example
To compute the LU factorization of the matrix $A$, where

$$A = \begin{pmatrix}
1.80 & 2.88 & 2.05 & -0.89 \\
5.25 & -2.95 & -0.95 & -3.80 \\
1.58 & -2.69 & -2.90 & -1.04 \\
-1.11 & -0.66 & -0.59 & 0.80
\end{pmatrix}$$

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation.
Library software and should be available on-line.

F07 -- Linear Equations (LAPACK)  
F07AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F07AEF (DGETRS) solves a real system of linear equations with multiple right-hand sides, \( AX=B \) or \( A^T X=B \), where \( A \) has been factorized by F07ADF (DGETRF).

2. Specification

```fortran
SUBROUTINE F07AEF (TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO)
ENTRY TRANS, N, NRHS, A, LDA, IPIV, B, LDB, INFO
INTEGER N, NRHS, LDA, IPIV(*), LDB, INFO
DOUBLE PRECISION A(LDA,*), B(LDB,*)
CHARACTER*1 TRANS
```

The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

To solve a real system of linear equations \( AX=B \) or \( A^T X=B \), this routine must be preceded by a call to F07ADF (DGETRF) which computes the LU factorization of \( A \) as \( A=PLU \). The solution is computed by forward and backward substitution.

If \( TRANS = 'N' \), the solution is computed by solving \( PLY=B \) and then \( UX=Y \).

If \( TRANS = 'T' \) or \( 'C' \), the solution is computed by solving \( U^T Y=B \) and then \( L^T P^T X=Y \).

4. References

5. Parameters

1: TRANS -- CHARACTER*1  
   On entry: indicates the form of the equations as follows:  
   if TRANS = 'N', then AX=B is solved for X;  
   if TRANS = 'T' or 'C', then A^T X=B is solved for X.  
   Constraint: TRANS = 'N', 'T' or 'C'.

2: N -- INTEGER  
   On entry: n, the order of the matrix A. Constraint: N >= 0.

3: NRHS -- INTEGER  
   On entry: r, the number of right-hand sides. Constraint: NRHS >= 0.

4: A(LDA,*) -- DOUBLE PRECISION array  
   Note: the second dimension of the array A must be at least  
   max(1,N).
   On entry: the LU factorization of A, as returned by F07ADF  
   (DGETRF).

5: LDA -- INTEGER  
   On entry:  
   the first dimension of the array A as declared in the  
   (sub)program from which F07AEF is called.  
   Constraint: LDA >= max(1,N).

6: IPIV(*) -- INTEGER array  
   Note: the dimension of the array IPIV must be at least  
   max(1,N).
   On entry: the pivot indices, as returned by F07ADF (DGETRF).

7: B(LDB,*) -- DOUBLE PRECISION array  
   Input/Output  
   Note: the second dimension of the array B must be at least  
   max(1,NRHS).
   On entry: the n by r right-hand side matrix B. On exit: the  
   n by r solution matrix X.

8: LDB -- INTEGER  
   On entry:  
   the first dimension of the array B as declared in the  
   (sub)program from which F07AEF is called.  
   Constraint: LDB >= max(1,N).

9: INFO -- INTEGER  
   Output
On exit: INFO = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

INFO < 0
If INFO = -i, the ith parameter has an illegal value. An explanatory message is output, and execution of the program is terminated.

7. Accuracy

For each right-hand side vector b, the computed solution x is the exact solution of a perturbed system of equations (A+E)x = b, where

\[ |E| \leq c(n)(\epsilon)\|L\|\|U\|, \]

D is a modest linear function of n, and (\epsilon) is the machine precision.

\[ ||x-x||_\infty \leq c(n)\text{cond}(A,x)(\epsilon)\frac{||x||_\infty}{-1} \]

where \text{cond}(A,x) = \frac{||A||_\infty||x||_\infty}{||x||_\infty} \leq (\kappa)(A). Note that \text{cond}(A,x) can be much smaller than \text{cond}(A), and \text{cond}(A) can be much larger (or smaller) than \text{cond}(A).

Forward and backward error bounds can be computed by calling F07AHF (DGERFS)(*), and an estimate for (\kappa)(A) can be obtained by calling F07AGF (DGECON)(*) with NORM = 'I'.

8. Further Comments

The total number of floating-point operations is approximately 2
This routine may be followed by a call to F07AHF (DGERFS)(*) to refine the solution and return an error estimate.

The complex analogue of this routine is F07ASF (ZGETRS)(*).

9. Example

To solve the system of equations $AX=B$, where

$$
A = \begin{pmatrix}
1.80 & 2.88 & 2.05 & -0.89 \\
5.25 & -2.95 & -0.95 & -3.80 \\
1.58 & -2.69 & -2.90 & -1.04 \\
-1.11 & -0.66 & -0.59 & 0.80
\end{pmatrix}
$$

and

$$
B = \begin{pmatrix}
9.52 & 18.47 \\
24.35 & 2.25 \\
0.77 & -13.28 \\
-6.22 & -6.21
\end{pmatrix}
$$

Here $A$ is unsymmetric and must first be factorized by F07ADF (DGETRF)).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

This routine forms the Cholesky factorization of a real symmetric positive-definite matrix $A$ either as $A = U^T U$ if $\text{UPLO} = 'U'$ or $A = L^T L$ if $\text{UPLO} = 'L'$, where $U$ is an upper triangular matrix and $L$ is lower triangular.

4. References


5. Parameters

1: UPLO -- CHARACTER*1  Input
   On entry: indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factorized, as follows:
   if $\text{UPLO} = 'U'$, then the upper triangular part of $A$ is stored and $A$ is factorized as $U^T U$, where $U$ is upper triangular;
   if $\text{UPLO} = 'L'$, then the lower triangular part of $A$ is stored and $A$ is factorized as $L^T L$, where $L$ is lower triangular.
   Constraint: $\text{UPLO} = 'U'$ or 'L'.

2: N -- INTEGER  Input
   On entry: $n$, the order of the matrix $A$. Constraint: $N \geq 0$.

3: A(LDA,*) -- DOUBLE PRECISION array  Input/Output
   Note: the second dimension of the array $A$ must be at least $\max(1,N)$.
   On entry: the $n$ by $n$ symmetric positive-definite matrix $A$.
   If $\text{UPLO} = 'U'$, the upper triangle of $A$ must be stored and the elements of the array below the diagonal are not
referenced; if UPLO = 'L', the lower triangle of A must be stored and the elements of the array above the diagonal are not referenced. On exit: the upper or lower triangle of A is overwritten by the Cholesky factor U or L as specified by UPLO.

4: LDA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F07FDF is called.
   Constraint: LDA >= max(1,N).

5: INFO -- INTEGER Output
   On exit: INFO = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

   INFO < 0
   If INFO = -i, the ith parameter has an illegal value. An explanatory message is output, and execution of the program is terminated.

   INFO > 0
   If INFO = i, the leading minor of order i is not positive-definite and the factorization could not be completed. Hence A itself is not positive-definite. This may indicate an error in forming the matrix A. To factorize a symmetric matrix which is not positive-definite, call F07MDF (DSYTRF)(*) instead.

7. Accuracy

   If UPLO = 'U', the computed factor U is the exact factor of a perturbed matrix A+E, where

   \[ |E| \leq c(n) \epsilon |U||U| \]

   c(n) is a modest linear function of n, and (epsilon) is the machine precision. If UPLO = 'L', a similar statement holds for the computed factor L. It follows that

   \[ |e_{ij}| \leq c(n) \epsilon \sqrt{a_{ii} a_{jj}} \]

8. Further Comments

   The total number of floating-point operations is approximately
A call to this routine may be followed by calls to the routines:

F07FEF (DPOTRS) to solve AX=B;
F07FGF (DPOCON)(*) to estimate the condition number of A;
F07FJF (DPOTRI)(*) to compute the inverse of A.

The complex analogue of this routine is F07FRF (ZPOTRF)(*).

9. Example

To compute the Cholesky factorization of the matrix A, where

\[
A = \begin{pmatrix}
4.16 & -3.12 & 0.56 & -0.10 \\
-3.12 & 5.03 & -0.83 & 1.18 \\
0.56 & -0.83 & 0.76 & 0.34 \\
-0.10 & 1.18 & 0.34 & 1.18
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The ENTRY statement enables the routine to be called by its
LAPACK name.

3. Description

To solve a real symmetric positive-definite system of linear
equations $AX = B$, this routine must be preceded by a call to
F07FDF (DPOTRF) which computes the Cholesky factorization of $A$.
The solution $X$ is computed by forward and backward substitution.

If $\text{UPLO} = 'U'$, $A = U^T U$, where $U$ is upper triangular; the solution $X$ is computed by solving $U^T Y = B$ and then $UX = Y$.

If $\text{UPLO} = 'L'$, $A = L L^T$, where $L$ is lower triangular; the solution $X$ is computed by solving $LY = B$ and then $L^T X = Y$.

4. References


5. Parameters

1: $\text{UPLO} -- \text{CHARACTER*1}$
   Input
   On entry: indicates whether the upper or lower triangular part of $A$ is stored and how $A$ is factorized, as follows:
   
   $T$
   if $\text{UPLO} = 'U'$, then $A = U^T U$ where $U$ is upper triangular;
   
   $T$
   if $\text{UPLO} = 'L'$, then $A = L L^T$ where $L$ is lower triangular.
   Constraint: $\text{UPLO} = 'U'$ or 'L'.

2: $N -- \text{INTEGER}$
   Input
   On entry: $n$, the order of the matrix $A$. Constraint: $N \geq 0$.

3: $\text{NRHS} -- \text{INTEGER}$
   Input
   On entry: $r$, the number of right-hand sides. Constraint: $\text{NRHS} \geq 0$.

4: $A(\text{LDA,*}) -- \text{DOUBLE PRECISION array}$
   Input
   Note: the second dimension of the array $A$ must be at least
   $\text{max}(1,N)$.
   On entry: the Cholesky factor of $A$, as returned by F07FDF.
5: LDA -- INTEGER  
   Input  
   On entry:  
   the first dimension of the array A as declared in the  
   (sub)program from which F07FEF is called.  
   Constraint: LDA >= max(1,N).

6: B(LDB,*) -- DOUBLE PRECISION array  
   Input/Output  
   Note: the second dimension of the array B must be at least  
   max(1,NRHS).  
   On entry: the n by r right-hand side matrix B.

7: LDB -- INTEGER  
   Input  
   On entry:  
   the first dimension of the array B as declared in the  
   (sub)program from which F07FEF is called.  
   Constraint: LDB >= max(1,N).

8: INFO -- INTEGER  
   Output  
   On exit: INFO = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

INFO < 0  
If INFO = -i, the ith parameter has an illegal value. An  
explanatory message is output, and execution of the program  
is terminated.

7. Accuracy

For each right-hand side vector b, the computed solution x is the  
exact solution of a perturbed system of equations (A+E)x=b, where  
\[ |E| \leq c(n)\epsilon |U||U| \text{ if } UPLD = 'U', \]
\[ |E| \leq c(n)\epsilon |L||L| \text{ if } UPLD = 'L', \]
c(n) is a modest linear function of n, and (epsilon) is the  
machine precision.

If x is the true solution, then the computed solution x satisfies  
a forward bound of the form  
\[ ||x-x|| \]
\[ c(\varepsilon) \leq \text{cond}(A, x) \] where \( \text{cond}(A, x) = \frac{||A||x||}{||x||} \)

Forward and backward error bounds can be computed by calling F07FHF (DPORFS)(*), and an estimate for \( (\kappa) (A) \) can be obtained by calling F07FGF (DPOCON)(*).

8. Further Comments

The total number of floating-point operations is approximately \( 2 \cdot 2n \cdot r \).

This routine may be followed by a call to F07FH (DPORFS)(*) to refine the solution and return an error estimate.

The complex analogue of this routine is F07FSF (ZPOTRS)(*).

9. Example

To compute the Cholesky factorization of the matrix \( A \), where

\[
A = \begin{pmatrix}
4.16 & -3.12 & 0.56 & -0.10 \\
-3.12 & 5.03 & -0.83 & 1.18 \\
0.56 & -0.83 & 0.76 & 0.34 \\
-0.10 & 1.18 & 0.34 & 1.18
\end{pmatrix}
\]

and

\[
B = \begin{pmatrix}
8.70 & 8.30 \\
-13.35 & 2.13 \\
1.89 & 1.61 \\
-4.14 & 5.00
\end{pmatrix}
\]

Here \( A \) is symmetric positive-definite and must first be factorized by F07FDF (DPOTRF).

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**NagLapack (NAGF07)**

Exports:

- `f07adf`  
- `f07aef`  
- `f07fdf`  
- `f07fef`

---

)abbrev package NAGF07 NagLapack  
++ Author: Godfrey Nolan and Mike Dewar  
++ Date Created: Jan 1994  
++ Date Last Updated: Thu May 12 17:45:42 1994  
++ Description:  
++ This package uses the NAG Library to compute matrix factorizations, and to solve systems of linear equations following the matrix factorizations.

NagLapack(): Exports == Implementation where  
S ==> Symbol  
FOP ==> FortranOutputStackPackage

Exports == with  
  f07adf : (Integer,Integer,Integer,Matrix DoubleFloat) -> Result  
  ++ f07adf(m,n,lda,a)  
  ++ (DGETRF) computes the LU factorization of a real m by n matrix.  
  ++ See \downlink{Manual Page}{manpageXXf07adf}.  
  f07aef : (String,Integer,Integer,Matrix DoubleFloat,  
           Integer,Matrix Integer,Integer,Matrix DoubleFloat) -> Result
++ f07aef(trans,n,nrhs,a,lda,ipiv,ldb,b)
++ (DGETRS) solves a real system of linear equations with
++ multiple right-hand sides, AX=B or XA=B, where A has been
++ factorized by F07ADF (DGETRF).
++ See `downlink`{Manual Page}{manpageXXf07aef}.

f07fdf : (String,Integer,Integer,Matrix DoubleFloat) -> Result
++ f07fdf(uplo,n,lda,a)
++ (DPOTRF) computes the Cholesky factorization of a real
++ symmetric positive-definite matrix.
++ See `downlink`{Manual Page}{manpageXXf07fdf}.

f07fef : (String,Integer,Integer,Matrix DoubleFloat,_
  Integer,Integer,Matrix DoubleFloat) -> Result
++ f07fef(uplo,n,nrhs,a,lda,ldb,b)
++ (DPOTRS) solves a real symmetric positive-definite system
++ of linear equations with multiple right-hand sides, AX=B, where A
++ has been factorized by F07FDF (DPOTRF).
++ See `downlink`{Manual Page}{manpageXXf07fef}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(String)
import AnyFunctions1(Matrix Integer)

f07adf(mArg:Integer,nArg:Integer,ldaArg:Integer,_
  aArg:Matrix DoubleFloat): Result ==
  [((invokeNagman(NIL$Lisp,_
    "f07adf",_
    ["ipiv"::S,"info"::S]$Lisp,_
    ["double"::S,["a"::S,"lda"::S,"n"::S]$Lisp_
      ]$Lisp,_
        ,"info"::S]$Lisp,_
      ]$Lisp,_
    ["ipiv"::S,"info"::S,"a"::S]$Lisp,_
    [([mArg::Any,nArg::Any,ldaArg::Any,aArg::Any ])
      @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result
f07aef(transArg: String, nArg: Integer, nrhsArg: Integer, _
aArg: Matrix DoubleFloat, ldaArg: Integer, ipivArg: Matrix Integer, _
ldbArg: Integer, bArg: Matrix DoubleFloat): Result ==
[[invokeNagman(NIL$Lisp, _
"f07aef", _
["trans": S, "n": S, "nrhs": S, "lda": S, "ldb": S, _
"info": S, "a": S, "ipiv": S, "b": S]$Lisp, _
["info": S]$Lisp, _
["double": S, ["a": S, "lda": S, "n": S]$Lisp, _
["b": S, "ldb": S, "nrhs": S]$Lisp]$Lisp
, ["integer": S, "n": S, "nrhs": S, "lda": S, ["ipiv": S, "n": S]$Lisp, _
"ldb": S, "info": S]$Lisp
, ["character": S, "trans": S]$Lisp
]$Lisp
, ["info": S, "b": S]$Lisp,
[[transArg:: Any, nArg:: Any, nrhsArg:: Any, ldaArg:: Any, _
ldbArg:: Any, aArg:: Any, ipivArg:: Any, bArg:: Any ])_
@List Any$Lisp]$Lisp)_
pretend List (Record(key: Symbol, entry: Any))$Result

f07fdf(uploArg: String, nArg: Integer, ldaArg: Integer, _
aArg: Matrix DoubleFloat): Result ==
[[invokeNagman(NIL$Lisp, _
"f07fdf", _
["uplo": S, "n": S, "lda": S, "info": S, "a": S]$Lisp, _
["info": S]$Lisp, _
["double": S, ["a": S, "lda": S, "n": S]$Lisp
]$Lisp
, ["integer": S, "n": S, "lda": S, "info": S]$Lisp
, ["character": S, "uplo": S]$Lisp
]$Lisp
, ["info": S, "a": S]$Lisp,
[[uploArg:: Any, nArg:: Any, ldaArg:: Any, aArg:: Any ])_
@List Any$Lisp]$Lisp)_
pretend List (Record(key: Symbol, entry: Any))$Result

f07fef(uploArg: String, nArg: Integer, nrhsArg: Integer, _
aArg: Matrix DoubleFloat, ldaArg: Integer, ldbArg: Integer, _
bArg: Matrix DoubleFloat): Result ==
[[invokeNagman(NIL$Lisp, _
"f07fef", _
["uplo": S, "n": S, "nrhs": S, "lda": S, "ldb": S, _
"info": S, "a": S, "b": S]$Lisp, _
["info": S]$Lisp, _
["double": S, ["a": S, "lda": S, "n": S]$Lisp
, ["b": S, "ldb": S, "nrhs": S]$Lisp]$Lisp
, ["integer": S, "n": S, "nrhs": S, "lda": S, "ldb": S, _
"info": S]$Lisp
, ["character": S, "uplo": S]$Lisp
]$Lisp,
package NAGF01 NagMatrixOperationsPackage

--- NagMatrixOperationsPackage.input ---

)set break resume
)sys rm -f NagMatrixOperationsPackage.output
)spool NagMatrixOperationsPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 130
)show NagMatrixOperationsPackage
--R
--R NagMatrixOperationsPackage is a package constructor
--R Abbreviation for NagMatrixOperationsPackage is NAGF01
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGF01
--R
--R-------------------------------------------------- Operations --------------------------
--R f01brf : (Integer,Integer,Integer,Integer,DoubleFloat,Boolean,Boolean,List(Boolean),Matrix(DoubleFloat),Matrix(Integer),Matrix(Integer),Integer) -> Result
--R f01bsf : (Integer,Integer,Integer,Matrix(Integer),Matrix(Integer),Matrix(Integer),Matrix(Integer),Boolean,DoubleFloat,Boolean,Matrix(Integer),Matrix(DoubleFloat),Integer) -> Result
--R f01maf : (Integer,Integer,Integer,Integer,List(Boolean),Matrix(DoubleFloat),Matrix(Integer),Matrix(Integer),DoubleFloat,DoubleFloat,Integer) -> Result
--R f01mcf : (Integer,Matrix(DoubleFloat),Integer,Matrix(Integer),Integer) -> Result
--R f01qcf : (Integer,Integer,Integer,Matrix(DoubleFloat),Integer) -> Result
--R f01qdf : (String,String,Integer,Integer,Matrix(DoubleFloat),Integer,Matrix(DoubleFloat),Integer,Integer,Matrix(DoubleFloat),Integer) -> Result
--R f01qef : (String,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R f01rcf : (Integer,Integer,Integer,Matrix(Complex(DoubleFloat)),Integer) -> Result
```plaintext
--R f01rdf : (String,String,Integer,Integer,Matrix(Complex(DoubleFloat)),Integer,Matrix(Complex(DoubleFloat)),Integer,Integer,Matrix(Complex(DoubleFloat)),Integer) -> Result
--R f01ref : (String,Integer,Integer,Integer,Integer,Matrix(Complex(DoubleFloat)),Matrix(Complex(DoubleFloat)),Integer) -> Result

)clear all

--S 2 of 130
showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 2

--S 3 of 130
showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 3

--S 4 of 130
n:=6
--R
--R (3) 6
--R Type: PositiveInteger
--E 4

--S 5 of 130
nz:=15
--R
--R (4) 15
--R Type: PositiveInteger
--E 5

--S 6 of 130
licn:=150
--R
--R (5) 150
--R Type: PositiveInteger
--E 6

--S 7 of 130
lirn:=75
--R
```
pivot:=0.1

lblock:=true

grow:=true

abort:=true ,true ,false ,true ]
--R (11)
--R [  
--R [5., 2., -1., 2., 3., -2., 1., 1., -1., 2., -3., -1., -1., 6.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,  
--R Type: Matrix(DoubleFloat)  
--E 12

--S 13 of 130
irn:Matrix Integer:=  
[[1,2,2,2,3,4,4,4,5,5,6,6,6,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R Type: Matrix(DoubleFloat)  
--E 13

--S 14 of 130
icn:Matrix Integer:=  
[[1,2,3,4,3,1,4,5,1,4,5,6,1,2,6,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,  
--R Type: Matrix(Integer)  
--E 13

--S 15 of 130

Type: Matrix(Integer)

ifail := 110

result := f01brf(n, nz, licn, lirn, pivot, lblock, grow, abort, a, irn, icn, ifail)

)clear all

showArrayValues true

showScalarValues true

n := 6
nz := 15
--R
--R
--R (4) 15
--R
--E 20

--S 21 of 130
nzmax := 50
--R
--R
--R (5) 50
--R
--E 21

--S 22 of 130
licn := 3*nzmax
--R
--R
--R (6) 150
--R
--E 22

--S 23 of 130
ta1:List SF := new(licn-nz,0.0);
--R
--R
--R
--E 23

--S 24 of 130
ta2:List SF := [5.,2.,-1.,2.,3.,-2.,1.,1.,-1.,-1.,2.,-3.,-1.,-1.,6.];
--R
--R
--R
--E 24

--S 25 of 130
ta3 := concat(ta2,ta1);
--R
--R
--R
--E 25

--S 26 of 130
a :Matrix SF := matrix [ta3]
--R
--R
--R (10)
--R
\[
\begin{array}{cccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
tc1 : List Integer := new(licn-nz,0);

Type: List(Integer)

tc2 := [1,2,3,4,3,1,4,5,1,4,5,6,1,2,6];

Type: List(PositiveInteger)

tc3 := concat(tc2,tc1);

Type: List(Integer)

icn := matrix [tc3]

Type: Matrix(Integer)

pivot : SF := 0.1

Type: DoubleFloat
lblock := true
--R
--R (21) true
--R
--E 37

--S 38 of 130
grow := true
--R
--R
--R (22) true
--R
--E 38

--S 39 of 130
abort := [true,true,false,true]
--R
--R
--R (23) [true,true,false,true]
--R
--E 39

--S 40 of 130
ifail := 110
--R
--R
--R (24) 110
--R
--E 40

--S 41 of 130
f01brfResult:=f01brf(n,nz,licn,lirn,pivot,lblock,grow,abort,a,irn,icn,ifail);
--E 41

--S 42 of 130
eta :SF:= .1 -- now run f01bsf
--R
--R
--R (25) 0.10000000000000001
--R
--E 42

--S 43 of 130
abort := true
--R
--R

eta := 0.1
--R  (26) true
--R
--E 43

--S 44 of 130
-- keep := f01brfResult.'ikeep;
--E 44

--S 45 of 130
-- disp := f01brfResult.'idisp;
--E 45

--S 46 of 130
-- cn := f01brfResult.'icn;
--E 46

--S 47 of 130
ta2:=[10.,12.,-3.,-1.,15.,-2.,10.,-1.,-1.,-5.,1.,-1.,-1.,-2.,6.];
--R
--R
--R Type: List(DoubleFloat)
--E 47

--S 48 of 130
a:= matrix [concat (ta2,ta1)]
--R
--R
--R (28)
--R [  
--R - 2., 6., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R ]
--R
--R Type: Matrix(DoubleFloat)
--E 48

--S 49 of 130
result:=f01bsf(n,nz,licn,matrix [tr2],matrix [tc2],cn,keep,grow,eta,abort,_
-- matrix [[disp(1,1),disp(1,2)],[a],ifail)
--E 49

)clear all

--S 50 of 130
showArrayValues true
--R
--R
--R (1) true
--R
--E 50

--S 51 of 130
showScalarValues true
--R
--R
--R (2) true
--R
--E 51

--S 52 of 130
n:=16
--R
--R
--R (3) 16
--R
--E 52

--S 53 of 130
nz:=40
--R
--R
--R (4) 40
--R
--E 53

--S 54 of 130
licn:=90
--R
--R
--R (5) 90
--R
--E 54

--S 55 of 130
lirn:=50
--R
--R
--R (6) 50
--R
--E 55

--S 56 of 130
abort:=[true, true, true]
--R
PACKAGE NAGF01 NAGMATRIXOPERATIONSPACKAGE
--R
--R
(7)
--R
--E 56

2381

[true,true,true]
Type: List(Boolean)

--S 57 of 130
avals:Matrix SF:=
[[1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,1.0_
,1.0 ,1.0 ,1.0 ,1.0 ,1.0 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25_
,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25_
,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25 ,-0.25_
,-0.25 ,-0.25 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0_
,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0_
,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ]]
--R
--R
--R
(8)
--R
[
--R
[1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., - 0.25,
--R
- 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25,
--R
- 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25, - 0.25,
--R
- 0.25, - 0.25, - 0.25, - 0.25, - 0.25, 0., 0., 0., 0., 0., 0., 0., 0.,
--R
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R
0., 0., 0., 0., 0., 0.]
--R
]
--R
Type: Matrix(DoubleFloat)
--E 57
--S 58 of 130
irn:Matrix Integer:=
[[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14_
,15 ,16 ,1 ,2 ,3 ,5 ,6 ,7 ,9 ,10 ,11 ,13 ,14_
,15 ,1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12_
,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ]]
--R
--R
--R
(9)
--R
[
--R
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 5, 6, 7,
--R
9, 10, 11, 13, 14, 15, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 0, 0, 0, 0,
--R
0, 0, 0, 0, 0, 0]
--R
]
--R
Type: Matrix(Integer)
--E 58
--S 59 of 130
icn:Matrix Integer:=
[[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,10 ,11 ,12 ,13 ,14_
,15 ,16 ,2 ,3 ,4 ,6 ,7 ,8 ,10 ,11 ,12 ,14 ,15 ,16_


droptl := 0.1

densw := 0.8

ifail := 111

result := f01maf(n, nz, licn, lirn, abort, avals, irn, icn, droptl, densw, ifail)

)clear all

showArrayValues true
showScalarValues true

n:=6

nrow:Matrix Integer:= [[1, 2, 1, 5, 3]]

result:=f01mcf(n, avals, lal, nrow, -1)
)clear all

showArrayValues true
--R
--R
--R (1) true
--R
--E 71

showScalarValues true
--R
--R
--R (2) true
--R
--E 72

m:=5
--R
--R
--R (3) 5
--R
--E 73

n:=3
--R
--R
--R (4) 3
--R
--E 74

lda:=5
--R
--R
--R (5) 5
--R
--E 75

a:Matrix SF:=
[[2.0 ,2.5 ,2.5 ],
 [2.0 ,2.5 ,2.5 ],
 [1.6 , 0.4 , 2.8 ],
 [2.0 , 0.5 , 0.5 ],
 [1.2 , 0.3 , 2.9 ]]
--R
--R  +  2.  2.5  2.5 +
--R  |
--R  |
--R  |
--R  (6) | 1.6000000000000001  - 0.39999999999999997  2.7999999999999998 |
--R  |
--R  |
--R  +  1.2  - 0.30000000000000004  - 2.9000000000000004+ 
--R  Type: Matrix(DoubleFloat)
--E 76

--S 77 of 130
-- result:=f01qcf(m,n,lda,a,-1)
--E 77

)clear all

--S 78 of 130
showArrayValues true
--R
--R  (1) true
--R    Type: Boolean
--E 78

--S 79 of 130
showScalarValues true
--R
--R  (2) true
--R    Type: Boolean
--E 79

--S 80 of 130
trans:="n"
--R
--R  (3) "n"
--R    Type: String
--E 80

--S 81 of 130
wheret:="i"
--R
--R  (4) "i"
--R    Type: String
--E 81
m:=5

Type: PositiveInteger

n:=3

Type: PositiveInteger

a:Matrix SF:=

\[
\begin{bmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9 \\
\end{bmatrix}
\]

+ 2. 2.5 2.5 +

Type: Matrix(DoubleFloat)

lda:=5

Type: PositiveInteger

zeta:Matrix SF:=

\[
\begin{bmatrix}
0.0 & 0.0 \\
\end{bmatrix}
\]
ncolb:=2

ldb:=5

b:Matrix SF:=
[[1.1,0.0],
 [0.9,0.0],
 [0.6,1.32],
 [0.0,1.1],
 [-0.8,-0.26]]

result:=f01qdf(trans,wheret,m,n,a,lda,zeta,ncolb,ldb,b,-1)

)clear all

showArrayValues true
--R
--R (1) true
--R
--E 91

--S 92 of 130
showScalarValues true
--R
--R
--R (2) true
--R
--E 92

--S 93 of 130
wheret:="s"
--R
--R
--R (3) "s"
--R
--E 93

--S 94 of 130
m:=5
--R
--R
--R (4) 5
--R
--E 94

--S 95 of 130
n:=3
--R
--R
--R (5) 3
--R
--E 95

--S 96 of 130
ncolq:=5
--R
--R
--R (6) 5
--R
--E 96

--S 97 of 130
lda:=5
--R
--R
--R (7) 5
zeta: Matrix SF:= \[
[1.2247, 1.1547, 1.2649],
\]

\(\text{(8)}\) \[
[1.2246999999999999, 1.1547000000000001, 1.2648999999999999]
\]

a: Matrix SF:=
\[
[-4.0, -3.0, 0.0, 0.0, 0.0],
[0.4085, -2.0, 0.0, 0.0, 0.0],
[0.3266, -0.4619, -4.0, 0.0, 0.0],
[0.4082, -0.5774, 0.0, 0.0, 0.0],
[0.2449, -0.3464, -0.6326, 0.0, 0.0]
\]

result:= f01qef(wheret, m, n, ncolq, lda, zeta, a, -1)

)clear all

)showArrayValues true

)showScalarValues true
m := 5

n := 3

lda := 5

a := Matrix Complex SF :=
[[complex(0, 0.5), complex(-0.5, 1.5), complex(-1.0, 1.0)],
 [complex(0.4, 0.3), complex(0.9, 1.3), complex(0.2, 1.4)],
 [complex(0.4, 0), complex(-0.4, 0.4), complex(1.8, 0)],
 [complex(0.3, -0.4), complex(0.1, 0.7), complex(0, 0)],
 [complex(0, -0.3), complex(0.3, 0.3), complex(0, 2.4)]]

[0.5 %i, -0.5 + 1.5 %i, -1. + %i],
[0.400000000000002 + 0.29999999999999999 %i,
 0.899999999999999991 + 1.29999999999999998 %i,
 0.200000000000000001 + 1.39999999999999999 %i]
[0.29999999999999999 - 0.39999999999999997 %i,
0.10000000000000001 + 0.69999999999999996 %i, 0.]

[- 0.30000000000000004 %i, 0.29999999999999999 + 0.29999999999999999 %i,
2.3999999999999999 %i]

Type: Matrix(Complex(DoubleFloat))
\begin{verbatim}
--R
--R
--R (5) 5
--R Type: PositiveInteger
--E 112

--S 113 of 130
n:=3
--R
--R
--R (6) 3
--R Type: PositiveInteger
--E 113

--S 114 of 130
a:Matrix Complex SF:=
\begin{bmatrix}
0.5 \cdot \text{i}, & -0.5 + 1.5 \cdot \text{i}, & -1.0 + 1.0 \cdot \text{i} \\
0.4 + 0.3 \cdot \text{i}, & 0.9 + 1.3 \cdot \text{i}, & 0.2 + 1.4 \cdot \text{i} \\
0.4, & -0.4 + 0.4 \cdot \text{i}, & 1.8 \\
0.3 - 0.4 \cdot \text{i}, & 0.1 + 0.7 \cdot \text{i}, & 0.0 \\
-0.3 \cdot \text{i}, & 0.3 + 0.3 \cdot \text{i}, & 2.4
\end{bmatrix}
--R
--R
--R (7)
--R \begin{bmatrix}
0.5 \cdot \text{i}, & -0.5 + 1.5 \cdot \text{i}, & -1.0 + 1.0 \cdot \text{i} \\
0.40000000000000002 + 0.29999999999999999 \cdot \text{i}, & 0.89999999999999991 + 1.2999999999999998 \cdot \text{i}, & 0.20000000000000001 + 1.3999999999999999 \cdot \text{i} \\
0.40000000000000002, & -0.39999999999999997 + 0.40000000000000002 \cdot \text{i}, & 1.79999999999999998 \\
0.29999999999999999 - 0.40000000000000002 \cdot \text{i}, & 0.10000000000000001 + 0.69999999999999996 \cdot \text{i}, & 0. \\
-0.29999999999999999 \cdot \text{i}, & 0.29999999999999999 + 0.29999999999999999 \cdot \text{i}, & 2.39999999999999999
\end{bmatrix}
--R
--R
--R
--R
--E 114

--S 115 of 130
lda:=5
--R
--R
--R (8) 5
\end{verbatim}
---R Type: PositiveInteger
---E 115

---S 116 of 130
theta: Matrix Complex SF := [[0.0 , 0.0 , 0.0 ]]
---R
---R (9) [0. 0. 0.]
---R
---E 116

---S 117 of 130
ncolb:=2
---R
---R (10) 2
---E 117

---S 118 of 130
ldb:=5
---R
---R (11) 5
---E 118

---S 119 of 130
b: Matrix Complex SF :=
  [[-0.55 + 1.05*%i , 0.45 + 1.05*%i ],
   [ 0.49 + 0.93*%i , 1.09 + 0.13*%i ],
   [ 0.56 - 0.16*%i , 0.64 + 0.16*%i ],
   [ 0.39 + 0.23*%i , -0.39 - 0.23*%i ],
   [ 1.13 + 0.83*%i , -1.13 + 0.77*%i ]]
---R
---R (12)
---E
[1.1299999999999999 + 0.82999999999999996 %i,
-1.1299999999999999 + 0.77000000000000002 %i]
Type: Matrix(Complex(DoubleFloat))
ncolq := 2

theta: Matrix Complex SF :=
[[1 + 0.5*%i , 1.0954 - 0.3333*%i , 1.2649 - 1.1565*%i ]]

a: Matrix Complex SF :=
[[ 1 , 1 + 1*%i , 1 + %i ],
 [-0.2 - 0.4*%i , -2 , -1 - %i ],
 [-0.32 - 0.16*%i , -0.3505 + 0.2629*%i , -3 ],
 [-0.4 + 0.2*%i , 0.5477*%i , 0.0 ],
 [-0.12 + 0.24*%i , 0.1972 + 0.2629*%i , 0.6325 ]]
This package uses the NAG Library to provide facilities for matrix factorizations and associated transformations.

F01 -- Matrix Factorization

Chapter F01
Matrix Factorization

1. Scope of the Chapter

This chapter provides facilities for matrix factorizations and associated transformations.

2. Background to the Problems

An n by n matrix may be factorized as

\[ A = PLUQ \]

where L and U are respectively lower and upper triangular matrices, and P and Q are permutation matrices. This is called an LU factorization. For general dense matrices it is usual to choose Q=I and to then choose P to ensure that the factorization is numerically stable. For sparse matrices, judicious choice of P and Q ensures numerical stability as well as maintaining as much sparsity as possible in the factors L and U. The LU factorization is normally used in connection with the solution of the linear equations

\[ Ax = b, \]
whose solution, $x$, may then be obtained by solving in succession the simpler equations

$$T$$
$$Ly = Pb, \quad Uz = y, \quad x = Qz$$

the first by forward substitution and the second by backward substitution. Routines to perform this solution are to be found in Chapter F04.

When $A$ is symmetric positive-definite then we can choose $U=L$ and $Q=P$, to give the Cholesky factorization. This factorization is numerically stable without permutations, but in the sparse case the permutations can again be used to try to maintain sparsity. The Cholesky factorization is sometimes expressed as

$$T$$
$$A = PLDLT P,$$

where $D$ is a diagonal matrix with positive diagonal elements and $L$ is unit lower triangular.

The LU factorization can also be performed on rectangular matrices, but in this case it is more usual to perform a QR factorization. When $A$ is an $m$ by $n$ ($m \geq n$) matrix this is given by

$$T$$
$$(R)$$
$$A = Q(0),$$

where $R$ is an $n$ by $n$ upper triangular matrix and $Q$ is an orthogonal (unitary in the complex case) matrix.

3. Recommendations on Choice and Use of Routines

Routine F07ADF performs the LU factorization of a real $m$ by $n$ dense matrix.

The LU factorization of a sparse matrix is performed by routine F01BRF. Following the use of F01BRF, matrices with the same sparsity pattern may be factorized by routine F01BSF.

The Cholesky factorization of a real symmetric positive-definite dense matrix is performed by routine F07FDF.

Routine F01MCF performs the Cholesky factorization of a real symmetric positive-definite variable band (skyline) matrix, and a general sparse symmetric positive-definite matrix may be factorized using routine F01MAF.
The QR factorization of an m by n (m\geq n) matrix is performed by routine F01QCF in the real case, and F01RCF in the complex case. Following the use of F01QCF, operations with Q may be performed using routine F01QDF and some, or all, of the columns of Q may be formed using routine F01QEF. Routines F01RDF and F01REF perform the same tasks following the use of F01RCF.

F01 -- Matrix Factorizations

Chapter F01

Matrix Factorizations

F01BRF  LU factorization of real sparse matrix

F01BSF  LU factorization of real sparse matrix with known sparsity pattern

F01MAF  LL factorization of real sparse symmetric positive-definite matrix

F01MCF  LDL factorization of real symmetric positive-definite variable-bandwidth matrix

F01QCF  QR factorization of real m by n matrix (m\geq n)

F01QDF  Operations with orthogonal matrices, compute QB or Q^H B after factorization by F01QCF

F01QEF  Operations with orthogonal matrices, form columns of Q after factorization by F01QCF

F01RCF  QR factorization of complex m by n matrix (m\geq n)

F01RDF  Operations with unitary matrices, compute QB or Q^H B after factorization by F01RCF

F01REF  Operations with unitary matrices, form columns of Q after factorization by F01RCF
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01BRF factorizes a real sparse matrix. The routine either forms the LU factorization of a permutation of the entire matrix, or, optionally, first permutes the matrix to block lower triangular form and then only factorizes the diagonal blocks.

2. Specification

```fortran
SUBROUTINE F01BRF (N, NZ, A, LICN, IRN, LIRN, ICN, PIVOT,
  1             IKEEP, IW, LBLOCK, GROW, ABORT,
  2             IDISP, IFAIL)
  INTEGER N, NZ, LICN, IRN(LIRN), LIRN, ICN(LICN),
  1     IKEEP(5*N), IW(8*N), IDISP(10), IFAIL
  DOUBLE PRECISION A(LICN), PIVOT, W(N)
  LOGICAL LBLOCK, GROW, ABORT(4)
```

3. Description

Given a real sparse matrix A, this routine may be used to obtain the LU factorization of a permutation of A,

\[ PAQ = LU \]

where P and Q are permutation matrices, L is unit lower triangular and U is upper triangular. The routine uses a sparse variant of Gaussian elimination, and the pivotal strategy is designed to compromise between maintaining sparsity and controlling loss of accuracy through round-off.

Optionally the routine first permutes the matrix into block lower triangular form and then only factorizes the diagonal blocks. For some matrices this gives a considerable saving in storage and execution time.

Extensive data checks are made; duplicated non-zeros can be accumulated.

The factorization is intended to be used by F04AXF to solve \( T \) sparse systems of linear equations \( Ax=b \) or \( A^T x=b \). If several matrices of the same sparsity pattern are to be factorized, F01BSF should be used for the second and subsequent matrices.

The method is fully described by Duff [1].
4. References


5. Parameters

1: N -- INTEGER
   On entry: n, the order of the matrix A. Constraint: N > 0.

2: NZ -- INTEGER
   On entry: the number of non-zero elements in the matrix A. Constraint: NZ > 0.

3: A(LICN) -- DOUBLE PRECISION array
   On entry: A(i), for i = 1,2,...,NZ must contain the non-zero elements of the sparse matrix A. They can be in any order since the routine will reorder them. On exit: the non-zero elements in the LU factorization. The array must not be changed by the user between a call of this routine and a call of F04AXF.

4: LICN -- INTEGER
   On entry:
   the dimension of the arrays A and ICN as declared in the (sub)program from which F01BRF is called.
   Since the factorization is returned in A and ICN, LICN should be large enough to accommodate this and should ordinarily be 2 to 4 times as large as NZ. Constraint: LICN >= NZ.

5: IRN(LIRN) -- INTEGER array
   On entry: IRN(i), for i = 1,2,...,NZ must contain the row index of the non-zero element stored in A(i). On exit: the array is overwritten and is not needed for subsequent calls of F01BSF or F04AXF.

6: LIRN -- INTEGER
   On entry:
   the dimension of the array IRN as declared in the (sub)program from which F01BRF is called.
   It need not be as large as LICN; normally it will not need to be very much greater than NZ. Constraint: LIRN >= NZ.

7: ICN(LICN) -- INTEGER array
   On entry: ICN(i), for i = 1,2,...,NZ must contain the column index of the non-zero element stored in A(i). On exit: the column indices of the non-zero elements in the
factorization. The array must not be changed by the user between a call of this routine and subsequent calls of F01BSF or F04AXF.

8: PIVOT -- DOUBLE PRECISION Input
On entry: PIVOT should have a value in the range 0.0 <= PIVOT <= 0.9999 and is used to control the choice of pivots. If PIVOT < 0.0, the value 0.0 is assumed, and if PIVOT > 0.9999, the value 0.9999 is assumed. When searching a row for a pivot, any element is excluded which is less than PIVOT times the largest of those elements in the row available as pivots. Thus decreasing PIVOT biases the algorithm to maintaining sparsity at the expense of stability. Suggested value: PIVOT = 0.1 has been found to work well on test examples.

9: IKEEP(5*N) -- INTEGER array Output
On exit: indexing information about the factorization. The array must not be changed by the user between a call of this routine and calls of F01BSF or F04AXF.

10: IW(8*N) -- INTEGER array Workspace

11: W(N) -- DOUBLE PRECISION array Output
On exit: if GROW = .TRUE., W(1) contains an estimate (an upper bound) of the increase in size of elements encountered during the factorization (see GROW); the rest of the array is used as workspace.
If GROW = .FALSE., the array is not used.

12: LBLOCK -- LOGICAL Input
On entry: if LBLOCK = .TRUE., the matrix is pre-ordered into block lower triangular form before the LU factorization is performed; otherwise the entire matrix is factorized. Suggested value: LBLOCK = .TRUE. unless the matrix is known to be irreducible.

13: GROW -- LOGICAL Input
On entry: if GROW = .TRUE., then on exit W(1) contains an estimate (an upper bound) of the increase in size of elements encountered during the factorization. If the matrix is well-scaled (see Section 8.2), then a high value for W(1) indicates that the LU factorization may be inaccurate and the user should be wary of the results and perhaps increase the parameter PIVOT for subsequent runs (see Section 7). Suggested value: GROW = .TRUE..

14: ABORT(4) -- LOGICAL array Input
On entry:
if ABORT(1) = .TRUE., the routine will exit immediately on detecting a structural singularity (one that depends on the pattern of non-zeros) and return IFAIL = 1; otherwise it will complete the factorization (see Section 8.3).

If ABORT(2) = .TRUE., the routine will exit immediately on detecting a numerical singularity (one that depends on the numerical values) and return IFAIL = 2; otherwise it will complete the factorization (see Section 8.3).

If ABORT(3) = .TRUE., the routine will exit immediately (with IFAIL = 5) when the arrays A and ICN are filled up by the previously factorized, active and unfactorized parts of the matrix; otherwise it continues so that better guidance on necessary array sizes can be given in IDISP(6) and IDISP(7), and will exit with IFAIL in the range 4 to 6. Note that there is always an immediate error exit if the array IRN is too small.

If ABORT(4) = .TRUE., the routine exits immediately (with IFAIL = 13) if it finds duplicate elements in the input matrix. If ABORT(4) = .FALSE., the routine proceeds using a value equal to the sum of the duplicate elements. In either case details of each duplicate element are output on the current advisory message unit (see X04ABF), unless suppressed by the value of IFAIL on entry.

Suggested values:
ABORT(1) = .TRUE.
ABORT(2) = .TRUE.
ABORT(3) = .FALSE.
ABORT(4) = .TRUE..

15: IDISP(10) -- INTEGER array
Output
On exit: IDISP is used to communicate information about the factorization to the user and also between a call of F01BRF and subsequent calls to F01BSF or F04AXF.
IDISP(1) and IDISP(2), indicate the position in arrays A and ICN of the first and last elements in the LU factorization of the diagonal blocks. (IDISP(2) gives the number of non-zeros in the factorization.)
IDISP(3) and IDISP(4), monitor the adequacy of 'elbow
room' in the arrays IRN and A/ICN respectively, by giving the number of times that the data in these arrays has been compressed during the factorization to release more storage. If either IDISP(3) or IDISP(4) is quite large (say greater than 10), it will probably pay the user to increase the size of the corresponding array(s) for subsequent runs. If either is very low or zero, then the user can perhaps save storage by reducing the size of the corresponding array(s).

IDISP(5), gives an upper bound on the rank of the matrix.

IDISP(6) and IDISP(7), give the minimum size of arrays IRN and A/ICN respectively which would enable a successful run on an identical matrix (but some 'elbow-room' should be allowed - see Section 8).

IDISP(8) to (10), are only used if LBLOCK = .TRUE..

IDISP(8), gives the structural rank of the matrix.

IDISP(9), gives the number of diagonal blocks.

IDISP(10), gives the size of the largest diagonal block.

IDISP(1) and IDISP(2), must not be changed by the user between a call of F01BRF and subsequent calls to F01BSF or F04AXF.

16: IFAIL -- INTEGER Input/Output
For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).

Before entry, IFAIL must be set to a value with the decimal expansion cba, where each of the decimal digits c, b and a must have a value of 0 or 1.

a=0 specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages will be printed (see Section 6);
c=0 suppresses warning messages, otherwise warning messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

IFAIL = -2
Successful factorization of a numerically singular matrix (which may also be structurally singular) (see Section 8.3).

IFAIL = -1
Successful factorization of a structurally singular matrix (see Section 8.3).

IFAIL = 1
The matrix is structurally singular and the factorization has been abandoned (ABORT(1) was .TRUE. on entry).

IFAIL = 2
The matrix is numerically singular and the factorization has been abandoned (ABORT(2) was .TRUE. on entry).

IFAIL = 3
LIRN is too small: there is not enough space in the array IRN to continue the factorization. The user is recommended to try again with LIRN (and the length of IRN) equal to at least IDISP(6) + N/2.

IFAIL = 4
LICN is much too small: there is much too little space in the arrays A and ICN to continue the factorization.

IFAIL = 5
LICN is too small: there is not enough space in the arrays A and ICN to store the factorization. If ABORT(3) was .FALSE. on entry, the factorization has been completed but some of the LU factors have been discarded to create space, IDISP(7) then gives the minimum value of LICN (i.e., the minimum length of A and ICN) required for a successful factorization of the same matrix.

IFAIL = 6
LICN and LIRN are both too small: effectively this is a combination of IFAIL = 3 and IFAIL = 5 (with ABORT(3) = .FALSE.).
IFAIL= 7
LICN is too small: there is not enough space in the arrays A
and ICN for the permutation to block triangular form.

IFAIL= 8
On entry N <= 0.

IFAIL= 9
On entry NZ <= 0.

IFAIL= 10
On entry LICN < NZ.

IFAIL= 11
On entry LIRN < NZ.

IFAIL= 12
On entry an element of the input matrix has a row or column
index (i.e., an element of IRN or ICN) outside the range 1
to N.

IFAIL= 13
Duplicate elements have been found in the input matrix and
the factorization has been abandoned (ABORT(4) = .TRUE. on
entry).

7. Accuracy

The factorization obtained is exact for a perturbed matrix whose
(i,j)th element differs from a by less than 3(epsilon)(rho)m
ij
where (epsilon) is the machine precision, (rho) is the growth
value returned in W(1) if GROW = .TRUE., and m the number of
ij
Gaussian elimination operations applied to element (i,j). The
value of m is not greater than n and is usually much less.
ij
Small (rho) values therefore guarantee accurate results, but
unfortunately large (rho) values may give a very pessimistic
indication of accuracy.

8. Further Comments

8.1. Timing

The time required may be estimated very roughly from the number
(tau) of non-zeros in the factorized form (output as IDISP(2))
and for this routine and its associates is
where our unit is the time for the inner loop of a full matrix code (e.g. solving a full set of equations takes about \(-n^{1.3}\) units). Note that the faster F01BSF time makes it well worthwhile to use this for a sequence of problems with the same pattern.

It should be appreciated that \((\tau)\) varies widely from problem to problem. For network problems it may be little greater than \(NZ\), the number of non-zeros in \(A\); for discretisation of 2-dimensional and 3-dimensional partial differential equations it may be about \(1.5n\) \(3n\log n\) and \(-n^{2/3}\), respectively.

The time taken to find the block lower triangular form (\(\text{LBLOCK} = \text{it is not found (\text{LBLOCK} = \text{.FALSE.}). If the matrix is irreducible (IDISP(9) = 1 after a call with LBLOCK = \text{.TRUE.}) then this time is wasted. Otherwise, particularly if the largest block is small (IDISP(10)<<n), the consequent savings are likely to be greater.\}

The time taken to estimate growth (\(\text{GROW} = \text{.TRUE.}) is typically under 2\% of the overall time.

The overall time may be substantially increased if there is inadequate 'elbow-room' in the arrays \(A\), IRN and ICN. When the sizes of the arrays are minimal (IDISP(6) and IDISP(7)) it can execute as much as three times slower. Values of IDISP(3) and IDISP(4) greater than about 10 indicate that it may be worthwhile to increase array sizes.

8.2. Scaling

The use of a relative pivot tolerance \(\text{PIVOT}\) essentially presupposes that the matrix is well-scaled, i.e., that the matrix elements are broadly comparable in size. Practical problems are often naturally well-scaled but particular care is needed for problems containing mixed types of variables (for example millimetres and neutron fluxes).

8.3. Singular and Rectangular Systems
It is envisaged that this routine will almost always be called for square non-singular matrices and that singularity indicates an error condition. However, even if the matrix is singular it is possible to complete the factorization. It is even possible for F04AXF to solve a set of equations whose matrix is singular provided the set is consistent.

Two forms of singularity are possible. If the matrix would be singular for any values of the non-zeros (e.g. if it has a whole row of zeros), then we say it is structurally singular, and continue only if ABORT(1) = .FALSE.. If the matrix is non-singular by virtue of the particular values of the non-zeros, then we say that it is numerically singular and continue only if ABORT(2) = .FALSE..

Rectangular matrices may be treated by setting N to the larger of the number of rows and numbers of columns and setting ABORT(1) = .FALSE.

Note: the soft failure option should be used (last digit of IFAIL = 1) if the user wishes to factorize singular matrices with ABORT(1) or ABORT(2) set to .FALSE..

8.4. Duplicated Non-zeros

The matrix A may consist of a sum of contributions from different sub-systems (for example finite elements). In such cases the user may rely on this routine to perform assembly, since duplicated elements are summed.

9. Example

To factorize the real sparse matrix:

\[
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & -1 & 2 & 0 & 0 \\
0 & 0 & 3 & 0 & 0 & 0 \\
-2 & 0 & 0 & 1 & 1 & 0 \\
-1 & 0 & 0 & -1 & 2 & -3 \\
-1 & -1 & 0 & 0 & 0 & 6
\end{pmatrix}
\]

This example program simply prints out some information about the factorization as returned by F01BRF in W(1) and IDISP. Normally the call of F01BRF would be followed by a call of F04AXF (see Example for F04AXF).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F01 -- Matrix Factorizations
F01BSF

F01BSF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01BSF factorizes a real sparse matrix using the pivotal sequence previously obtained by F01BRF when a matrix of the same sparsity pattern was factorized.

2. Specification

```fortran
SUBROUTINE F01BSF (N, NZ, A, LICN, IVECT, JVECT, ICN, IKEEP, IW, W, GROW, ETA, RPMIN, ABORT,
  2 IDISP, IFAIL)
  INTEGER N, NZ, LICN, IVECT(NZ), JVECT(NZ), ICN
  1 (LICN), IKEEP(5*N), IW(8*N), IDISP(2),
  2 IFAIL
  DOUBLE PRECISION A(LICN), W(N), ETA, RPMIN
  LOGICAL GROW, ABORT
```

3. Description

This routine accepts as input a real sparse matrix of the same sparsity pattern as a matrix previously factorized by a call of F01BRF. It first applies to the matrix the same permutations as were used by F01BRF, both for permutation to block triangular form and for pivoting, and then performs Gaussian elimination to obtain the LU factorization of the diagonal blocks.

Extensive data checks are made; duplicated non-zeros can be accumulated.

The factorization is intended to be used by F04AXF to solve $T$ sparse systems of linear equations $Ax=b$ or $A^T x=b$.

F01BSF is much faster than F01BRF and in some applications it is expected that there will be many calls of F01BSF for each call of F01BRF.

The method is fully described in Duff [1].

4. References
5. Parameters

1: N -- INTEGER Input
On entry: n, the order of the matrix A. Constraint: N > 0.

2: NZ -- INTEGER Input
On entry: the number of non-zeros in the matrix A.
Constraint: NZ > 0.

3: A(LICN) -- DOUBLE PRECISION array Input/Output
On entry: A(i), for i = 1,2,...,NZ must contain the non-zero elements of the sparse matrix A. They can be in any order since the routine will reorder them. On exit: the non-zero elements in the factorization. The array must not be changed by the user between a call of this routine and a call of F04AXF.

4: LICN -- INTEGER Input
On entry:
the dimension of the arrays A and ICN as declared in the (sub)program from which F01BSF is called.
It should have the same value as it had for F01BRF.
Constraint: LICN >= NZ.

5: IVECT(NZ) -- INTEGER array Input

6: JVECT(NZ) -- INTEGER array Input
On entry: IVECT(i) and JVECT(i), for i = 1,2,...,NZ must contain the row index and the column index respectively of the non-zero element stored in A(i).

7: ICN(LICN) -- INTEGER array Input
On entry: the same information as output by F01BRF. It must not be changed by the user between a call of this routine and a call of F04AXF.

8: IKEEP(5*N) -- INTEGER array Input
On entry: the same indexing information about the factorization as output from F01BRF. It must not be changed between a call of this routine and a call of F04AXF.

9: IW(8*N) -- INTEGER array Workspace

10: W(N) -- DOUBLE PRECISION array Output
On exit: if GROW = .TRUE., W(1) contains an estimate (an upper bound) of the increase in size of elements encountered
during the factorization (see GROW); the rest of the array is used as workspace.

If GROW = .FALSE., the array is not used.

11: GROW -- LOGICAL
Input
On entry: if GROW = .TRUE., then on exit W(1) contains an estimate (an upper bound) of the increase in size of elements encountered during the factorization. If the matrix is well-scaled (see Section 8.2), then a high value for W(1) indicates that the LU factorization may be inaccurate and the user should be wary of the results and perhaps increase the parameter PIVOT for subsequent runs (see Section 7).

12: ETA -- DOUBLE PRECISION
Input
On entry: the relative pivot threshold below which an error diagnostic is provoked and IFAIL is set to 7. If ETA is greater than 1.0, then no check on pivot size is made.
-4
Suggested value: ETA = 10.

13: RPMIN -- DOUBLE PRECISION
Output
On exit: if ETA is less than 1.0, then RPMIN gives the smallest ratio of the pivot to the largest element in the row of the corresponding upper triangular factor thus monitoring the stability of the factorization. If RPMIN is very small it may be advisable to perform a new factorization using F01BRF.

14: ABORT -- LOGICAL
Input
On entry: if ABORT = .TRUE., the routine exits immediately (with IFAIL = 8) if it finds duplicate elements in the input matrix. If ABORT = .FALSE., the routine proceeds using a value equal to the sum of the duplicate elements. In either case details of each duplicate element are output on the current advisory message unit (see X04ABF), unless suppressed by the value of IFAIL on entry. Suggested value: ABORT = .TRUE..

15: IDISP(2) -- INTEGER array
Input
On entry: IDISP(1) and IDISP(2) must be unchanged since the previous call of F01BRF.

16: IFAIL -- INTEGER
Input/Output
For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential Introduction).

Before entry, IFAIL must be set to a value with the decimal
expansion cba, where each of the decimal digits c, b and a must have a value of 0 or 1.

- \( a=0 \) specifies hard failure, otherwise soft failure;
- \( b=0 \) suppresses error messages, otherwise error messages will be printed (see Section 6);
- \( c=0 \) suppresses warning messages, otherwise warning messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

**IFAIL= 1**

On entry \( N \leq 0 \).

**IFAIL= 2**

On entry \( NZ \leq 0 \).

**IFAIL= 3**

On entry \( LICN < NZ \).

**IFAIL= 4**

On entry an element of the input matrix has a row or column index (i.e., an element of IVECT or JVECT) outside the range 1 to \( N \).

**IFAIL= 5**

The input matrix is incompatible with the matrix factorized by the previous call of F01BRF (see Section 8).

**IFAIL= 6**

The input matrix is numerically singular.

**IFAIL= 7**

A very small pivot has been detected (see Section 5, ETA). The factorization has been completed but is potentially unstable.

**IFAIL= 8**
Duplicate elements have been found in the input matrix and the factorization has been abandoned (ABORT = .TRUE. on entry).

7. Accuracy

The factorization obtained is exact for a perturbed matrix whose (i,j)th element differs from a by less than $3(\epsilon)(\rho)m_{ij}$

where $\epsilon$ is the machine precision, $\rho$ is the growth value returned in $W(1)$ if $\text{GROW} = .\text{TRUE.}$, and $m$ the number of $ij$

Gaussian elimination operations applied to element $(i,j)$.

If $\rho = W(1)$ is very large or RPMIN is very small, then a fresh call of F01BRF is recommended.

8. Further Comments

If the user has a sequence of problems with the same sparsity pattern then this routine is recommended after F01BRF has been called for one such problem. It is typically 4 to 7 times faster but is potentially unstable since the previous pivotal sequence is used. Further details on timing are given in document F01BRF.

If growth estimation is performed (GROW = .TRUE.), then the time increases by between 5% and 10%. Pivot size monitoring (ETA <= 1.0) involves a similar overhead.

We normally expect this routine to be entered with a matrix having the same pattern of non-zeros as was earlier presented to F01BRF. However there is no record of this pattern, but rather a record of the pattern including all fill-ins. Therefore we permit additional non-zeros in positions corresponding to fill-ins.

If singular matrices are being treated then it is also required that the present matrix be sufficiently like the previous one for the same permutations to be suitable for factorization with the same set of zero pivots.

9. Example

To factorize the real sparse matrices

\[
\begin{pmatrix}
5 & 0 & 0 & 0 & 0 \\
0 & 2 & -1 & 2 & 0 \\
0 & 0 & 3 & 0 & 0 \\
-2 & 0 & 0 & 1 & 1 \\
-1 & 0 & 0 & -1 & 2 \\
-1 & -1 & 0 & 0 & 6
\end{pmatrix}
\]
This example program simply prints the values of $W(1)$ and $\text{RPMIN}$ returned by F01BSF. Normally the calls of F01BRF and F01BSF would be followed by calls of F04AXF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

F01MAF computes an incomplete Cholesky factorization of a real sparse symmetric positive-definite matrix $A$.

F01MAF computes an incomplete Cholesky factorization

\[ C = PLDL^T, \quad WAW = C + E \]
for the sparse symmetric positive-definite matrix A, where P is a permutation matrix, L is a unit lower triangular matrix, D is a diagonal matrix with positive diagonal elements, E is an error matrix representing elements dropped during the factorization and diagonal elements that have been modified to ensure that C is positive-definite, and W is a diagonal matrix, chosen to make the diagonal elements of $WAW$ unity.

$$W^{-1}$$

$W^{-1}CW$ is a pre-conditioning matrix for A, and the factorization of C is intended to be used by F04MAF to solve systems of linear equations $Ax=b$.

The permutation matrix P is chosen to reduce the amount of fill-in that occurs in L and the user-supplied parameter DROPTL can also be used to control the amount of fill-in that occurs.

Full details on the factorization can be found in Munksgaard [1].

F01MAF is based on the Harwell Library routine MA31A.

4. References


5. Parameters

1: N -- INTEGER Input
   On entry: n, the order of the matrix A. Constraint: $N \geq 1$.

2: NZ -- INTEGER Input
   On entry: the number of non-zero elements in the upper triangular part of the matrix A, including the number of elements on the leading diagonal. Constraint: $NZ \geq N$.

3: A(LICN) -- DOUBLE PRECISION array Input/Output
   On entry: the first NZ elements of the array A must contain the non-zero elements of the upper triangular part of the sparse positive-definite symmetric matrix A, including the elements on the leading diagonal. On exit: the first $(NZ-N)$ elements of A contain the elements above the diagonal of the matrix $WAW$, where W is a diagonal matrix whose $i$th diagonal element is $w = a_{ii}^{-1/2}$. These elements are returned in order by rows and the value returned in ICN(k) gives the column index of the element returned in A(k). The value $w$ is returned in $i$
the $i$th element of the array WKEEP. The remaining $\text{LROW}-\text{NZ}+\text{N}$ elements of $A$, where $\text{LROW}$ is the value returned in INFORM(1), return details of the factorization for use by F04MAF.

4: LICN -- INTEGER
On entry:
the dimension of the array $A$ as declared in the (sub)program from which F01MAF is called.
If fill-in is expected during the factorization, then a larger value of LICN will allow fewer elements to be dropped during the factorization, thus giving a more accurate factorization, which in turn will almost certainly mean that fewer iterations will be required by F04MAF. Constraint: $\text{LICN} \geq 2 \times \text{NZ}$.

5: IRN(LIRN) -- INTEGER array
On entry: IRN($k$), for $k = 1,2,\ldots,\text{NZ}$ must contain the row index of the non-zero element of the matrix $A$ supplied in $A$ ($k$). On exit: the first $L\text{COL}$ elements of IRN, where $L\text{COL}$ is the value returned in INFORM(2), return details of the factorization for use by F04MAF.

6: LIRN -- INTEGER
On entry:
the dimension of the array IRN as declared in the (sub)program from which F01MAF is called.
$L\text{IRN}$ must be at least $\text{NZ}$, but, as with LICN, if fill-in is expected then a larger value of LIRN will allow a more accurate factorization. For this purpose LIRN should exceed $\text{NZ}$ by the same amount that LICN exceeds $2 \times \text{NZ}$. Constraint: $L\text{IRN} \geq \text{NZ}$.

7: ICN(LICN) -- INTEGER array
On entry: ICN($k$), for $k = 1,2,\ldots,\text{NZ}$ must contain the column index of the non-zero element of the matrix $A$ supplied in $A$ ($k$). Thus $a_{ij} = A(k)$, where $i = \text{IRN}(k)$ and $j = \text{ICN}(k)$. On exit: the first ($\text{NZ}-\text{N}$) elements of ICN give the column indices of the first ($\text{NZ}-\text{N}$) elements returned in $A$. The remaining $\text{LROW} - \text{NZ} + \text{N}$ elements of ICN return details of the factorization for use by F04MAF.

8: DROPTL -- DOUBLE PRECISION
On entry: a value in the range $[-1.0,1.0]$ to be used as a tolerance in deciding whether or not to drop elements during the factorization. At the $k$th pivot step the element $a_{ij}$ is dropped if it would cause fill-in and if
If DROPTL is supplied as negative, then it is not altered during the factorization and so is unchanged on exit, but if DROPTL is supplied as positive then it may be altered by the routine with the aim of obtaining an accurate factorization in the space available. If DROPTL is supplied as -1.0, then no fill-in will occur during the factorization; and if DROPTL is supplied as 0.0 then a complete factorization is performed. On exit: may be overwritten with the value used by the routine in order to obtain an accurate factorization in the space available, if DROPTL > 0.0 on entry.

9: DENS W -- DOUBLE PRECISION Input/Output
On entry: a value in the range [0.0,1.0] to be used in deciding whether or not to regard the active part of the matrix at the kth pivot step as being full. If the ratio of non-zero elements to the total number of elements is greater than or equal to DENS W, then the active part is regarded as full. If DENS W < 1.0, then the storage used is likely to increase compared to the case where DENS W = 0, but the execution time is likely to decrease. Suggested value: DENS W = 0.8. On exit: if on entry DENS W is not in the range [0.0,1.0], then it is set to 0.8. Otherwise it is unchanged.

10: WKEEP(3*N) -- DOUBLE PRECISION array Output
On exit: information which must be passed unchanged to F04MAF. The first N elements contain the values w_i for i=1,2,...,n, and the next N elements contain the diagonal elements of D.

11: IKEEP(2*N) -- INTEGER array Output
On exit: information which must be passed unchanged to F04MAF.

12: IWORK(6*N) -- INTEGER array Workspace

13: ABORT(3) -- LOGICAL array Input
On entry:
if ABORT(1) = .TRUE., the routine will exit immediately on detecting duplicate elements and return IFAIL = 5. Otherwise when ABORT(1) = .FALSE., the calculations will continue using the sum of the duplicate entries. In either case details of the duplicate elements are output on the current advisory message unit (see X04ABF), unless suppressed by the value of IFAIL on entry.
If ABORT(2) = .TRUE., the routine will exit immediately on detecting a zero or negative pivot element and return IFAIL = 6. Otherwise when ABORT(2) = .FALSE., the zero or negative pivot element will be modified to ensure positive-definiteness and a message will be printed on the current advisory message unit, unless suppressed by the value of IFAIL on entry.

If ABORT(3) = .TRUE., the routine will exit immediately if the arrays A and ICN have been filled up and return IFAIL = 7. Otherwise when ABORT(3) = .FALSE., the data in the arrays is compressed to release more storage and a message will be printed on the current advisory message unit, unless suppressed by the value of IFAIL on entry. If DROPTL is positive on entry, it may be modified in order to allow a factorization to be completed in the available space.

Suggested values:
ABORT(1) = .TRUE.,

ABORT(2) = .TRUE.,

ABORT(3) = .TRUE..

14: INFORM(4) -- INTEGER array Output
On exit:
INFORM(1) returns the number of elements of A and ICN that have been used by the routine. Thus at least the first INFORM(1) elements of A and of ICN must be supplied to F04MAF.

Similarly, INFORM(2) returns the number of elements of IRN that have been used by the routine and so at least the first INFORM(2) elements must be supplied to F04MAF.

INFORM(3) returns the number of entries supplied in A that corresponded to diagonal and duplicate elements. If no duplicate entries were found, then INFORM(3) will return the value of N.

INFORM(4) returns the value k of the pivot step from which the active matrix was regarded as full. INFORM must be passed unchanged to F04MAF.

15: IFAIL -- INTEGER Input/Output
For this routine, the normal use of IFAIL is extended to control the printing of error and warning messages as well as specifying hard or soft failure (see the Essential
Before entry, IFAIL must be set to a value with the decimal expansion cba, where each of the decimal digits c, b and a must have a value of 0 or 1.

- \(a=0\) specifies hard failure, otherwise soft failure;
- \(b=0\) suppresses error messages, otherwise error messages will be printed (see Section 6);
- \(c=0\) suppresses warning messages, otherwise warning messages will be printed (see Section 6).

The recommended value for inexperienced users is 110 (i.e., hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

**IFAIL= 1**

On entry \(N < 1\),

- or \(NZ < N\),
- or \(LIRN < NZ\),
- or \(LICN < 2 \times NZ\).

**IFAIL= 2**

One of the conditions \(0 < IRN(k) \leq ICN(k) \leq N\) is not satisfied so that \(A(k)\) is not in the upper triangle of the matrix. No further computation is attempted.

**IFAIL= 3**

One of the diagonal elements of the matrix \(A\) is zero or negative so that \(A\) is not positive-definite. No further computation is attempted.

**IFAIL= 4**

The available space has been used and no further compressions are possible. The user should either increase DROPTL, or allocate more space to \(A\), IRN and ICN.
For all the remaining values of IFAIL the computations will continue in the case of soft failure, so that more than one advisory message may be printed.

IFAIL = 5
Duplicate elements have been detected and ABORT(1) = .TRUE..

IFAIL = 6
A zero or negative pivot element has been detected during the factorization and ABORT(2) = .TRUE..

This should not happen if \( A \) is an M-matrix (see Munksgaard [1]), but may occur for other types of positive-definite matrix.

IFAIL = 7
The available space has been used and ABORT(3) = .TRUE..

7. Accuracy

The accuracy of the factorization will be determined by the size of the elements that are dropped and the size of the modifications made to the diagonal elements. If these sizes are small then the computed factors will correspond to a matrix close to \( A \) and the number of iterations required by F04MAF will be small. The more incomplete the factorization, the higher the number of iterations required by F04MAF.

8. Further Comments

The time taken by the routine will depend upon the sparsity pattern of the matrix and the number of fill-ins that occur during the factorization. At the very least the time taken can be expected to be roughly proportional to \( n(\tau) \), where \( (\tau) \) is the number of non-zeros.

The routine is intended for use with positive-definite matrices, but the user is warned that it will not necessarily detect non-positive-definiteness. Indeed the routine may return a factorization that can satisfactorily be used by F04MAF even when \( A \) is not positive-definite, but this should not be relied upon as F04MAF may not converge.

9. Example

The example program illustrates the use of F01MAF in conjunction with F04MAF to solve the 16 linear equations \( Ax=b \), where

\[
\begin{pmatrix}
1 & z & z \\
z & 1 & z \\
z & z & 1
\end{pmatrix}
\]
The matrix $A=(z \begin{pmatrix} 1 & 0 & z \\ z & 0 & 1 & z \\ z & z & 1 & z \\ z & z & z & 1 \\ z & z & 1 & 0 \\ z & 0 & 1 & z \\ z & z & 1 \\ z & z & 1 & \end{pmatrix})$ arises in the solution of Laplace's equation in a unit square, using a 5-point formula with a 6 by 6 discretisation, with unity on the boundaries.

The drop tolerance, DROPTL, is taken as 0.1, and the density factor, DENSW, is taken as 0.8. The value IFAIL = 111 is used so that advisory and error messages will be printed, but soft failure would occur if IFAIL were returned as non-zero.

A relative accuracy of about 0.0001 is requested in the solution from F04MAF, with a maximum of 50 iterations.

The example program for F02FJF illustrates the use of F01MAF and F04MAF in solving an eigenvalue problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

F01 -- Matrix Factorizations

F01MCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

F01MCF computes the Cholesky factorization of a real symmetric positive-definite variable-bandwidth matrix.

2. Specification

SUBROUTINE F01MCF (N, A, LAL, NROW, AL, D, IFAIL)
INTEGER N, LAL, NROW(N), IFAIL
DOUBLE PRECISION A(LAL), AL(LAL), D(N)

3. Description

This routine determines the unit lower triangular matrix $L$ and

$\begin{bmatrix}
\end{bmatrix}

$T$

the diagonal matrix $D$ in the Cholesky factorization $A=LDL^T$ of a symmetric positive-definite variable-bandwidth matrix $A$ of order $n$. (Such a matrix is sometimes called a 'sky-line' matrix.)

The matrix $A$ is represented by the elements lying within the envelope of its lower triangular part, that is, between the first non-zero of each row and the diagonal (see Section 9 for an example). The width $\text{NROW}(i)$ of the $i$th row is the number of elements between the first non-zero element and the element on the diagonal, inclusive. Although, of course, any matrix possesses an envelope as defined, this routine is primarily intended for the factorization of symmetric positive-definite matrices with an average bandwidth which is small compared with $n$ (also see Section 8).

The method is based on the property that during Cholesky factorization there is no fill-in outside the envelope.

The determination of $L$ and $D$ is normally the first of two steps in the solution of the system of equations $Ax=b$. The remaining $T$ step, viz. the solution of $LDL^T x=b$ may be carried out using F04MCF.

4. References


5. Parameters
1: N -- INTEGER  
   Input 
   On entry: n, the order of the matrix A. Constraint: N >= 1.

2: A(LAL) -- DOUBLE PRECISION array  
   Input 
   On entry: the elements within the envelope of the lower 
   triangle of the positive-definite symmetric matrix A, taken 
   in row by row order. The following code assigns the matrix 
   elements within the envelope to the correct elements of the 
   array:

   K = 0
   DO 20 I = 1, N
   DO 10 J = I-NROW(I)+1, I
   K = K + 1
   A(K) = matrix (I,J)
   10 CONTINUE
   20 CONTINUE

   See also Section 8.

3: LAL -- INTEGER  
   Input 
   On entry: the smaller of the dimensions of the arrays A and 
   AL as declared in the calling (sub)program from which F01MCF 
   is called. Constraint: LAL >= NROW(1) + NROW(2) +... + NROW( 
   n).

4: NROW(N) -- INTEGER array  
   Input 
   On entry: NROW(i) must contain the width of row i of the 
   matrix A, i.e., the number of elements between the first 
   (leftmost) non-zero element and the element on the diagonal, 
   inclusive. Constraint: 1 <= NROW(i) <= i, for i=1,2,...,n.

5: AL(LAL) -- DOUBLE PRECISION array  
   Output 
   On exit: the elements within the envelope of the lower 
   triangular matrix L, taken in row by row order. The envelope 
   of L is identical to that of the lower triangle of A. The 
   unit diagonal elements of L are stored explicitly. See also 
   Section 8.

6: D(N) -- DOUBLE PRECISION array  
   Output 
   On exit: the diagonal elements of the the diagonal matrix D 
   . Note that the determinant of A is equal to the product of 
   these diagonal elements. If the value of the determinant is 
   required it should not be determined by forming the product 
   explicitly, because of the possibility of overflow or 
   underflow. The logarithm of the determinant may safely be 
   formed from the sum of the logarithms of the diagonal 
   elements.
7: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry N < 1,
       or for some i, NROW(i) < 1 or NROW(i) > i,
       or LAL < NROW(1) + NROW(2) +... + NROW(N).

IFAIL= 2
   A is not positive-definite, or this property has been
   destroyed by rounding errors. The factorization has not been
   completed.

IFAIL= 3
   A is not positive-definite, or this property has been
   destroyed by rounding errors. The factorization has been
   completed but may be very inaccurate (see Section 7).

7. Accuracy

If IFAIL = 0 on exit, then the computed L and D satisfy the
relation LDL =A+F, where

\[ \|F\| \leq km \max_{i} \epsilon_{a,ii} \]

and

\[ \|F\| \leq km \epsilon_{\|A\|} \]

where k is a constant of order unity, m is the largest value of
NROW(i), and \( \epsilon_{\|A\|} \) is the machine precision. See Wilkinson
and Reinsch [2], pp 25--27, 54--55. If IFAIL = 3 on exit, then
the factorization has been completed although the matrix was not
positive-definite. However the factorization may be very
inaccurate and should be used only with great caution. For instance, if it is used to solve a set of equations \( Ax = b \) using F04MCF, the residual vector \( b - Ax \) should be checked.

8. Further Comments

The time taken by the routine is approximately proportional to the sum of squares of the values of NROW(i).

The distribution of row widths may be very non-uniform without undue loss of efficiency. Moreover, the routine has been designed to be as competitive as possible in speed with routines designed for full or uniformly banded matrices, when applied to such matrices.

Unless otherwise stated in the Users' Note for your implementation, the routine may be called with the same actual array supplied for parameters A and AL, in which case L overwrites the lower triangle of A. However this is not standard Fortran 77 and may not work in all implementations.

9. Example

To obtain the Cholesky factorization of the symmetric matrix, whose lower triangle is:

\[
\begin{pmatrix}
1 & & & \\
2 & 5 & & \\
0 & 3 & 13 & \\
0 & 0 & 0 & 16 \\
5 & 14 & 18 & 8 & 55 \\
0 & 0 & 0 & 24 & 17 & 77
\end{pmatrix}
\]

For this matrix, the elements of NROW must be set to 1, 2, 2, 1, 5, 3, and the elements within the envelope must be supplied in row order as:

1, 2, 5, 3, 13, 16, 5, 14, 18, 8, 55, 24, 17, 77.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01QCF finds the QR factorization of the real m by n matrix A, where m>=n.

2. Specification

SUBROUTINE F01QCF (M, N, A, LDA, ZETA, IFAIL)
INTEGER M, N, LDA, IFAIL
DOUBLE PRECISION A(LDA,*), ZETA(*)

3. Description

The m by n matrix A is factorized as

(R)
A=Q(0) when m>n,
A=QR when m=n,

where Q is an m by m orthogonal matrix and R is an n by n upper triangular matrix. The factorization is obtained by Householder’s method. The kth transformation matrix, Q, which is used to introduce zeros into the kth column of A is given in the form

(I 0 )
Q =(0 T )
k ( k)

where

T
T =I-u u ,
k k k

((zeta) )
( k)
u = (z ),
k ( k )

(zeta) is a scalar and z is an (m-k) element vector. (zeta)
k k k
and z are chosen to annihilate the elements below the triangular part of A.
The vector u is returned in the kth element of the array ZETA
k
and in the kth column of A, such that (zeta) is in ZETA(k) and
k
the elements of z are in A(k+1,k),...,A(m,k). The elements of R
k
are returned in the upper triangular part of A.

Q is given by

\[ Q^T = (Q \ Q \ ... \ Q) \]

\[ n \ n-1 \ 1 \]

Good background descriptions to the QR factorization are given in
Dongarra et al [1] and Golub and Van Loan [2], but note that this
routine is not based upon LINPACK routine DQRDC.

4. References

LINPACK Users’ Guide. SIAM, Philadelphia.


Oxford University Press.

5. Parameters

1: M -- INTEGER Input
   On entry: m, the number of rows of A. Constraint: M >= N.

2: N -- INTEGER Input
   On entry: n, the number of columns of A.
   When N = 0 then an immediate return is effected.
   Constraint: N >= 0.

3: A(LDA,*) -- DOUBLE PRECISION array Input/Output
   Note: the second dimension of the array A must be at least
   max(1,n).
   On entry: the leading m by n part of the array A must
   contain the matrix to be factorized. On exit: the n by n
   upper triangular part of A will contain the upper triangular
   matrix R and the m by n strictly lower triangular part of A
   will contain details of the factorization as described in
4: LDA -- INTEGER  
   Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which F01QCF is called.
   Constraint: LDA >= max(1,M).

5: ZETA(*) -- DOUBLE PRECISION array  
   Output
   Note: the dimension of the array ZETA must be at least max
   (1,n) On exit: ZETA(k) contains the scalar (zeta) for the k
   th transformation. If T =I then ZETA(k)=0.0, otherwise ZETA(
   k) contains (zeta) as described in Section 3 and (zeta) is
   k
   always in the range (1.0, \sqrt{2.0}).

6: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
   On entry M < N,
   or N < 0,
   or LDA < M.

7. Accuracy

The computed factors Q and R satisfy the relation

(R)
Q(0)=A+E,

where
\[ \| \|E\|\| \leq c(\text{epsilon})\| \|A\|\|, \]

and \((\text{epsilon})\) is the machine precision (see X02AJF(*)), \(c\) is a modest function of \(m\) and \(n\) and \(\| \| \| \|\) denotes the spectral (two) norm.

8. Further Comments

The approximate number of floating-point operations is given by
\[
2 \left( 2n \frac{(3m-n)}{3} \right).
\]

Following the use of this routine the operations
\[
T \\
B := QB \text{ and } B := Q B,
\]

where \(B\) is an \(m\) by \(k\) matrix, can be performed by calls to F01QDF.

The operation \(B := QB\) can be obtained by the call:

\[
\begin{align*}
\text{IFAIL} & = 0 \\
\text{CALL F01QDF('No transpose', 'Separate', M, N, A, LDA, ZETA,} \\
* & \text{K, B, LDB, WORK, IFAIL)}
\end{align*}
\]

\[T\]

and \(B := QB\) can be obtained by the call:

\[
\begin{align*}
\text{IFAIL} & = 0 \\
\text{CALL F01QDF('Transpose', 'Separate', M, N, A, LDA, ZETA,} \\
* & \text{K, B, LDB, WORK, IFAIL)}
\end{align*}
\]

In both cases \(\text{WORK}\) must be a \(k\) element array that is used as workspace. If \(B\) is a one-dimensional array (single column) then the parameter LDB can be replaced by \(M\). See F01QDF for further details.

The first \(k\) columns of the orthogonal matrix \(Q\) can either be obtained by setting \(B\) to the first \(k\) columns of the unit matrix and using the first of the above two calls, or by calling F01QEF, which overwrites the \(k\) columns of \(Q\) on the first \(k\) columns of the array \(A\). \(Q\) is obtained by the call:

\[
\text{CALL F01QEF('Separate', M, N, K, A, LDA, ZETA, WORK, IFAIL)}
\]

As above \(\text{WORK}\) must be a \(k\) element array. If \(k\) is larger than \(N\), then \(A\) must have been declared to have at least \(k\) columns.
Operations involving the matrix $R$ can readily be performed by the Level 2 BLAS routines DTRSV and DTRMV (see Chapter F06), but note that no test for near singularity of $R$ is incorporated in DTRSV. If $R$ is singular, or nearly singular then F02WUF(*) can be used to determine the singular value decomposition of $R$.

9. Example

To obtain the QR factorization of the 5 by 3 matrix

\[
A = \begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

F01 -- Matrix Factorizations
F01QDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details.

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01QDF performs one of the transformations

\[ T \begin{cases} 
B := QB \\
B := Q B 
\end{cases} \]

where $B$ is an $m$ by $ncolb$ real matrix and $Q$ is an $m$ by $m$ orthogonal matrix, given as the product of Householder transformation matrices.

This routine is intended for use following F01QCF or F01QFF(*).

2. Specification

```fortran
SUBROUTINE F01QDF (TRANS, WHERET, M, N, A, LDA, ZETA, 1 NCOLB, B, LDB, WORK, IFAIL)
INTEGER M, N, LDA, NCOLB, LDB, IFAIL
DOUBLE PRECISION A(LDA,*), ZETA(*), B(LDB,*), WORK(*)
CHARACTER*1 TRANS, WHERET
```
3. Description

Q is assumed to be given by

\[ Q^T = Q_n Q_{n-1} \ldots Q_1, \]

where

\[ Q_k = \begin{pmatrix} I & 0 \\ 0 & T \end{pmatrix}, \]

and

\[ T_k = I - u_k u_k^T, \]

\[ u_k = (z_k), \]

\[ z_k \]

(zeta) is a scalar and z is an (m-k) element vector. z must be supplied in the kth column of A in elements A(k+1,k),...,A(m,k) and (zeta) must be supplied either in A(k,k) or in ZETA(k), depending upon the parameter WHERET.

To obtain Q explicitly B may be set to I and pre-multiplied by Q.

This is more efficient than obtaining Q.

4. References


5. Parameters

1: TRANS -- CHARACTER*1

   Input
On entry: the operation to be performed as follows:
TRANS = 'N' (No transpose)
   Perform the operation \( B := Q B \).
TRANS = 'T' or 'C' (Transpose)
   \( T \)
   Perform the operation \( B := Q^T B \).
Constraint: TRANS must be one of 'N', 'T' or 'C'.

2: WHERET -- CHARACTER*1 Input
On entry: indicates where the elements of (zeta) are to be
found as follows:
WHERET = 'I' (In A)
The elements of (zeta) are in A.
WHERET = 'S' (Separate)
The elements of (zeta) are separate from A, in ZETA.
Constraint: WHERET must be one of 'I' or 'S'.

3: M -- INTEGER Input
On entry: m, the number of rows of A. Constraint: M >= N.

4: N -- INTEGER Input
On entry: n, the number of columns of A.

When N = 0 then an immediate return is effected.
Constraint: N >= 0.

5: A(LDA,*) -- DOUBLE PRECISION array Input
Note: the second dimension of the array A must be at least
max(1,N).
On entry: the leading m by n strictly lower triangular part
of the array A must contain details of the matrix Q. In
addition, when WHERET = 'I', then the diagonal elements of A
must contain the elements of (zeta) as described under the
argument ZETA below.

When WHERET = 'S', the diagonal elements of the array A are
referenced, since they are used temporarily to store the
(zeta), but they contain their original values on return.

6: LDA -- INTEGER Input
On entry:
the first dimension of the array A as declared in the
(sub)program from which F01QDF is called.
Constraint: LDA >= max(1,M).

7: ZETA(*) -- DOUBLE PRECISION array Input
Note: when WHERET = 'S', the dimension of the array ZETA
must be greater than or equal to \(\max(1, N)\). On entry: if \(\text{WHERE} = 'S'\), the array \(ZETA\) must contain the elements of \((zeta)\). If \(ZETA(k) = 0.0\) then \(T\) is assumed to be \(I_k\), otherwise \(ZETA(k)\) is assumed to contain \((zeta)\).

When \(\text{WHERE} = 'I'\), \(ZETA\) is not referenced.

8: \(\text{NCOLB} -- \text{INTEGER}\) Input
On entry: \(\text{ncolb}\), number of columns of \(B\).
When \(\text{NCOLB} = 0\) then an immediate return is effected.
Constraint: \(\text{NCOLB} \geq 0\).

9: \(B(LDB,*) -- \text{DOUBLE PRECISION array}\) Input/Output
Note: the second dimension of the array \(B\) must be at least \(\max(1, \text{NCOLB})\).
On entry: the leading \(m\) by \(\text{ncolb}\) part of the array \(B\) must contain the matrix to be transformed. On exit: \(B\) is overwritten by the transformed matrix.

10: \(\text{LDB} -- \text{INTEGER}\) Input
On entry: the first dimension of the array \(B\) as declared in the (sub)program from which \(F01QDF\) is called.
Constraint: \(\text{LDB} \geq \max(1, M)\).

11: \(\text{WORK(*)} -- \text{DOUBLE PRECISION array}\) Workspace
Note: the dimension of the array \(\text{WORK}\) must be at least \(\max(1, \text{NCOLB})\).

12: \(\text{IFAIL} -- \text{INTEGER}\) Input/Output
On entry: \(\text{IFAIL}\) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: \(\text{IFAIL} = 0\) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \(\text{IFAIL} = 0\) or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

\(\text{IFAIL} = -1\)
On entry \(\text{TRANS} /= 'N', 'T'\) or 'C',
or \[ \text{WHERE}T \neq 'I' \text{ or } 'S', \]

or \[ M < N, \]

or \[ N < 0, \]

or \[ \text{LDA} < M, \]

or \[ \text{NCO} \text{LB} < 0, \]

or \[ \text{LDB} < M. \]

7. Accuracy

Letting \( C \) denote the computed matrix \( Q^T B \), \( C \) satisfies the relation

\[ QC = B + E, \]

where

\[ \|E\| \leq c(\text{epsilon})\|B\|, \]

and (epsilon) the machine precision (see \texttt{X02AJF(*)}), \( c \) is a modest function of \( m \) and \( \|\|.\|\| \) denotes the spectral (two) norm. An equivalent result holds for the computed matrix \( QB \). See also Section 7 of \texttt{F01QCF}.

8. Further Comments

The approximate number of floating-point operations is given by

\[ 2n(2m-n)\text{ncolb}. \]

9. Example

To obtain the matrix \( Q^T B \) for the matrix \( B \) given by

\[
\begin{pmatrix}
1.1 & 0.00 \\
0.9 & 0.00 \\
0.6 & 1.32 \\
0.0 & 1.10 \\
-0.8 & -0.26
\end{pmatrix}
\]

following the QR factorization of the 5 by 3 matrix \( A \) given by

\[
\begin{pmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8
\end{pmatrix}
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

F01 -- Matrix Factorizations

F01QEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01QEF returns the first ncolq columns of the real m by m orthogonal matrix Q, where Q is given as the product of Householder transformation matrices.

This routine is intended for use following F01QCF or F01QFF(*).

2. Specification

```fortran
SUBROUTINE F01QEF (WHERET, M, N, NCOLQ, A, LDA, ZETA,
                    WORK, IFAIL)
   INTEGER M, N, NCOLQ, LDA, IFAIL
   DOUBLE PRECISION A(LDA,*), ZETA(*), WORK(*)
   CHARACTER*1 WHERET
```

3. Description

Q is assumed to be given by

\[
Q^T = \begin{pmatrix} Q & \ldots & Q \end{pmatrix},
\]
\[
\begin{array}{cccc}
\end{array}
\]

Q being given in the form

\[
Q = \begin{pmatrix} I & 0 \\ 0 & T \\ k & k \end{pmatrix}
\]

where
\[ T = I - u u^T, \]
\[ k \quad k \quad k \]
\[ ((\text{zeta}) ) \]
\[ ( \quad k ) \]
\[ u = (z^T ), \]
\[ k \quad k \quad k \]

(zeta) is a scalar and z is an \((m-k)\) element vector. z must be supplied in the \(k\)th column of \(A\) in elements \(A(k+1,k),...,A(m,k)\) and (zeta) must be supplied either in \(A(k,k)\) or in \(ZETA(k)\), depending upon the parameter WHERET.

4. References


5. Parameters

1: WHERET -- CHARACTER*1 Input
On entry: indicates where the elements of (zeta) are to be found as follows:
\[ \text{WHERET} = 'I' \ (\text{In } A) \]
The elements of (zeta) are in A.
\[ \text{WHERET} = 'S' \ (\text{Separate}) \]
The elements of (zeta) are separate from A, in ZETA. Constraint: WHERET must be one of 'I' or 'S'.

2: M -- INTEGER Input
On entry: \(m\), the number of rows of A. Constraint: \(M \geq N\).

3: N -- INTEGER Input
On entry: \(n\), the number of columns of A. Constraint: \(N \geq 0\).

4: NCOLQ -- INTEGER Input
On entry: ncolq, the required number of columns of Q. Constraint: \(0 \leq \text{NCOLQ} \leq M\).

When NCOLQ = 0 then an immediate return is effected.
5: A(LDA,*) -- DOUBLE PRECISION array  Input/Output
Note: the second dimension of the array A must be at least
max(1,N,NCOLQ).
On entry: the leading m by n strictly lower triangular part
of the array A must contain details of the matrix Q. In
addition, when WHERET = 'I', then the diagonal elements of A
must contain the elements of (zeta) as described under the
argument ZETA below. On exit: the first NCOLQ columns of the
array A are overwritten by the first NCOLQ columns of the m
by m orthogonal matrix Q. When N = 0 then the first NCOLQ
columns of A are overwritten by the first NCOLQ columns of
the identity matrix.

6: LDA -- INTEGER  Input
On entry:
the first dimension of the array A as declared in the
(sub)program from which F01QEF is called.
Constraint: LDA >= max(1,M).

7: ZETA(*) -- DOUBLE PRECISION array  Input
Note: the dimension of the array ZETA must be at least
max(1,N).
On entry: with WHERET = 'S', the array ZETA must contain the
elements of (zeta). If ZETA(k) = 0.0 then T is assumed to
be I, otherwise ZETA(k) is assumed to contain (zeta).

When WHERET = 'I', the array ZETA is not referenced.

8: WORK(*) -- DOUBLE PRECISION array  Workspace
Note: the dimension of the array WORK must be at least
max(1,NCOLQ).

9: IFAIL -- INTEGER  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL=-1
On entry WHERE T /= 'I' or 'S',
or M < N,
or N < 0,
or NCOLQ < 0 or NCOLQ > M,
or LDA < M.

7. Accuracy

The computed matrix Q satisfies the relation

\[ Q = P + E, \]

where P is an exactly orthogonal matrix and

\[ \|E\| \leq c(\epsilon) \]

(\epsilon) is the machine precision (see X02AJF()), c is a modest function of m and \( \|\| \) denotes the spectral (two) norm. See also Section 7 of F01QCF.

8. Further Comments

The approximate number of floating-point operations required is
given by

\[ 2 \cdot n \cdot ((3m-n)(2ncolq-n)-n(ncolq-n)), \quad ncolq > n, \]
\[ 3 \]
\[ 2 \]
\[ -ncolq (3m-ncolq), \quad ncolq <= n. \]
\[ 3 \]

9. Example

To obtain the 5 by 5 orthogonal matrix Q following the QR factorization of the 5 by 3 matrix A given by

\[
\begin{bmatrix}
2.0 & 2.5 & 2.5 \\
2.0 & 2.5 & 2.5 \\
1.6 & -0.4 & 2.8 \\
2.0 & -0.5 & 0.5 \\
1.2 & -0.3 & -2.9
\end{bmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation.
Library software and should be available on-line.

F01 -- Matrix Factorizations
F01RCF
F01RCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

F01RCF finds the QR factorization of the complex m by n matrix A, where m>=n.

2. Specification

SUBROUTINE F01RCF (M, N, A, LDA, THETA, IFAIL)
INTEGER M, N, LDA, IFAIL
COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*)

3. Description

The m by n matrix A is factorized as

\[
\begin{align*}
R \\
A &= Q(0) \quad \text{when } m>n, \\
&= QR \quad \text{when } m=n,
\end{align*}
\]

where Q is an m by m unitary matrix and R is an n by n upper triangular matrix with real diagonal elements.

The factorization is obtained by Householder’s method. The kth transformation matrix, Q, which is used to introduce zeros into the kth column of A is given in the form

\[
\begin{align*}
&= (I \ 0) \\
Q &= (0 \ T), \\
&= (1 \ 0) \\
T &= I-(\gamma) u u,
\end{align*}
\]

where

\[
T = I-(\gamma) u u,
\]

\[
\begin{align*}
&= (1 \ 0) \\
&= (1 \ 0)
\end{align*}
\]
((zeta) )
( k)
u =\( (z_{k}) \),
{k ( k )

(gamma) is a scalar for which Re (gamma) =1.0, (zeta) is a real
k k k scalar and z is an (m-k) element vector. (gamma), (zeta) and
k k k z are chosen to annihilate the elements below the triangular
k part of A and to make the diagonal elements real.

The scalar (gamma) and the vector u are returned in the kth
k k k element of the array THETA and in the kth column of A, such that
k

(\theta) =((zeta), \text{Im}(gamma)),
k k k

is in THETA(k) and the elements of z are in a \ldots,a . The
k k+1,k m,k elements of R are returned in the upper triangular part of A.

Q is given by

\[ Q = (Q_{Q \ldots Q} ) . \]
\[ n \quad n-1 \quad 1 \]

A good background description to the QR factorization is given in
Dongarra et al [1], but note that this routine is not based upon
LINPACK routine ZQRDC.

4. References

LINPACK Users' Guide. SIAM, Philadelphia.

Oxford University Press.

5. Parameters

1: M -- INTEGER
   Input
   On entry: m, the number of rows of A. Constraint: M >= N.

2: N -- INTEGER
   Input
On entry: n, the number of columns of A. Constraint: N >= 0.

When N = 0 then an immediate return is effected.

3: A(LDA,*) -- COMPLEX(KIND(1.0D0)) array Input/Output
   Note: the second dimension of the array A must be at least max(1,N).
   On entry: the leading m by n part of the array A must contain the matrix to be factorized. On exit: the n by n upper triangular part of A will contain the upper triangular matrix R, with the imaginary parts of the diagonal elements set to zero, and the m by n strictly lower triangular part of A will contain details of the factorization as described above.

4: LDA -- INTEGER Input
   On entry: the first dimension of the array A as declared in the (sub)program from which F01RCF is called.
   Constraint: LDA >= max(1,M).

5: THETA(*) -- COMPLEX(KIND(1.0D)) array Output
   Note: the dimension of the array THETA must be at least max(1,N).
   On exit: the scalar (theta) for the kth transformation. If T =I then THETA(k) = 0.0; if k
           (\alpha) 0
           T = ( 0 1)  Re(\alpha)<0.0,
           k
   then THETA(k)=(\alpha); otherwise THETA(k) contains THETA(k) as described in Section 3 and Re(THETA(k)) is always in the range (1.0, \sqrt{2.0}).

6: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL=-1
   On entry M < N,
   or N < 0,
   or LDA < M.

7. Accuracy
The computed factors Q and R satisfy the relation

\[
Q(0) = A + E,
\]

where

\[ ||E|| \leq c(\epsilon) ||A||, \]

(\epsilon) being the machine precision, c is a modest function of m and n and ||.|| denotes the spectral (two) norm.

8. Further Comments
The approximate number of real floating-point operations is given by 8n (3m-n)/3.

Following the use of this routine the operations

\[ H \]
\[ B := QB \]
\[ B := QB \]

where B is an m by k matrix, can be performed by calls to F01RDF. The operation B := QB can be obtained by the call:

```
IFAIL = 0
CALL F01RDF('No conjugate', 'Separate', M, N, A, LDA, THETA,
*            K, B, LDB, WORK, IFAIL)
```

and B := QB can be obtained by the call:

```
IFAIL = 0
CALL F01RDF('Conjugate', 'Separate', M, N, A, LDA, THETA,
*            K, B, LDB, WORK, IFAIL)
```
In both cases WORK must be a k element array that is used as workspace. If B is a one-dimensional array (single column) then the parameter LDB can be replaced by M. See F01RDF for further details.

The first k columns of the unitary matrix Q can either be obtained by setting B to the first k columns of the unit matrix and using the first of the above two calls, or by calling F01REF, which overwrites the k columns of Q on the first k columns of the array A. Q is obtained by the call:

\[
\text{CALL F01REF('Separate', M, N, K, A, LDA, THETA, WORK, IFAIL)}
\]

As above, WORK must be a k element array. If k is larger than n, then A must have been declared to have at least k columns.

Operations involving the matrix R can readily be performed by the Level 2 BLAS routines ZTRSV and ZTRMV (see Chapter F06), but note that no test for near singularity of R is incorporated in ZTRSV. If R is singular, or nearly singular, then F02XUF(*) can be used to determine the singular value decomposition of R.

9. Example

To obtain the QR factorization of the 5 by 3 matrix

\[
\begin{pmatrix}
  0.5i & -0.5+1.5i & -1.0+1.0i \\
  0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
  0.4 & -0.4+0.4i & 1.8 \\
  0.3-0.4i & 0.1+0.7i & 0.0 \\
  -0.3i & 0.3+0.3i & 2.4i \\
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
where $B$ is an $m$ by $n_{colb}$ complex matrix and $Q$ is an $m$ by $m$
unitary matrix, given as the product of Householder
transformation matrices.

This routine is intended for use following F01RCF or F01RFF(*).

2. Specification

```
SUBROUTINE F01RDF (TRANS, WHERET, M, N, A, LDA, THETA,
1                  NCOLB, B, LDB, WORK, IFAIL)
        INTEGER M, N, LDA, NCOLB, LDB, IFAIL
        COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*), B(LDB,*), WORK(*)
        CHARACTER*1 TRANS, WHERET
```

3. Description

The unitary matrix $Q$ is assumed to be given by

$$
Q = (Q_{n-1} \cdots Q_{1}) ,
$$

$Q$ being given in the form

$$
\begin{bmatrix}
(I & 0 \\
0 & T ,
\end{bmatrix},
$$

where

$$
T = I - (\gamma) u u ,
$$

$$
((\zeta) )
\begin{bmatrix}
( & k \\
z & k
\end{bmatrix},
$$

$(\gamma)$ is a scalar for which $\text{Re} (\gamma) = 1.0$, $(\zeta)$ is a real
$k$ scalar and $z$ is an $(m-k)$ element vector.

$z$ must be supplied in the $k$th column of $A$ in elements
CHAPTER 15. CHAPTER N

k
a ,...,a  and (theta) , given by
k+1,k m,k k

(\theta) = (\zeta) , Im (\gamma) ,
k k k

must be supplied either in a  or in THETA(k), depending upon
k,k
the parameter WHERE.T.

To obtain Q explicitly B may be set to I and pre-multiplied by Q.

This is more efficient than obtaining Q. Alternatively, F01REF
may be used to obtain Q overwritten on A.

4. References

    Oxford University Press.

5. Parameters

1: TRANS -- CHARACTER*1 Input
   On entry: the operation to be performed as follows:
   TRANS = 'N' (No transpose)
       Perform the operation B := QB.

   TRANS = 'C' (Conjugate transpose)
       H
       Perform the operation B := Q^H B.
   Constraint: TRANS must be one of 'N' or 'C'.

2: WHERE.T -- CHARACTER*1 Input
   On entry: the elements of (theta) are to be found as follows:
   WHERE.T = 'I' (In A)
       The elements of (theta) are in A.

   WHERE.T = 'S' (Separate)
       The elements of (theta) are separate from A, in THETA.
   Constraint: WHERE.T must be one of 'I' or 'S'.

3: M -- INTEGER Input
   On entry: m, the number of rows of A. Constraint: M >= N.

4: N -- INTEGER Input
   On entry: n, the number of columns of A. Constraint: N >= 0.
When \( N = 0 \) then an immediate return is effected.

5: \( A(LDA,*) \) -- COMPLEX(KIND(1.0D)) array Input
   Note: the second dimension of the array \( A \) must be at least \( \max(1,N) \).
   On entry: the leading \( m \) by \( n \) strictly lower triangular part of the array \( A \) must contain details of the matrix \( Q \). In addition, when \( \text{WERET} = 'I' \), then the diagonal elements of \( A \) must contain the elements of \((\theta)\) as described under the argument \( \text{THETA} \) below.

   When \( \text{WERET} = 'S' \), then the diagonal elements of the array \( A \) are referenced, since they are used temporarily to store the \((zeta)\), but they contain their original values on return.

6: \( LDA \) -- INTEGER Input
   On entry: the first dimension of the array \( A \) as declared in the (sub)program from which \( \text{F01RDF} \) is called.
   Constraint: \( LDA \geq \max(1,M) \).

7: \( \text{THETA} (*) \) -- COMPLEX(KIND(1.0D)) array Input
   Note: the dimension of the array \( \text{THETA} \) must be at least \( \max(1,N) \).
   On entry: with \( \text{WERET} = 'S' \), the array \( \text{THETA} \) must contain the elements of \((\theta)\). If \( \text{THETA}(k)=0.0 \) then \( T \) is assumed to be \( I \); if \( \text{THETA}(k)=(\alpha), \text{with} \ Re(\alpha)<0.0 \), then \( T \) is assumed to be of the form
   \[
   \begin{pmatrix}
   (\alpha) & 0 \\
   0 & I
   \end{pmatrix}
   \]
   otherwise \( \text{THETA}(k) \) is assumed to contain \((\theta)\) given by
   \[
   (\theta) = ((zeta), Im(\gamma)) \]

   When \( \text{WERET} = 'I' \), the array \( \text{THETA} \) is not referenced, and may be dimensioned of length 1.

8: \( \text{NCOLB} \) -- INTEGER Input
   On entry: \( ncolb \), the number of columns of \( B \). Constraint:
   \( \text{NCOLB} \geq 0 \).
   When \( \text{NCOLB} = 0 \) then an immediate return is effected.

9: \( B(LDB,*) \) -- COMPLEX(KIND(1.0D)) array Input/Output
   Note: the second dimension of the array \( B \) must be at least \( \max(1,\text{NCOLB}) \).
On entry: the leading \( m \) by \( \text{ncolb} \) part of the array \( B \) must contain the matrix to be transformed. On exit: \( B \) is overwritten by the transformed matrix.

10: \( \text{LDB} \) -- INTEGER Input
On entry:
the first dimension of the array \( B \) as declared in the (sub)program from which \text{F01RDF} is called.
Constraint: \( \text{LDB} \geq \max(1,\text{M}) \).

11: \( \text{WORK}(*) \) -- COMPLEX(KIND(1.0D)) array Workspace
Note: the dimension of the array \( \text{WORK} \) must be at least \( \max(1,\text{NCOLB}) \).

12: \( \text{IFAIL} \) -- INTEGER Input/Output
On entry: \( \text{IFAIL} \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: \( \text{IFAIL} = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \( \text{IFAIL} = 0 \) or -1, explanatory error messages are output on the current error message unit (as defined by \text{X04AAF}).

\( \text{IFAIL} = -1 \)
On entry \( \text{TRANS} /= 'N' \) or 'C',
or \( \text{WHERE} /= 'I' \) or 'S',
or \( M < N \),
or \( N < 0 \),
or \( \text{LDA} < M \),
or \( \text{NCOLB} < 0 \),
or \( \text{LDB} < M \).

7. Accuracy

Letting \( C \) denote the computed matrix \( Q \times B \), \( C \) satisfies the relation

\[ QC = B + E, \]
where

\[ ||E|| \leq c(\varepsilon)||B||, \]

(\varepsilon) being the machine precision, \( c \) is a modest function of \( m \) and \( ||.|| \) denotes the spectral (two) norm. An equivalent result holds for the computed matrix QB. See also Section 7 of F01RCF.

8. Further Comments

The approximate number of real floating-point operations is given by \( 8n(2m-n)\text{ncolb} \).

9. Example

To obtain the matrix QB for the matrix B given by

\[
\begin{pmatrix}
-0.55+1.05i & 0.45+1.05i \\
0.49+0.93i & 1.09+0.13i \\
0.56-0.16i & 0.64+0.16i \\
0.39+0.23i & -0.39-0.23i \\
1.13+0.83i & -1.13+0.77i
\end{pmatrix}
\]

following the QR factorization of the 5 by 3 matrix A given by

\[
\begin{pmatrix}
0.5 & -0.5+1.5i & -1.0+1.0i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.4-0.4i & 1.8 & . \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
F01REF returns the first ncolq columns of the complex m by m unitary matrix Q, where Q is given as the product of Householder transformation matrices.

This routine is intended for use following F01RCF.

2. Specification

```plaintext
SUBROUTINE F01REF (WHERE, M, N, NCOLQ, A, LDA, THETA,
                    WORK, IFAIL)
INTEGER M, N, NCOLQ, LDA, IFAIL
COMPLEX(KIND(1.0D0)) A(LDA,*), THETA(*), WORK(*)
CHARACTER*1 WHERE
```

3. Description

The unitary matrix Q is assumed to be given by

\[
Q = (Q_{n} \ldots Q_{1}) ,
\]

where

\[
Q_{k} = (I_{0} 0) ,
\]

\[
Q_{k} = (0 T),
\]

\[
T = I - (\gamma) u u, \quad k \quad k \quad k \quad k
\]

\[
((\zeta))
\]

\[
(0 \quad k)
\]

\[
(z) ,
\]

\[
(k \quad k)
\]

(\gamma) is a scalar for which Re (\gamma) = 1.0, (\zeta) is a real scalar and z is an (m-k) element vector.

z must be supplied in the kth column of A in elements

\[
a_{k+1,k} \ldots a_{m,k}
\]

and (\theta), given by

\[
\theta_{k+1,k} \ldots \theta_{m,k}
\]
\[ (\theta) = ((\zeta), \text{Im} (\gamma)) \]

must be supplied either in \( a \) or in \( \text{THETA}(k) \) depending upon the parameter \( \text{WERET} \).

4. References


5. Parameters

1: \( \text{WERET} -- \text{CHARACTER} \cdot 1 \) Input
   On entry: the elements of \( (\theta) \) are to be found as follows:
   \( \text{WERET} = 'I' \) (In \( A \))
   The elements of \( (\theta) \) are in \( A \).
   \( \text{WERET} = 'S' \) (Separate)
   The elements of \( (\theta) \) are separate from \( A \), in \( \text{THETA} \).
   Constraint: \( \text{WERET} \) must be one of 'I' or 'S'.

2: \( M -- \text{INTEGER} \) Input
   On entry: \( m \), the number of rows of \( A \). Constraint: \( M \geq N \).

3: \( N -- \text{INTEGER} \) Input
   On entry: \( n \), the number of columns of \( A \). Constraint: \( N \geq 0 \).

4: \( \text{NCOLQ} -- \text{INTEGER} \) Input
   On entry: \( \text{ncolq} \), the required number of columns of \( Q \).
   Constraint: \( 0 \leq \text{NCOLQ} \leq M \).
   When \( \text{NCOLQ} = 0 \) then an immediate return is effected.

5: \( A(LDA,*) -- \text{COMPLEX} \cdot (\text{KIND}(1.0D)) \) array Input/Output
   Note: the second dimension of the array \( A \) must be at least \( \max(1,N,\text{NCOLQ}) \).
   On entry: the leading \( m \) by \( n \) strictly lower triangular part of the array \( A \) must contain details of the matrix \( Q \). In addition, when \( \text{WERET} = 'I' \), then the diagonal elements of \( A \) must contain the elements of \( (\theta) \) as described under the argument \( \text{THETA} \) below.
   On exit: the first \( \text{NCOLQ} \) columns of the array \( A \) are overwritten by the first \( \text{NCOLQ} \) columns of the \( m \) by \( m \) unitary matrix \( Q \). When \( N = 0 \) then the first \( \text{NCOLQ} \) columns of \( A \) are overwritten by the first \( \text{NCOLQ} \) columns of the unit matrix.
6: LDA -- INTEGER  
   On entry:  
   the first dimension of the array A as declared in the  
   (sub)program from which F01REF is called.  
   Constraint: LDA >= max(1,M).

7: THETA(*) -- COMPLEX(KIND(1.0D)) array  
   Input  
   Note: the dimension of the array THETA must be at least  
   max(1,N).  
   On entry: if WHERET = 'S', the array THETA must contain the  
   elements of (theta). If THETA(k)=0.0 then T is assumed to  
   be I; if THETA(k)=(alpha), with Re(alpha)<0.0, then T is  
   assumed to be of the form  
   \[(\alpha 0)\]  
   \[T = ( \begin{array}{cc} 0 & I \\ \end{array} )\]  
   \[k\]  
   otherwise THETA(k) is assumed to contain (theta) given by  
   \[(\theta) = ((\zeta), \text{Im}(\gamma)) \]  
   \[k \quad k \quad k\]  
   When WHERET = 'I', the array THETA is not referenced.

8: WORK(*) -- COMPLEX(KIND(1.0D)) array  
   Workspace  
   Note: the dimension of the array WORK must be at least  
   max(1,NCOLQ).

9: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are  
output on the current error message unit (as defined by X04AAF).

IFAIL=-1  
On entry WHERET /= 'I' or 'S',  
or \[M < N,\]
or \( N < 0 \),

or \( \text{NCOLQ} < 0 \) or \( \text{NCOLQ} > M \),

or \( \text{LDA} < M \).

7. Accuracy

The computed matrix \( Q \) satisfies the relation

\[ Q = P + E, \]

where \( P \) is an exactly unitary matrix and

\[ ||E|| <= c(\epsilon), \]

\( \epsilon \) being the machine precision, \( c \) is a modest function of \( m \) and \( ||||.||| \) denotes the spectral (two) norm. See also Section 7 of F01RCF.

8. Further Comments

The approximate number of real floating-point operations required is given by

\[
8 - n \{(3m-n)(2\text{ncolq}-n)-n(\text{ncolq}-n)\}, \text{ncolq}>n \\
3 \]

\[
8 - 2 \text{ncolq} (3m-\text{ncolq}), \text{ncolq}<=n \\
3 \]

9. Example

To obtain the 5 by 5 unitary matrix \( Q \) following the QR factorization of the 5 by 3 matrix \( A \) given by

\[
A = \begin{pmatrix}
0.5i & -0.5+1.5i & -1.0+1.4i \\
0.4+0.3i & 0.9+1.3i & 0.2+1.4i \\
0.4 & -0.4+0.4i & 1.8 \\
0.3-0.4i & 0.1+0.7i & 0.0 \\
-0.3i & 0.3+0.3i & 2.4i
\end{pmatrix}
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagMatrixOperationsPackage (NAGF01)

Exports:
f01brf  f01bsf  f01maf  f01mcf  f01qcf
f01qdf  f01qef  f01rcf  f01 rdf  f01ref

— package NAGF01 NagMatrixOperationsPackage —

)abbrev package NAGF01 NagMatrixOperationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:15 1994
++ Description:
++ This package uses the NAG Library to provide facilities for matrix
++ factorizations and associated transformations.

NagMatrixOperationsPackage():Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
f01brf : (Integer,Integer,Integer,Integer,_,DoubleFloat,Boolean,Boolean,List Boolean,Matrix DoubleFloat,_,Matrix Integer,Integer) -> Result
++ f01brf(n,nz,licn,lirn,pivot,lblock,grow,abort,a,irn,icn,ifail)
++ factorizes a real sparse matrix. The routine either forms
++ the LU factorization of a permutation of the entire matrix, or,
++ optionally, first permutes the matrix to block lower triangular
++ form and then only factorizes the diagonal blocks.
++ See \downlink{Manual Page}{manpageXXf01brf}.

f01bsf : (Integer,Integer,Integer,Matrix Integer,_,Matrix Integer,Matrix Integer,Matrix Integer,Boolean,DoubleFloat,_,Boolean,Matrix Integer,Matrix DoubleFloat,Integer) -> Result
++ f01bsf(n,nz,licn,ivect,jvect,icn,ikeep,grow,
++ eta,abort,idisp,avals,ifail)
++ factorizes a real sparse matrix using the pivotal sequence
++ previously obtained by F01BRF when a matrix of the same sparsity
++ pattern was factorized.
++ See downlink{Manual Page}{manpageXXf01bsf}.
f01maf : (Integer,Integer,Integer,Integer,Integer,Boolean,Matrix DoubleFloat,Matrix Integer,Matrix Integer,DoubleFloat,DoubleFloat,Integer) -> Result
++ f01maf(n,nz,licn,lirn,abort,avals,irn,icn,droptl,densw,ifail)
++ computes an incomplete Cholesky factorization of a real
++ sparse symmetric positive-definite matrix A.
++ See downlink{Manual Page}{manpageXXf01maf}.
f01mcf : (Integer,Matrix DoubleFloat,Integer,Integer) -> Result
++ f01mcf(n,avals,lal,nrow,ifail)
++ computes the Cholesky factorization of a real symmetric
++ positive-definite variable-bandwidth matrix.
++ See downlink{Manual Page}{manpageXXf01mcf}.
f01qcf : (Integer,Integer,Integer,Matrix DoubleFloat,Integer) -> Result
++ f01qcf(m,n,lda,a,ifail)
++ finds the QR factorization of the real m by n matrix A,
++ where m>=n.
++ See downlink{Manual Page}{manpageXXf01qcf}.
f01qdf : (String,String,Integer,Integer,Matrix DoubleFloat,Integer,Integer,Integer,Matrix DoubleFloat,Integer) -> Result
++ f01qdf(trans,wheret,m,n,a,lda,zeta,ncolb,ldb,b,ifail)
++ performs one of the transformations
++ See downlink{Manual Page}{manpageXXf01qdf}.
f01qef : (String,Integer,Integer,Integer,Integer,Integer,Matrix DoubleFloat,Integer) -> Result
++ f01qef(wheret,m,n,ncolq,lda,zeta,a,ifail)
++ returns the first ncolq columns of the real m by m
++ orthogonal matrix Q, where Q is given as the product of
++ Householder transformation matrices.
++ See downlink{Manual Page}{manpageXXf01qef}.
f01rcf : (Integer,Integer,Integer,Integer,Matrix Complex DoubleFloat,Integer) -> Result
++ f01rcf(m,n,lda,a,ifail)
++ finds the QR factorization of the complex m by n matrix A,
++ where m>=n.
++ See downlink{Manual Page}{manpageXXf01rcf}.
f01rdf : (String,String,Integer,Integer,Matrix Complex DoubleFloat,Integer,Integer,Integer,Matrix Complex DoubleFloat,Integer) -> Result
++ f01rdf(trans,wheret,m,n,a,lda,theta,ncolb,ldb,b,ifail)
++ performs one of the transformations
++ See downlink{Manual Page}{manpageXXf01rdf}.
f01ref : (String,Integer,Integer,Integer,Integer) -> Result
++ f01ref(trans,wheret,m,n,a,lda,theta,ncolb,ldb,b,ifail)
++ performs one of the transformations
++ See downlink{Manual Page}{manpageXXf01ref}.
Integer, Matrix Complex DoubleFloat, Matrix Complex DoubleFloat, Integer) -> Result

++ \[f01ref(\text{where}\text{t}, m, n, ncolq, lda, theta, a, ifail)}\]
++ returns the first ncolq columns of the complex m by m
++ unitary matrix Q, where Q is given as the product of Householder
++ transformation matrices.
++ See \[\text{downlink}\{Manual Page\}\{manpageXXf01ref\}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Boolean)
import AnyFunctions1(String)
import AnyFunctions1(List Boolean)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Complex DoubleFloat)
import AnyFunctions1(Matrix Integer)

f01brf(nArg:Integer, nzArg:Integer, licnArg:Integer, 
        lirnArg:Integer, pivotArg:DoubleFloat, lblockArg:Boolean, 
        growArg:Boolean, abortArg:List Boolean, aArg:Matrix DoubleFloat, 
        irnArg:Matrix Integer, icnArg:Matrix Integer, 
        ifailArg:Integer): Result == 
    [(invokeNagman(NIL$Lisp, 
        "f01brf", 
        "w"::S,"idisp"::S,"a"::S, 
        "i"::S,"icn"::S,"iw"::S]$Lisp, 
    ["ikeep"::S,"w"::S,"idisp"::S,"iw"::S]$S$Lisp, 
    ["double"::S,"pivot"::S, ["w"::S,"n"::S]$S$Lisp, 
    ["a"::S,"licn"::S]$S$Lisp]$Lisp, 
    ["ikeep"::S, ["]::S,"$"::S,"n"::S]$S$Lisp]$Lisp, 
    ["idisp"::S, 10$Lisp]$S$Lisp, ["irn"::S,"lirn"::S]$S$Lisp, 
    ["icn"::S,"licn"::S]$S$Lisp, 
    ["logical"::S,"lblock"::S, ["double"::S, 4$Lisp]$S$Lisp]$Lisp]$Lisp, 
    ["ikeep"::S, ["w"::S,"idisp"::S, ["a"::S,"irn"::S, 
    ["icn"::S,"ifail"::S]$S$Lisp,$Lisp,}]}
PACKAGE NAGF01 NAGMATRIXOPERATIONSPACKAGE

f01bsf(nArg:Integer,nzArg:Integer,licnArg:Integer,_
ivectArg:Matrix Integer,jvectArg:Matrix Integer,icnArg:Matrix Integer,_
ikeepArg:Matrix Integer,growArg:Boolean,etaArg:DoubleFloat,_
abortArg:Boolean,idispArg:Matrix Integer,avalsArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"f01bsf",_,
"icn":S,"ikeep":S,"idisp":S_,
,"w":S,"avals":S,"iw":S]$Lisp_,
["w":S,"rpmin":S,"iw":S]$Lisp_,
[["double":S,"eta":S,"w":S,"n":S]$Lisp_,
,"rpmin":S,[]$Lisp],
[[idisp":S,2$Lisp]$Lisp],
["logical":S,"grow":S,"abort":S]$Lisp_]
]$Lisp_]

f01maf(nArg:Integer,nzArg:Integer,licnArg:Integer,_
lirnArg:Integer,abortArg:List Boolean,avalsArg:Matrix DoubleFloat,_
irnArg:Matrix Integer,icnArg:Matrix Integer,droptlArg:DoubleFloat,_
denswArg:DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"f01maf",_,
"inform":S,"avals":S_,
,"irn":S,"icn":S,"iwork":S]$Lisp_,
[["double":S,"wkeep":S,"*":S,3$Lisp,"n":S]$Lisp]$Lisp_,
,[["ikeep":S,"*":S,2$Lisp,"n":S]$Lisp]$Lisp_],
[["inform":S,4$Lisp]$Lisp],
["irn":S,"lirn":S]$Lisp_,
["inform":S,4$Lisp]$Lisp],
["irn":S,"lirn":S]$Lisp_,
["inform":S,4$Lisp]$Lisp],
["irn":S,"lirn":S]$Lisp_)]

pretend List (Record(key:Symbol,entry:Any))$Result
CHAPTER 15. CHAPTER N

["icn"::S,"licn"::S]$Lisp_,
,"logical"::S,"*"::S,6$Lisp,]$Lisp]
,"icn"::S,"droptl"::S,"densw"::S,"ifail"::S]$Lisp,]_n,
[([nArg::Any,nzArg::Any,licnArg::Any,lirnArg::Any,droptlArg::Any,]_n
densArg::Any,ifailArg::Any,abortArg::Any,avalsArg::Any,]_n
delmArg::Any,icnArg::Any])]
"List Any]$Lisp)$Lisp)_$Result

f01mcf(nArg:Integer,avalsArg:Matrix DoubleFloat,lalArg:Integer,]_n
nrowArg:Matrix Integer,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,]_n
"f01mcf",]_n
["al"::S,"d"::S]$Lisp,]_n
,"d"::S,"n"::S]$Lisp]$Lisp]$Lisp,]_n
,"ifail"::S]$Lisp,]_n
"al"::S,"d"::S,"ifail"::S]$Lisp,]_n
[([nArg::Any,lalArg::Any,ifailArg::Any,avalsArg::Any,nrowArg::Any])]
"List Any]$Lisp)$Lisp)_$Result

f01qcf(mArg:Integer,nArg:Integer,ldaArg:Integer,]_n
aArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,]_n
"f01qcf",]_n
["zeta"::S]$Lisp,]_n
]_n
]_n
"List "$Lisp]_n
["zeta"::S,"a"::S,"ifail"::S]$Lisp,]_n
[([mArg::Any,nArg::Any,ldaArg::Any,ifailArg::Any,aArg::Any])]
"List Any]$Lisp)$Lisp)_$Result

f01qdf(transArg:String,wheretArg:String,mArg:Integer,]_n
nArg:Integer,aArg:Matrix DoubleFloat,ldaArg:Integer,]_n
delmArg::Any,avalsArg::Any,nrowArg::Any])]
"List Any]$Lisp)$Lisp)_$Result
f01qef(wheretArg: String, mArg: Integer, nArg: Integer, 
ncolqArg: Integer, ldaArg: Integer, zetaArg: Matrix DoubleFloat, 
aArg: Matrix DoubleFloat, ifailArg: Integer): Result == 
[(invokeNagman(NIL$Lisp, _
"f01qef",_ 
"work"::S]$Lisp, _
,"ldb"::S,"ifail"::S]$Lisp_ 
["b"::S,"ifail"::S]$Lisp, _
[(transArg::Any,wheretArg::Any,mArg::Any,nArg::Any,ldaArg::Any, _
 ncolbArg::Any,ldbArg::Any,ifailArg::Any,aArg::Any, _
 zetaArg::Any,bArg::Any ])_ 
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

f01rcf(mArg: Integer, nArg: Integer, ldaArg: Integer, 
aArg: Matrix Complex DoubleFloat, ifailArg: Integer): Result == 
[(invokeNagman(NIL$Lisp, _
"f01rcf",_ 
["theta"::S]$Lisp, _
,["double complex"::S,["theta"::S,"n"::S]$Lisp, _
["theta"::S,"a"::S,"ifail"::S]$Lisp, _
[(mArg::Any,nArg::Any,ldaArg::Any,ifailArg::Any,aArg::Any ])_ 
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
((invokeNagman(NIL$Lisp, "f01rdf"),
  ["character"::S,"trans"::S,"wheret"::S]]$Lisp,
  [["double complex"::S,"a"::S,"lda"::S,"n"::S]]$Lisp,
  [["work"::S,"ncolb"::S]]$Lisp,$Lisp,
  [["b"::S,"ifail"::S]]$Lisp,
  [[transArg::Any, wheretArg::Any, mArg::Any, nArg::Any, ldaArg::Any, ncolbArg::Any, ldbArg::Any, ifailArg::Any, aArg::Any, thetaArg::Any, bArg::Any ]]@List Any$Lisp)
pretend List (Record(key:Symbol, entry:Any))$Result

((invokeNagman(NIL$Lisp, "f01ref"),
  ["work"::S]$Lisp,
  [["integer"::S,"m"::S,"n"::S,"ncolq"::S,"lda"::S, "ifail"::S]]$Lisp,
  ["character"::S,"wheret"::S]]$Lisp,
  [["double complex"::S,"theta"::S,"m"::S]]$Lisp,$Lisp,
  ["a"::S,"lda"::S,"n"::S]]$Lisp,$Lisp,$Lisp,$Lisp,
  [["a"::S,"ifail"::S]]$Lisp,$Lisp,
  [[wheretArg::Any, mArg::Any, nArg::Any, ncolqArg::Any, ldaArg::Any, ifailArg::Any, thetaArg::Any, aArg::Any ]]@List Any$Lisp)
pretend List (Record(key:Symbol, entry:Any))$Result

— NAGF01.dotabb —
package NAGE04 NagOptimisationPackage

--- NagOptimisationPackage.input ---

)set break resume
/sys rm -f NagOptimisationPackage.output
/spool NagOptimisationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 153
)show NagOptimisationPackage
--R
--R NagOptimisationPackage is a package constructor
--R Abbreviation for NagOptimisationPackage is NAGE04
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGE04
--R
--R------------------------------- Operations --------------------------------
--R e04dgf : (Integer,DoubleFloat,DoubleFloat,Integer,DoubleFloat,Boolean,DoubleFloat,DoubleFloat,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp49(OBJFUN))) -> Result
--R e04fdf : (Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp50(LSFUN))) -> Result
--R e04gcf : (Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp19(LSFUN))) -> Result
--R e04jaf : (Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp24(FUNCT1))) -> Result
--R e04mbf : (Integer,Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Boolean,Integer,Integer,Matrix(DoubleFloat),Integer,Union(fn: FileName,fp: Asp20(QPHESS))) -> Result
--R e04ucf : (Integer,Integer,Integer,Integer,Integer,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Union(fn: FileName,fp: Asp55(CONFUN)),Union(fn: FileName,fp: Asp49(OBJFUN))) -> Result

--E 1

--S 2 of 153
digits 20
--R
--R (1) 20
--R
--E 2

--S 3 of 153
showArrayValues true
--R
showScalarValues true

n:=2

es:=1.0

fu:=0.4373903597E-14

it:=50

lin:=0.9
--R
--E 9

--S 10 of 153
list:=true
--R
--R
--R (9) true
--R

--E 10

--S 11 of 153
ma:=1.0E+20
--R
--R
--R (10) 100 0000000000 00000000.0
--R

--E 11

--S 12 of 153
op:=3.26E-12
--R
--R
--R (11) 0.326 E -11
--R

--E 12

--S 13 of 153
pr:=10
--R
--R
--R (12) 10
--R

--E 13

--S 14 of 153
sta:=1
--R
--R
--R (13) 1
--R

--E 14

--S 15 of 153
sto:=2
--R
--R
--R (14) 2
--R

--E 15
ve:=0

x:Matrix SF:= [[-1.0, 1.0]]


objfun:ASP49(OBJFUN):= retract ef

result:=e04dgf(n,es,fu,it,lin,list,ma,op,sta,sto,ve,x,-1,objfun)

)clear all

showArrayValues true

true
showScalarValues true

m := 15

n := 3

liw := 1

lw := 171

x: Matrix SF := [[0.5, 1.0, 1.5]]
vef: Vector Expression Float := vector

\[
\begin{align*}
((XC[3]+15\cdot XC[2])^{(-1)}+XC[1]-0.14 ,
2\cdot(2\cdot XC[3]+14\cdot XC[2])^{(-1)}+XC[1]-0.18 ,
3\cdot(3\cdot XC[3]+13\cdot XC[2])^{(-1)}+XC[1]-0.22 ,
4\cdot(4\cdot XC[3]+12\cdot XC[2])^{(-1)}+XC[1]-0.25 ,
5\cdot(5\cdot XC[3]+11\cdot XC[2])^{(-1)}+XC[1]-0.29 ,
6\cdot(6\cdot XC[3]+10\cdot XC[2])^{(-1)}+XC[1]-0.32 ,
7\cdot(7\cdot XC[3]+9\cdot XC[2])^{(-1)}+XC[1]-0.35 ,
8\cdot(8\cdot XC[3]+8\cdot XC[2])^{(-1)}+XC[1]-0.39 ,
9\cdot(9\cdot XC[3]+7\cdot XC[2])^{(-1)}+XC[1]-0.37 ,
10\cdot(6\cdot XC[3]+6\cdot XC[2])^{(-1)}+XC[1]-0.58 ,
11\cdot(5\cdot XC[3]+5\cdot XC[2])^{(-1)}+XC[1]-0.73 ,
12\cdot(4\cdot XC[3]+4\cdot XC[2])^{(-1)}+XC[1]-0.96 ,
13\cdot(3\cdot XC[3]+3\cdot XC[2])^{(-1)}+XC[1]-1.34 ,
14\cdot(2\cdot XC[3]+2\cdot XC[2])^{(-1)}+XC[1]-2.1 ,
15\cdot(1\cdot XC[3]+1\cdot XC[2])^{(-1)}+XC[1]-4.39
\end{align*}
\]

\[
\begin{align*}
&\text{---------------------------,} \\
&\text{XC + 15.0 XC} \\
&\text{3 2} \\
&\text{---------------------------,} \\
&\text{XC + 7.0 XC} \\
&\text{3 2} \\
&\text{---------------------------,} \\
&\text{(XC - 0.22)XC} \\
&\text{1 3} \\
&\text{---------------------------,} \\
&\text{XC + 3.0 XC} \\
&\text{3 2} \\
&\text{---------------------------,} \\
&\text{(XC - 0.29)XC} \\
&\text{1 3} \\
&\text{---------------------------,} \\
&\text{XC + 2.2 XC} \\
&\text{3 2}
\end{align*}
\]
(X C - 4.39)X C + (X C - 4.39)X C + 15.0
-----------
1 3 1 2
-----------------------------
(X C + X C)
3 2
Type: Vector(Expression(Float))

lsfun1:ASP50(LSFUN1):= retract vef
(9) LSFUN1
Type: Asp50(LSFUN1)

result:=e04fdf(m,n,liw,lw,x,-1,lsfun1)

m:=15
n:=3
liw:=1

lw:=177

x:Matrix SF:= [[0.5, 1.0, 1.5]]

vef:Vector Expression Float:= vector

[(XC[3]+15*XC[2])^(-1)+XC[1]-0.14 ,
2*(2*XC[3]+14*XC[2])^(-1)+XC[1]-0.18 ,
3*(3*XC[3]+13*XC[2])^(-1)+XC[1]-0.22 ,
4*(4*XC[3]+12*XC[2])^(-1)+XC[1]-0.25 ,
5*(5*XC[3]+11*XC[2])^(-1)+XC[1]-0.29 ,
6*(6*XC[3]+10*XC[2])^(-1)+XC[1]-0.32 ,
7*(7*XC[3]+9*XC[2])^(-1)+XC[1]-0.35 ,
8*(8*XC[3]+8*XC[2])^(-1)+XC[1]-0.39 ,
9*(7*XC[3]+7*XC[2])^(-1)+XC[1]-0.37 ,
10*(6*XC[3]+6*XC[2])^(-1)+XC[1]-0.58 ,
11*(5*XC[3]+5*XC[2])^(-1)+XC[1]-0.73 ,
12*(4*XC[3]+4*XC[2])^(-1)+XC[1]-0.96 ,
13*(3*XC[3]+3*XC[2])^(-1)+XC[1]-1.34 ,
14*(2*XC[3]+2*XC[2])^(-1)+XC[1]-2.1 ,
--R 1 3 1 2
--R ----------------------------------------------------------,
--R XC + XC
--R 3 2
--R (XC - 0.58)XC + (XC - 0.58)XC + 1.6666666666 666666667
--R 1 3 1 2
--R ----------------------------------------------------------,
--R XC + XC
--R 3 2
--R (XC - 0.73)XC + (XC - 0.73)XC + 2.2
--R 1 3 1 2
--R ----------------------------------------------------------,
--R XC + XC
--R 3 2
--R (XC - 0.96)XC + (XC - 0.96)XC + 3.0
--R 1 3 1 2
--R ----------------------------------------------------------,
--R XC + XC
--R 3 2
--R (XC - 1.34)XC + (XC - 1.34)XC + 4.3333333333 333333333
--R 1 3 1 2
--R ----------------------------------------------------------,
--R XC + XC
--R 3 2
--R (XC - 2.1)XC + (XC - 2.1)XC + 7.0
--R 1 3 1 2
--R ----------------------------------------------------------,
--R XC + XC
--R 3 2
--R (XC - 4.39)XC + (XC - 4.39)XC + 15.0
--R 1 3 1 2
--R ----------------------------------------------------------

Type: Vector(Expression(Float))

--E 38

--S 39 of 153
lsfun2:Asp19(LSFUN2):= retract vef
--R
--R (9) LSFUN2
--R Type: Asp19(LSFUN2)
--E 39

--S 40 of 153
-- result:=e04gcf(m,n,liw,lw,x,-1,lsfun2)
--E 40

)clear all
showArrayValues true
R
(1) true
Type: Boolean
E 41

showScalarValues true
R
(2) true
Type: Boolean
E 42

n:=4
R
(3) 4
Type: PositiveInteger
E 43

ibound:=0
R
(4) 0
Type: NonNegativeInteger
E 44

liw:=6
R
(5) 6
Type: PositiveInteger
E 45

lw:=54
R
(6) 54
Type: PositiveInteger
E 46
bl: Matrix SF := [[1, -2, -1.0E-6, 1]]

bu: Matrix SF := [[3, 0, 1.0E6, 3]]

x: Matrix SF := [[3, -1, 0, 1]]

ef: Expression Float :=

10*(XC[1] - XC[4])^4)::EXPR FLOAT
funct1:Asp24(FUNCT1):=retract ef
--R
--R
--R  (11) FUNCT1
--R
--E 51

--S 52 of 153
-- result:=e04jaf(n,ibound,liw,lw,bl,pu,-1,funct1)
--E 52
)
clear all

--S 53 of 153
showArrayValues true
--R
--R
--R  (1) true
--R
--E 53

--S 54 of 153
showScalarValues true
--R
--R
--R  (2) true
--R
--E 54

--S 55 of 153
itmax:=20
--R
--R
--R  (3) 20
--R
--E 55

--S 56 of 153
msglvl1:=1
--R
--R
--R  (4) 1
--R
--E 56

--S 57 of 153
n:=7
--R
--R
nclin:=7

nctotl:=14

nrowa:=7

a:Matrix SF:=

[[1,1,1,1,1,1,1],
  [0.15,0.04,0.02,0.04,0.02,0.01,0.03],
  [0.03,0.05,0.08,0.02,0.06,0.01,0],
  [0.02,0.04,0.01,0.02,0.02,0,0],
  [0.02,0.03,0,0,0.01,0,0],
  [0.7,0.75,0.8,0.75,0.8,0.97,0],
  [0.02,0.06,0.08,0.12,0.02,0.01,0.97]]
```
\text{Type: Matrix(DoubleFloat)}
\]
cvec: Matrix SF:= [\[-0.02, -0.2, -0.2, -0.2, -0.2, 0.04, 0.04\]]

linobj:=true

x: Matrix SF:= [\[-0.01, -0.03, 0.0, -0.01, -0.1, 0.02, 0.01\]]
--R Type: Matrix(DoubleFloat)
--E 68

--S 69 of 153
result:=e04mbf(itmax,msglvl,n,nclin,nctotl,nrowa,a,bl,bu,_
         cvec,linobj,liwork,lwork,x,-1)
--E 69

)clear all

--S 70 of 153
showArrayValues true
--R
--R
--R (1) true
--R
--E 70

--S 71 of 153
showScalarValues true
--R
--R
--R (2) true
--R
--E 71

--S 72 of 153
itmax:=20
--R
--R
--R (3) 20
--R
--E 72

--S 73 of 153
msglvl:=-1
--R
--R
--R (4) -1
--R
--E 73

--S 74 of 153
n:=7
--R
--R
--R (5) 7
--R
--E 74
a:Matrix SF:=
\[
\begin{bmatrix}
1,1,1,1,1,1,1,1 \\
0.15,0.04,0.02,0.04,0.02,0.01,0.03 \\
0.03,0.05,0.08,0.02,0.06,0.01,0 \\
0.02,0.04,0.01,0.02,0.02,0,0 \\
0.02,0.03,0,0,0.01,0,0 \\
0.7,0.75,0.8,0.75,0.8,0.97,0 \\
0.02,0.06,0.08,0.12,0.02,0.01,0.97
\end{bmatrix}
\]

---R
---R (12)
---R [[1.,1.,1.,1.,1.,1.],
---R [0.14999999999999999, 3.9999999999999994E-2, 1.9999999999999997E-2,
---R 3.9999999999999994E-2, 1.9999999999999997E-2, 9.9999999999985E-3,
---R 2.9999999999999999E-2],
---R [2.9999999999999999E-2, 5.0000000000000003E-2, 7.9999999999999998E-2,
---R 1.9999999999999997E-2, 5.9999999999999998E-2, 9.9999999999985E-3, 0.],
---R [1.9999999999999997E-2, 3.9999999999999994E-2, 9.9999999999985E-3,
---R 1.9999999999999997E-2, 1.9999999999999997E-2, 0., 0.],
---R [1.9999999999999997E-2, 2.9999999999999999E-2, 0., 0.,
---R 9.9999999999985E-3, 0., 0.],
---R [0.69999999999999996, 0.75, 0.8000000000000004, 0.75,
---R 0.8000000000000004, 0.9699999999999997, 0.],
---R [1.9999999999999997E-2, 5.9999999999999998E-2, 7.9999999999999998E-2,
---R 0.12, 1.9999999999999997E-2, 9.9999999999985E-3, 0.9699999999999997]}
---R Type: Matrix(DoubleFloat)
---E 81

---S 82 of 153
bl:Matrix SF:=
\[
\begin{bmatrix}
-0.1 , -0.1 , -0.1 , -0.1 , -0.1 , -0.1 , -0.13 ,
-1.0e+21 , -1.0e+21 , -1.0e+21 , -1.0e+21 , -1.0e+21 , -0.0992 , -0.003
\end{bmatrix}
\]

---R
---R (13)
---R [ ]
---R [- 9.99999999999999985E-3, - 9.99999999999999992E-2,
---R - 9.99999999999999985E-3, - 3.99999999999999994E-2]
bu: Matrix SF:=

```
[[0.01 , 0.15 , 0.03 , 0.02 , 0.05 , 1.0e+21 , 1.0e+21 , -0.13 , -0.0049 , -0.0064 , -0.0037 , -0.0012 , 1.0e+21 , 0.002 ]
```

(c14)

```
Type: Matrix(DoubleFloat)
```

---

--S 83 of 153

```
cvec: Matrix SF:=

```

```
[[-0.02 , -0.2 , -0.2 , -0.2 , -0.2 , 0.04 , 0.04 ]
```

(c15)

```
Type: Matrix(DoubleFloat)
```

---

--S 84 of 153

```
featol: Matrix SF:=

```

```
[1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8, 1.0529999999999999E-8]
```

(c16)
hess := Matrix(2,0,0,0,0,0,0,1,  
    0,2,0,0,0,0,0,0,2,2,0,0,0,0,0,2,2,0,0,0,0,0,0,0,0,0,0,-2,-2,-2,-2)

cold := true

lpp := false

orthog := true
--R  
--R (20) true  
--R Type: Boolean  
--E 89

--S 90 of 153
liwork:=14
--R
--R  
--R (21) 14  
--R Type: PositiveInteger  
--E 90

--S 91 of 153
lwork:=238
--R
--R  
--R (22) 238  
--R Type: PositiveInteger  
--E 91

--S 92 of 153
x:Matrix SF:= 
[[-0.01 , -0.03 , 0.0 , -0.01 , -0.1 , 0.02 , 0.01 ]]
--R
--R  
--R (23)  
--R [  
--R [- 9.9999999999999985E-3, - 2.9999999999999999E-2, 0.,  
--R - 9.9999999999999992E-3, - 9.9999999999999992E-2, 1.9999999999999997E-2,  
--R 9.9999999999999993E-3]  
--R ]  
--R Type: Matrix(DoubleFloat)  
--E 92

--S 93 of 153
istate:Matrix Integer:= [[0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ]]  
--R  
--R  
--R (24) [0 0 0 0 0 0 0 0 0 0 0 0 0 0 ]  
--R Type: Matrix(Integer)  
--E 93

--S 94 of 153
mei:Matrix Expression Integer:=
[[2 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ],  
[0 ,2 ,0 ,0 ,0 ,0 ,0 ,0 ],  
[0 ,0 ,2 ,2 ,0 ,0 ,0 ,0 ],  
[0 ,0 ,2 ,2 ,0 ,0 ,0 ,0 ],  
[0 ,0 ,0 ,0 ,2 ,0 ,0 ,0 ],  
[0 ,0 ,0 ,0 ,0 ,-2 ,-2 ,0 ]]  
--E 94

--S 95 of 153
mei:Matrix Expression Integer:=
[[2 ,0 ,0 ,0 ,0 ,0 ,0 ,0 ],  
[0 ,2 ,0 ,0 ,0 ,0 ,0 ,0 ],  
[0 ,0 ,2 ,2 ,0 ,0 ,0 ,0 ],  
[0 ,0 ,2 ,2 ,0 ,0 ,0 ,0 ],  
[0 ,0 ,0 ,0 ,2 ,0 ,0 ,0 ],  
[0 ,0 ,0 ,0 ,0 ,-2 ,-2 ,0 ]]  
--E 95
\[
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & -2 & -2 \\
0 & 0 & 0 & 0 & 0 & -2 & -2 \\
+2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 2 & 2 & 0 & 0 & 0 \\
0 & 0 & 2 & 2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -2 & -2 \\
0 & 0 & 0 & 0 & 0 & -2 & -2 \\
+0 & 0 & 0 & 0 & 0 & -2 & -2 \\
\end{bmatrix}
\]

Type: Matrix(Expression(Integer))

qphess:Asp20(QPHESS):= retract mei

result:=e04naf(itmax,msglvl,n,nclin,nctotl,nrowa,nrowh,ncolh,
bigbnd,a,bl,bu,cvec,featol,
hess,cold,lpp,orthog,liwork,lwork,x,istate,-1,qphess)

);clear all

showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean
n:=9

Type: PositiveInteger

nclin:=4

Type: PositiveInteger

ncnln:=14

Type: PositiveInteger

nrowa:=4

Type: PositiveInteger

nrowj:=14

Type: PositiveInteger

nrowr:=9

Type: PositiveInteger
\[ a: \text{Matrix SF} := \begin{bmatrix} -1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & -1.0 & 1.0 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & -1.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 & -1.0 & 0.0 & 0.0 & 0.0 \end{bmatrix} \]

\[ b: \text{Matrix SF} := \begin{bmatrix} 0.0 & -1.0e15 & -1.0 \times 10^{15} & -1.0 \times 10^{15} & 0.0 & 0.0 & 0.0 & -1.0e15 \\ -1.0e15 & 0.0 & 0.0 & 0.0 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 \\ -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 \\ -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 & -1.0e15 \end{bmatrix} \]

\[ c: \text{Matrix SF} := \begin{bmatrix} 1.0e15 & 1.0e15 & 1.0 \times 10^{15} & 1.0e15 & 1.0e15 & 1.0e15 & 1.0e15 & 0.0 \times 10^{15} \\ 0.0 \times 10^{15} & 0.0 \times 10^{15} & 1.0e15 & 1.0e15 & 1.0e15 & 1.0e15 & 1.0e15 & 0.0 \times 10^{15} \\ 0.0 \times 10^{15} & 0.0 \times 10^{15} & 0.0 \times 10^{15} & 0.0 \times 10^{15} \end{bmatrix} \]
liwork:=59
  Type: PositiveInteger

lwork:=968
  Type: PositiveInteger

sta:=false
  Type: Boolean

cra:= 0.01
  Type: Float

der:=3
  Type: PositiveInteger

fea:=0.1053671201E-7
  Type: Float
fun:=0.4373903510E-14

hes:= true

infb:= 1.00E+15

infs:= 1.00E+15

linf:= 0.1053671201E-7

lint:= 0.9

list:= true
(24) true
Type: Boolean

--E 120

maji:= 30
Type: PositiveInteger

--E 121

majp:= 1
Type: PositiveInteger

--E 122

mini:= 81
Type: PositiveInteger

--E 123

minp:= 0
Type: NonNegativeInteger

--E 124

mon:= -1
Type: Integer

--E 125

nonf:= 1.05E-08

opt := 3.26E-08

ste := 2.0

stao := 1

stac := 1

stoo := 9

stoc := 9
ve:= 3

istate: Matrix Integer:=new(1,27,0)

$cjac$: Matrix SF:=new(14,9,0.0)

Type: PositiveInteger

Type: Matrix(Integer)
\begin{verbatim}
--R Type: Matrix(DoubleFloat)
--E 135

--S 136 of 153
clama:Matrix SF:=new(1,27,0.0)
--R
--R
--R (40)
--R [ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
--R ]
--R ) Type: Matrix(DoubleFloat)
--E 136

--S 137 of 153
r:Matrix SF:=new(9,9,0.0)
--R
--R
--R +0. 0. 0. 0. 0. 0. 0. 0. 0.+ 
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. |
--R | 0. 0. 0. 0. 0. 0. 0. 0. 0. | Type: Matrix(DoubleFloat)
--E 137

--S 138 of 153
x:Matrix SF:=
\{[0.1 ,0.125 ,0.666666 ,0.142857 ,0.111111 ,0.2 ,0.25 ,-0.2 ,-0.25 ]
--R
--R
--R (42)
--R [ 0.10000000000000001, 0.125, 0.66666599999999998, 0.14285700000000001,
--R 0.11111099999999999, 0.20000000000000001, 0.25, -0.19999999999999998,
--R -0.25]
--R )
--E 153
\end{verbatim}
PACKAGE NAGE04 NAGOPTIMISATIONPACKAGE

vef: Vector Expression Float := vector
  [X[1]^2 + X[6]^2 ,
   (X[2] - X[1])^2 + (X[7] - X[6])^2 ,
   (X[3] - X[1])^2 + X[6]*2 ,
   (X[1] - X[4])^2 + (X[6] - X[8])^2 ,
   X[2]^2 + X[7]^2 ,
   (X[4] - X[3])^2 + X[8]^2 ,
   X[4]^2 + X[8]^2 ,
   (X[1] - X[5])^2 + (X[6] - X[9])^2 ,

confun: Asp55(CONFUN):= retract vef

Type: Matrix(DoubleFloat)

Type: Vector(Expression(Float))
2492

CHAPTER 15. CHAPTER N

--R (44) CONFUN
--E 140

--S 141 of 153

--R

--R (45) X X + (- 1.0 X + X )X + (- 1.0 X + X )X - 1.0 X X
--R 4 9 5 3 8 3 1 7 2 6
--E 141

--S 142 of 153
objfun: Asp49(OBJFUN) := retract ef
--R
--R

--R (46) OBJFUN
--E 142

--S 143 of 153
result := e04ucf(n, nclin, ncnln, nrowa, nrowj, nrowr, a, bl, bu, liwork, lwork,
-- sta, cra, der, fea, fun, hes, infb, infl, lint, list, maji, majp,
-- mini, minp, mon, nonf, opt, ste, stao, stac, stoo, stoc, ve, istate,
-- cjac, clamda, r, x, -1, confun, objfun)
--E 143

)clear all

--S 144 of 153
showArrayValues true
--R
--R

--R (1) true
--R

--E 144

--S 145 of 153
showScalarValues true
--R
--R

--R (2) true
--R

--E 145

--S 146 of 153
job := 0
m:=15
n:=3
fsumsq:=0.0082148773065789729
s:Matrix SF:= 
\[
\begin{bmatrix}
4.0965034571419325 & 1.5949579400198182 & 0.061258491120317927 \\
\end{bmatrix}
\]
v:Matrix SF:=
\[
\begin{bmatrix}
0.9353959087369782, & -0.25922842715810668, & -0.24048932714625143, \\
0.35295122094583287, & 0.64323460044494585, & 0.67946647041872243
\end{bmatrix}
\]
This package uses the NAG Library to perform optimization. An optimization problem involves minimizing a function (called the objective function) of several variables, possibly subject to restrictions on the values of the variables defined by a set of constraint functions. The routines in the NAG Foundation Library are concerned with function minimization only, since the problem of maximizing a given function can be transformed into a minimization problem simply by multiplying the function by $-1$.

Contents of this Introduction:

1. Scope of the Chapter
2. Background to the Problems
   2.1. Types of Optimization Problems
       2.1.1. Unconstrained minimization
       2.1.2. Nonlinear least-squares problems
2.1.3. Minimization subject to bounds on the variables

2.1.4. Minimization subject to linear constraints

2.1.5. Minimization subject to nonlinear constraints

2.2. Geometric Representation and Terminology

2.2.1. Gradient vector

2.2.2. Hessian matrix

2.2.3. Jacobian matrix; matrix of constraint normals

2.3. Sufficient Conditions for a Solution

2.3.1. Unconstrained minimization

2.3.2. Minimization subject to bounds on the variables

2.3.3. Linearly-constrained minimization

2.3.4. Nonlinearly-constrained minimization

2.4. Background to Optimization Methods

2.4.1. Methods for unconstrained optimization

2.4.2. Methods for nonlinear least-squares problems

2.4.3. Methods for handling constraints

2.5. Scaling

2.5.1. Transformation of variables

2.5.2. Scaling the objective function

2.5.3. Scaling the constraints

2.6. Analysis of Computed Results

2.6.1. Convergence criteria

2.6.2. Checking results

2.6.3. Monitoring progress

2.6.4. Confidence intervals for least-squares solutions
1. Scope of the Chapter

An optimization problem involves minimizing a function (called the objective function) of several variables, possibly subject to restrictions on the values of the variables defined by a set of constraint functions. The routines in the NAG Foundation Library are concerned with function minimization only, since the problem of maximizing a given function can be transformed into a minimization problem simply by multiplying the function by -1.

This introduction is only a brief guide to the subject of optimization designed for the casual user. Anyone with a difficult or protracted problem to solve will find it beneficial to consult a more detailed text, such as Gill et al [5] or Fletcher [3].

Readers who are unfamiliar with the mathematics of the subject may find some sections difficult at first reading; if so, they should concentrate on Sections 2.1, 2.2, 2.5, 2.6 and 3.

2. Background to the Problems

2.1. Types of Optimization Problems

Solution of optimization problems by a single, all-purpose, method is cumbersome and inefficient. Optimization problems are therefore classified into particular categories, where each category is defined by the properties of the objective and constraint functions, as illustrated by some examples below.

<table>
<thead>
<tr>
<th>Properties of Objective Function</th>
<th>Properties of Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nonlinear</td>
<td>Nonlinear</td>
</tr>
<tr>
<td>Sums of squares of</td>
<td>Sparse linear</td>
</tr>
</tbody>
</table>
nonlinear functions

Quadratic   Linear

Sums of squares of linear Bounds
functions

Linear       None

For instance, a specific problem category involves the minimization of a nonlinear objective function subject to bounds on the variables. In the following sections we define the particular categories of problems that can be solved by routines contained in this Chapter.

2.1.1. Unconstrained minimization

In unconstrained minimization problems there are no constraints on the variables. The problem can be stated mathematically as follows:

$$\minimize F(x)$$

where $x$ is in $\mathbb{R}^n$, that is, $x = (x_1, x_2, \ldots, x_n)$.

2.1.2. Nonlinear least-squares problems

Special consideration is given to the problem for which the function to be minimized can be expressed as a sum of squared functions. The least-squares problem can be stated mathematically as follows:

$$\minimize \{ f \}_{i=1}^{m} f(x), \quad x \in \mathbb{R}^n$$

where the $i$th element of the $m$-vector $f$ is the function $f_i(x)$.

2.1.3. Minimization subject to bounds on the variables

These problems differ from the unconstrained problem in that at least one of the variables is subject to a simple restriction on its value, e.g. $x \leq 10$, but no constraints of a more general form
are present.

The problem can be stated mathematically as follows:

\[
\begin{align*}
\text{minimize } & \quad F(x), \quad x \text{ is in } \mathbb{R}^n \\
\text{subject to } & \quad l_i \leq x_i \leq u_i, \quad i=1,2,\ldots,n.
\end{align*}
\]

This format assumes that upper and lower bounds exist on all the variables. By conceptually allowing \( u_i = \infty \) and \( l_i = -\infty \) all the variables need not be restricted.

2.1.4. Minimization subject to linear constraints

A general linear constraint is defined as a constraint function that is linear in more than one of the variables, e.g. \( 3x_1 + 2x_2 \geq 4 \).

The various types of linear constraint are reflected in the following mathematical statement of the problem:

\[
\begin{align*}
\text{minimize } & \quad F(x), \quad x \text{ is in } \mathbb{R}^n \\
\text{subject to the } & \quad \begin{align*}
T & \quad \text{equality } \quad a_i x = b_i, \quad i=1,2,\ldots,m; \\
\text{constraints: } & \quad i_i 1 \\
T & \quad \text{inequality } \quad a_i x \geq b_i, \quad i=m+1,m+2,\ldots,m; \\
\text{constraints: } & \quad i_i 1 1 2 \\
T & \quad a_i x \leq b_i, \quad i=m+1,m+2,\ldots,m; \\
\text{constraints: } & \quad i_i 2 2 3 \\
T & \quad \text{range } \quad s \leq a_j x \leq t_j, \quad i=m+1,m+2,\ldots,m; \\
\text{constraints: } & \quad j_i j 3 3 4 \\
\text{bounds } & \quad l_i \leq x_i \leq u_i, \quad i=1,2,\ldots,n
\end{align*}
\]

where each \( a_i \) is a vector of length \( n \); \( b, s \) and \( t \) are constant

scalars; and any of the categories may be empty.

Although the bounds on \( x \) could be included in the definition of

general linear constraints, we prefer to distinguish between them

for reasons of computational efficiency.

If \( F(x) \) is a linear function, the linearly-constrained problem is

termed a linear programming problem (LP problem); if \( F(x) \) is a

quadratic function, the problem is termed a quadratic programming

problem (QP problem). For further discussion of LP and QP

problems, including the dual formulation of such problems, see

Dantzig [2].

2.1.5. Minimization subject to nonlinear constraints

A problem is included in this category if at least one constraint

function is nonlinear, e.g. \( x_1^2 + x_3^2 - 2 > 0 \). The mathematical

statement of the problem is identical to that for the linearly-

constrained case, except for the addition of the following

constraints:

\[
\begin{align*}
\text{equality} & \quad c_i(x) = 0 \quad i=1,2,\ldots,m; \\
\text{constraints:} & \quad i \\
\text{inequality} & \quad c_i(x) \geq 0 \quad i=m+1,m+2,\ldots,m; \\
\text{constraints:} & \quad i \\
\text{range} & \quad v \leq c_i(x) \leq w \quad i=m+1,m+2,\ldots,m, \\
\text{constraints:} & \quad j \quad j \\
\end{align*}
\]

where each \( c_i \) is a nonlinear function; \( v \) and \( w \) are constant

scalars; and any category may be empty. Note that we do not

include a separate category for constraints of the form \( c_i(x) \leq 0 \),

since this is equivalent to \( -c_i(x) \geq 0 \).

2.2. Geometric Representation and Terminology

To illustrate the nature of optimization problems it is useful to
consider the following example in two dimensions

\[
\begin{array}{ccc}
 1 & 2 & 2 \\
1 & 2 & 1 & 2 & 2 \\
\end{array}
\]

\[F(x) = e^{(4x + 2x + 4x + 2x + 1)}.\]

(This function is used as the example function in the documentation for the unconstrained routines.)

Figure 1
Please see figure in printed Reference Manual

Figure 1 is a contour diagram of \(F(x)\). The contours labelled \(F_0, F_1, \ldots, F_4\) are isovalue contours, or lines along which the function \(F(x)\) takes specific constant values. The point \(x^*\) is a local unconstrained minimum, that is, the value of \(F(x^*)\) is less than at all the neighbouring points. A function may have several such minima. The lowest of the local minima is termed a global minimum. In the problem illustrated in Figure 1, \(x^*\) is the only local minimum. The point \(x\) is said to be a saddle point because it is a minimum along the line AB, but a maximum along CD.

If we add the constraint \(x \geq 0\) to the problem of minimizing \(F(x)\), the solution remains unaltered. In Figure 1 this constraint is represented by the straight line passing through \(x = 0\), and the shading on the line indicates the unacceptable region. The region in \(\mathbb{R}\) satisfying the constraints of an optimization problem is termed the feasible region. A point satisfying the constraints is defined as a feasible point.

If we add the nonlinear constraint \(x_1 + x_2 - x_1 x_2 - 1.5 \geq 0\), represented by the curved shaded line in Figure 1, then \(x\) is not a feasible point. The solution of the new constrained problem is \(x\), the feasible point with the smallest function value.

2.2.1. Gradient vector
The vector of first partial derivatives of $F(x)$ is called the
gradient vector, and is denoted by $g(x)$, i.e.,

$$
g(x) = \left[ \frac{\partial F(x)}{\partial x_1}, \frac{\partial F(x)}{\partial x_2}, \ldots, \frac{\partial F(x)}{\partial x_n} \right]^T.
$$

For the function illustrated in Figure 1,

$$
g(x) = \left[ e^{(4x_1 + 4x_2 + 2)} \right]_2.
$$

The gradient vector is of importance in optimization because it
must be zero at an unconstrained minimum of any function with
continuous first derivatives.

2.2.2. Hessian matrix

The matrix of second partial derivatives of a function is termed
its Hessian matrix. The Hessian matrix of $F(x)$ is denoted by $G(x)$
and its $(i,j)$th element is given by $\frac{\partial^2 F(x)}{\partial x_i \partial x_j}$. If $F(x)$
has continuous second derivatives, then $G(x)$ must be positive
semi-definite at any unconstrained minimum of $F$.

2.2.3. Jacobian matrix; matrix of constraint normals

In nonlinear least-squares problems, the matrix of first partial
derivatives of the vector-valued function $f(x)$ is termed the
Jacobian matrix of $f(x)$ and its $(i,j)$th component is $\frac{\partial f_i}{\partial x_j}$.

The vector of first partial derivatives of the constraint $c_i(x)$
is denoted by

$$
c_i(x) = \left[ \frac{\partial c_i(x)}{\partial x_1}, \frac{\partial c_i(x)}{\partial x_2}, \ldots, \frac{\partial c_i(x)}{\partial x_n} \right]_n.
$$
At a point, \( x \), the vector \( \mathbf{a}(x) \) is orthogonal (normal) to the \( \mathbf{i} \) isovalue contour of \( c(x) \) passing through \( x \); this relationship is \( \mathbf{i} \) illustrated for a two-dimensional function in Figure 2.

Figure 2
Please see figure in printed Reference Manual

The matrix whose columns are the vectors \( \{\mathbf{a}_i\} \) is termed the \( \mathbf{i} \) matrix of constraint normals. Note that if \( c(x) \) is a linear \( \mathbf{1} \) constraint involving \( \mathbf{a}_i \), then its vector of first partial \( \mathbf{i} \) derivatives is simply the vector \( \mathbf{a}_i \).

2.3. Sufficient Conditions for a Solution

All nonlinear functions will be assumed to have continuous second derivatives in the neighbourhood of the solution.

2.3.1. Unconstrained minimization

* The following conditions are sufficient for the point \( x \) to be an unconstrained local minimum of \( F(x) \):

* (i) \( |||\mathbf{g}(x)|||=0 \); and

* (ii) \( \mathbf{G}(x) \) is positive-definite,

where \( ||\mathbf{g}|| \) denotes the Euclidean length of \( g \).

2.3.2. Minimization subject to bounds on the variables

At the solution of a bounds-constrained problem, variables which are not on their bounds are termed free variables. If it is known in advance which variables are on their bounds at the solution, the problem can be solved as an unconstrained problem in just the free variables; thus, the sufficient conditions for a solution are similar to those for the unconstrained case, applied only to
the free variables.

Sufficient conditions for a feasible point \( x \) to be the solution of a bound-constrained problem are as follows:

* (i) \( ||g(x)||=0; \) and

* (ii) \( G(x) \) is positive-definite; and

* (iii) \( g_j(x) < 0, x = u_j \); \( g_j(x) > 0, x = l_j \),

where \( g(x) \) is the gradient of \( F(x) \) with respect to the free variables, and \( G(x) \) is the Hessian matrix of \( F(x) \) with respect to the free variables. The extra condition (iii) ensures that \( F(x) \) cannot be reduced by moving off one or more of the bounds.

2.3.3. Linearly-constrained minimization

For the sake of simplicity, the following description does not include a specific treatment of bounds or range constraints, since the results for general linear inequality constraints can be applied directly to these cases.

* At a solution \( x \) of a linearly-constrained problem, the constraints which hold as equalities are called the active or binding constraints. Assume that there are \( t \) active constraints at the solution \( x \), and let \( A \) denote the matrix whose columns are the columns of \( A \) corresponding to the active constraints, with \( b \) the vector similarly obtained from \( b \); then

\[ ^T A x = b. \]

The matrix \( Z \) is defined as an \( n \) by \( (n-t) \) matrix satisfying:

\[ ^T T \]
\[ A Z=0; \quad Z Z=I. \]
The columns of $Z$ form an orthogonal basis for the set of vectors orthogonal to the columns of $A$.

Define

$$
g^T(x) = Z^T g(x), \text{ the projected gradient vector of } F(x);$$

$$
T^T G(x) = Z^T G(x) Z, \text{ the projected Hessian matrix of } F(x).
$$

At the solution of a linearly-constrained problem, the projected gradient vector must be zero, which implies that the gradient vector $g(x)$ can be written as a linear combination of the columns of $A$, i.e., $g(x) = \sum_{i=1}^t (\lambda_i a_i) = A(\lambda)$. The scalar $\lambda_i$ is defined as the Lagrange multiplier corresponding to the $i$th active constraint. A simple interpretation of the $i$th Lagrange multiplier is that it gives the gradient of $F(x)$ along the $i$th active constraint normal; a convenient definition of the Lagrange multiplier vector (although not a recommended method for computation) is:

$$
(\lambda) = (A^T A)^{-1} A^T g(x).
$$

Sufficient conditions for $x$ to be the solution of a linearly-constrained problem are:

(i) $x$ is feasible, and $A x = b$; and

(ii) $\|g(x)\|_2 = 0$, or equivalently, $g(x) = A(\lambda)$; and

(iii) $G(x)$ is positive-definite; and
(iv) \( (\lambda)_i > 0 \) if \( (\lambda)_i \) corresponds to a constraint
\[ \begin{align*}
^T a_i x & \geq b_i; \\
^T a_i x & \leq b_i.
\end{align*} \]

\((\lambda)_i < 0\) if \( (\lambda)_i \) corresponds to a constraint
\[ \begin{align*}
^T a_i x & \geq b_i; \\
^T a_i x & \leq b_i.
\end{align*} \]

The sign of \( (\lambda)_i \) is immaterial for equality constraints, which by definition are always active.

2.3.4. Nonlinearly-constrained minimization

For nonlinearly-constrained problems, much of the terminology is defined exactly as in the linearly-constrained case. The set of active constraints at \( x \) again means the set of constraints that hold as equalities at \( x \), with corresponding definitions of \( c \) and \( A \): the vector \( c(x) \) contains the active constraint functions, and the columns of \( A(x) \) are the gradient vectors of the active constraints. As before, \( Z \) is defined in terms of \( A(x) \) as a matrix such that:

\[ ^T A Z = 0; \quad Z Z = I \]

where the dependence on \( x \) has been suppressed for compactness.

The projected gradient vector \( g^*_z(x) \) is the vector \( Z g(x) \). At the solution \( x \) of a nonlinearly-constrained problem, the projected gradient must be zero, which implies the existence of Lagrange multipliers corresponding to the active constraints, i.e.,

\[ g(x) = A(x) (\lambda). \]

The Lagrangian function is given by:
\[ L(x, \lambda) = F(x) - \lambda c(x). \]

We define \( g(x) \) as the gradient of the Lagrangian function; \( L(x) \) as its Hessian matrix, and \( L^T \) as its projected Hessian matrix, i.e., \( L^T = Z L G Z \).

Sufficient conditions for \( x \) to be a solution of nonlinearly-constrained problem are:

- (i) \( x \) is feasible, and \( c(x) = 0 \); and
- (ii) \( ||g(x)|| = 0 \), or, equivalently, \( g(x) = A(x) \lambda \); and
- (iii) \( L(x) \) is positive-definite; and
- (iv) \( \lambda > 0 \) if \( \lambda \) corresponds to a constraint of the form \( c \geq 0 \); the sign of \( \lambda \) is immaterial for an equality constraint.

Note that condition (ii) implies that the projected gradient of the Lagrangian function must also be zero at \( x \), since the application of \( Z \) annihilates the matrix \( A(x) \).

2.4. Background to Optimization Methods

All the algorithms contained in this Chapter generate an iterative sequence \( \{x(k)\} \) that converges to the solution \( x \) in the limit, except for some special problem categories (i.e., linear and quadratic programming). To terminate computation of the sequence, a convergence test is performed to determine whether the current estimate of the solution is an adequate approximation. The convergence tests are discussed in Section 2.6.
Most of the methods construct a sequence \{x^{(k)}\} satisfying:

\[ x^{(k+1)} = x^{(k)} + (\alpha^{(k)}) p^{(k)} \]

where the vector \( p^{(k)} \) is termed the direction of search, and \( (\alpha^{(k)}) \) is the steplength. The steplength \( (\alpha^{(k)}) \) is chosen so that \( F(x^{(k+1)}) < F(x^{(k)}) \).

2.4.1. Methods for unconstrained optimization

The distinctions among methods arise primarily from the need to use varying levels of information about derivatives of \( F(x) \) in defining the search direction. We describe three basic approaches to unconstrained problems, which may be extended to other problem categories. Since a full description of the methods would fill several volumes, the discussion here can do little more than allude to the processes involved, and direct the reader to other sources for a full explanation.

(a) Newton-type Methods (Modified Newton Methods)

Newton-type methods use the Hessian matrix \( G(x^{(k)}) \), or a finite difference approximation to \( G(x^{(k)}) \), to define the search direction. The routines in the Library either require a subroutine that computes the elements of \( G(x^{(k)}) \), or they approximate \( G(x^{(k)}) \) by finite differences.

Newton-type methods are the most powerful methods available for general problems and will find the minimum of a quadratic function in one iteration. See Sections 4.4. and 4.5.1. of Gill et al [5].

(b) Quasi-Newton Methods

Quasi-Newton methods approximate the Hessian \( G(x^{(k)}) \) by a matrix \( B^{(k)} \), which is modified at each iteration to include information obtained about the curvature of \( F \) along the latest search direction. Although not as robust as Newton-
type methods, quasi-Newton methods can be more efficient (k) because G(x) is not computed, or approximated by finite-differences. Quasi-Newton methods minimize a quadratic function in n iterations. See Section 4.5.2 of Gill et al [5].

(c) Conjugate-Gradient Methods

Unlike Newton-type and quasi-Newton methods, conjugate gradient methods do not require the storage of an n by n matrix and so are ideally suited to solve large problems. Conjugate-gradient type methods are not usually as reliable or efficient as Newton-type, or quasi-Newton methods. See Section 4.8.3 of Gill et al [5].

2.4.2. Methods for nonlinear least-squares problems

These methods are similar to those for unconstrained optimization, but exploit the special structure of the Hessian matrix to give improved computational efficiency.

Since

\[
F(x) = \sum_{i=1}^{m-2} f(x)
\]

the Hessian matrix G(x) is of the form

\[
G(x) = 2 \sum_{i=1}^{m} [J(x) J(x)^T] + \sum_{i=1}^{T} f(x) G_i(x),
\]

where J(x) is the Jacobian matrix of f(x), and G_i(x) is the Hessian matrix of f(x).

In the neighbourhood of the solution, \|\|f(x)\|\| is often small compared to \|\|J(x) J(x)^T\|\| (for example, when f(x) represents the goodness of fit of a nonlinear model to observed data). In such cases, 2J(x) J(x) may be an adequate approximation to G(x),
thereby avoiding the need to compute or approximate second derivatives of \( f(x) \). See Section 4.7 of Gill et al [5].

2.4.3. Methods for handling constraints

Bounds on the variables are dealt with by fixing some of the variables on their bounds and adjusting the remaining free variables to minimize the function. By examining estimates of the Lagrange multipliers it is possible to adjust the set of variables fixed on their bounds so that eventually the bounds active at the solution should be correctly identified. This type of method is called an active set method. One feature of such methods is that, given an initial feasible point, all \( (k) \) approximations \( x^{(k)} \) are feasible. This approach can be extended to general linear constraints. At a point, \( x \), the set of constraints which hold as equalities being used to predict, or approximate, the set of active constraints is called the working set.

Nonlinear constraints are more difficult to handle. If at all possible, it is usually beneficial to avoid including nonlinear constraints during the formulation of the problem. The methods currently implemented in the Library handle nonlinearly constrained problems either by transforming them into a sequence of bound constraint problems, or by transforming them into a sequence of quadratic programming problems. A feature of almost \( (k) \) all methods for nonlinear constraints is that \( x^{(k)} \) is not guaranteed to be feasible except in the limit, and this is certainly true of the routines currently in the Library. See Chapter 6, particularly Section 6.4 and Section 6.5 of Gill et al [5].

Anyone interested in a detailed description of methods for optimization should consult the references.

2.5. Scaling

Scaling (in a broadly defined sense) often has a significant influence on the performance of optimization methods. Since convergence tolerances and other criteria are necessarily based on an implicit definition of 'small' and 'large', problems with unusual or unbalanced scaling may cause difficulties for some algorithms. Nonetheless, there are currently no scaling routines in the Library, although the position is under constant review. In light of the present state of the art, it is considered that sensible scaling by the user is likely to be more effective than any automatic routine. The following sections present some
general comments on problem scaling.

2.5.1. Transformation of variables

One method of scaling is to transform the variables from their original representation, which may reflect the physical nature of the problem, to variables that have certain desirable properties in terms of optimization. It is generally helpful for the following conditions to be satisfied:

(i) the variables are all of similar magnitude in the region of interest;

(ii) a fixed change in any of the variables results in similar changes in $F(x)$. Ideally, a unit change in any variable produces a unit change in $F(x)$;

(iii) the variables are transformed so as to avoid cancellation error in the evaluation of $F(x)$.

Normally, users should restrict themselves to linear transformations of variables, although occasionally nonlinear transformations are possible. The most common such transformation (and often the most appropriate) is of the form

$$x_{\text{new}} = D x_{\text{old}},$$

where $D$ is a diagonal matrix with constant coefficients. Our experience suggests that more use should be made of the transformation

$$x_{\text{new}} = D x_{\text{old}} + v,$$

where $v$ is a constant vector.

Consider, for example, a problem in which the variable $x$ represents the position of the peak of a Gaussian curve to be fitted to data for which the extreme values are 150 and 170; therefore $x$ is known to lie in the range 150--170. One possible scaling would be to define a new variable $x$, given by

$$x$$
A better transformation, however, is given by defining $x$ as

\[
x = \frac{x - 160}{3}.
\]

Frequently, an improvement in the accuracy of evaluation of $F(x)$ can result if the variables are scaled before the routines to evaluate $F(x)$ are coded. For instance, in the above problem just mentioned of Gaussian curve fitting, $x$ may always occur in terms of the form $(x - x_m^3)$, where $x_m$ is a constant representing the mean peak position.

2.5.2. Scaling the objective function

The objective function has already been mentioned in the discussion of scaling the variables. The solution of a given problem is unaltered if $F(x)$ is multiplied by a positive constant, or if a constant value is added to $F(x)$. It is generally preferable for the objective function to be of the order of unity in the region of interest; thus, if in the original formulation $F(x)$ is always of the order of $10^{+5}$ (say), then the value of $F(x)$ should be multiplied by $10^{-5}$ when evaluating the function within the optimization routines. If a constant is added or subtracted in the computation of $F(x)$, usually it should be omitted - i.e., it is better to formulate $F(x)$ as $x + x$ rather than as $x + x + 1000$ or even $x + x + 1$. The inclusion of such a constant in the calculation of $F(x)$ can result in a loss of significant figures.

2.5.3. Scaling the constraints

The solution of a nonlinearly-constrained problem is unaltered if the $i$th constraint is multiplied by a positive weight $w_i$. At the approximation of the solution determined by a Library routine,
the active constraints will not be satisfied exactly, but will have 'small' values (for example, $c_1 = 10$, $c_2 = 10$, etc.). In general, this discrepancy will be minimized if the constraints are weighted so that a unit change in $x$ produces a similar change in each constraint.

A second reason for introducing weights is related to the effect of the size of the constraints on the Lagrange multiplier estimates and, consequently, on the active set strategy. Additional discussion is given in Gill et al [5].

2.6. Analysis of Computed Results

2.6.1. Convergence criteria

The convergence criteria inevitably vary from routine to routine, since in some cases more information is available to be checked (for example, is the Hessian matrix positive-definite?), and different checks need to be made for different problem categories (for example, in constrained minimization it is necessary to verify whether a trial solution is feasible). Nonetheless, the underlying principles of the various criteria are the same; in non-mathematical terms, they are:

(i) is the sequence $\{x_k\}$ converging?

(ii) is the sequence $\{F_k\}$ converging?

(iii) are the necessary and sufficient conditions for the solution satisfied?

The decision as to whether a sequence is converging is necessarily speculative. The criterion used in the present routines is to assume convergence if the relative change occurring between two successive iterations is less than some prescribed quantity. Criterion (iii) is the most reliable but often the conditions cannot be checked fully because not all the required information may be available.

2.6.2. Checking results

Little a priori guidance can be given as to the quality of the solution found by a nonlinear optimization algorithm, since no guarantees can be given that the methods will always work. Therefore, it is necessary for the user to check the computed solution even if the routine reports success. Frequently a
solution' may have been found even when the routine does not report a success. The reason for this apparent contradiction is that the routine needs to assess the accuracy of the solution. This assessment is not an exact process and consequently may be unduly pessimistic. Any 'solution' is in general only an approximation to the exact solution, and it is possible that the accuracy specified by the user is too stringent.

Further confirmation can be sought by trying to check whether or not convergence tests are almost satisfied, or whether or not some of the sufficient conditions are nearly satisfied. When it is thought that a routine has returned a non-zero value of IFAIL only because the requirements for 'success' were too stringent it may be worth restarting with increased convergence tolerances.

For nonlinearly-constrained problems, check whether the solution returned is feasible, or nearly feasible; if not, the solution returned is not an adequate solution.

Confidence in a solution may be increased by resolving the problem with a different initial approximation to the solution. See Section 8.3 of Gill et al [5] for further information.

2.6.3. Monitoring progress

Many of the routines in the Chapter have facilities to allow the user to monitor the progress of the minimization process, and users are encouraged to make use of these facilities. Monitoring information can be a great aid in assessing whether or not a satisfactory solution has been obtained, and in indicating difficulties in the minimization problem or in the routine's ability to cope with the problem.

The behaviour of the function, the estimated solution and first derivatives can help in deciding whether a solution is acceptable and what to do in the event of a return with a non-zero value of IFAIL.

2.6.4. Confidence intervals for least-squares solutions

When estimates of the parameters in a nonlinear least-squares problem have been found, it may be necessary to estimate the variances of the parameters and the fitted function. These can be calculated from the Hessian of \( F(x) \) at the solution.

In many least-squares problems, the Hessian is adequately approximated at the solution by \( G=2J^T J \) (see Section 2.4.3). The Jacobian, \( J \), or a factorization of \( J \) is returned by all the comprehensive least-squares routines and, in addition, a routine
is supplied in the Library to estimate variances of the parameters following the use of most of the nonlinear least-squares routines, in the case that \( G = 2J \) is an adequate approximation.

Let \( H \) be the inverse of \( G \), and \( S \) be the sum of squares, both calculated at the solution \( x \); an unbiased estimate of the variance of the \( i \)th parameter \( x \) is

\[
\text{var } x = \frac{2S}{i \cdot (m-n)^2}
\]

and an unbiased estimate of the covariance of \( x \) and \( x \) is

\[
\text{covar}(x, x) = \frac{2S}{i \cdot j \cdot (m-n)^2}
\]

* If \( x \) is the true solution, then the \( 100(1-(\beta)) \) confidence interval on \( x \) is

\[
\frac{x - \text{var } x \cdot t}{\sqrt{i \cdot (1-(\beta)/2, m-n) \cdot i}} < x < \frac{x + \text{var } x \cdot t}{\sqrt{i \cdot (1-(\beta)/2, m-n) \cdot i}}
\]

where \( t \) is the \( 100(1-(\beta))/2 \) percentage point of the \( t \)-distribution with \( m-n \) degrees of freedom.

In the majority of problems, the residuals \( f_i \), for \( i = 1, 2, \ldots, m \),
contain the difference between the values of a model function 
(\(\phi\))(z, x) calculated for \(m\) different values of the independent 
variable z, and the corresponding observed values at these 
points. The minimization process determines the parameters, or 

constants x, of the fitted function (\(\phi\))(z, x). For any value, z, 
of the independent variable z, an unbiased estimate of the 
variance of (\(\phi\)) is 

\[
\text{var} (\phi) = \frac{2S}{m-n} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{d}{ddx_i} \frac{d}{ddx_j} \phi(z, x) 
\]

The 100(1-(\(\beta\))) confidence interval on \(F\) at the point \(z\) is 

\[
(\phi)(z, x) - \frac{\text{var} (\phi).t}{2} < (\phi)(z, x) < (\phi)(z, x) + \frac{\text{var} (\phi).t}{2}, \quad ((\beta)/2, m-n) 
\]

For further details on the analysis of least-squares solutions 
see Bard [1] and Wolberg [7].

2.7. References

Press.

Princeton University Press.

(2nd Edition).


CHAPTER 15. CHAPTER N


3. Recommendations on Choice and Use of Routines

The choice of routine depends on several factors: the type of problem (unconstrained, etc.); the level of derivative information available (function values only, etc.); the experience of the user (there are easy-to-use versions of some routines); whether or not storage is a problem; and whether computational time has a high priority.

3.1. Choice of Routine

Routines are provided to solve the following types of problem:

Nonlinear Programming E04UCF
Quadratic Programming E04NAF
Linear Programming E04MBF
Nonlinear Function E04DGF
(using 1st derivatives)
Nonlinear Function, unconstrained or simple bounds E04JAF
(using function values only)
Nonlinear least-squares E04FDF
(using function values only)
Nonlinear least-squares E04GCF
(using function values and 1st derivatives)

E04UCF can be used to solve unconstrained, bound-constrained and linearly-constrained problems.

E04NAF can be used as a comprehensive linear programming solver; however, in most cases the easy-to-use routine E04MBF will be adequate.

E04MBF can be used to obtain a feasible point for a set of linear constraints.

E04DGF can be used to solve large scale unconstrained problems.

The routines can be used to solve problems in a single variable.

3.2. Service Routines

One of the most common errors in use of optimization routines is that the user's subroutines incorrectly evaluate the relevant partial derivatives. Because exact gradient information normally
enhances efficiency in all areas of optimization, the user should be encouraged to provide analytical derivatives whenever possible. However, mistakes in the computation of derivatives can result in serious and obscure run-time errors, as well as complaints that the Library routines are incorrect.

E04UCF incorporates a check on the gradients being supplied and users are encouraged to utilize this option; E04GCF also incorporates a call to a derivative checker.

E04YCF estimates selected elements of the variance-covariance matrix for the computed regression parameters following the use of a nonlinear least-squares routine.

3.3. Function Evaluations at Infeasible Points

Users must not assume that the routines for constrained problems will require the objective function to be evaluated only at points which satisfy the constraints, i.e., feasible points. In the first place some of the easy-to-use routines call a service routine which will evaluate the objective function at the user-supplied initial point, and at neighbouring points (to check user-supplied derivatives or to estimate intervals for finite differencing). Apart from this, all routines will ensure that any evaluations of the objective function occur at points which approximately satisfy any simple bounds or linear constraints. Satisfaction of such constraints is only approximate because:

(a) routines which have a parameter FEATOL may allow such constraints to be violated by a margin specified by FEATOL;

(b) routines which estimate derivatives by finite differences may require function evaluations at points which just violate such constraints even though the current iteration just satisfies them.

There is no attempt to ensure that the current iteration satisfies any nonlinear constraints. Users who wish to prevent their objective function being evaluated outside some known region (where it may be undefined or not practically computable), may try to confine the iteration within this region by imposing suitable simple bounds or linear constraints (but beware as this may create new local minima where these constraints are active).

Note also that some routines allow the user-supplied routine to return a parameter (MODE) with a negative value to force an immediate clean exit from the minimization when the objective function cannot be evaluated.

3.4. Related Problems
Apart from the standard types of optimization problem, there are other related problems which can be solved by routines in this or other chapters of the Library.

E04MBF can be used to find a feasible point for a set of linear constraints and simple bounds.

Two routines in Chapter F04 solve linear least-squares problems,

\[ \text{minimize } \sum_{i=1}^{m} \sum_{j=1}^{n} r_i(x) \text{ where } r_i(x) = b_i - a_{ij}x_j. \]

E02GAF solves an overdetermined system of linear equations in the \(l_1\) norm, i.e., minimizes \(\sum_{i=1}^{m} |r_i(x)|\), with \(r_i\) as above.

E04 -- Minimizing or Maximizing a Function

Minimizing or Maximizing a Function

E04DGF Unconstrained minimum, pre-conditioned conjugate gradient algorithm, function of several variables using 1st derivatives

E04DJF Read optional parameter values for E04DGF from external file

E04DKF Supply optional parameter values to E04DGF

E04FDF Unconstrained minimum of a sum of squares, combined Gauss-Newton and modified Newton algorithm using function values only

E04GCF Unconstrained minimum of a sum of squares, combined Gauss-Newton and quasi-Newton algorithm, using 1st derivatives

E04JAF Minimum, function of several variables, quasi-Newton algorithm, simple bounds, using function values only

E04MBF Linear programming problem
E04NAF  Quadratic programming problem
E04UCF  Minimum, function of several variables, sequential QP method, nonlinear constraints, using function values and optionally 1st derivatives
E04UDF  Read optional parameter values for E04UCF from external file
E04UEF  Supply optional parameter values to E04UCF
E04YCF  Covariance matrix for nonlinear least-squares problem

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function
E04DGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library. In particular, the optional parameters of the NAG routine are now included in the parameter list. These are described in section 5.1.2, below.

1. Purpose

E04DGF minimizes an unconstrained nonlinear function of several variables using a pre-conditioned, limited memory quasi-Newton conjugate gradient method. First derivatives are required. The routine is intended for use on large scale problems.

2. Specification

```fortran
SUBROUTINE E04DGF(N,OBJFUN,ITER,OBJF,OBJGRD,X,IWORK,WORK,IUSER,
  1   USER,ES,FU,IT,LIN,LIST,MA,OP,PR,STA,STO,
  2   VE,IFAIL)
      INTEGER N, ITER, IWORK(N+1), IUSER(*),
      1 IT, PR, STA, STO, VE, IFAIL
      DOUBLE PRECISION OBJF, OBJGRD(N), X(N), WORK(13*N), USER(*)
      1 ES, FU, LIN, OP, MA
      LOGICAL LIST
      EXTERNAL OBJFUN
```

3. Description
EO4DGF uses a pre-conditioned conjugate gradient method and is based upon algorithm PLMA as described in Gill and Murray [1] and Gill et al [2] Section 4.8.3.

The algorithm proceeds as follows:

Let \( x \) be a given starting point and let \( k \) denote the current iteration, starting with \( k=0 \). The iteration requires \( g_k \), the gradient vector evaluated at \( x_k \), the \( k \)th estimate of the minimum.

At each iteration a vector \( p_k \) (known as the direction of search) is computed and the new estimate \( x_{k+1} \) is given by

\[
x_{k+1} = x_k + (\alpha_k) p_k
\]

where \( \alpha_k \) (the step length) minimizes the function

\[
F(x + (\alpha_k) p_k)
\]

with respect to the scalar \( \alpha_k \). A choice of initial step \( \alpha_0 \) is taken as

\[
\alpha_0 = \min\{1, 2(\frac{|F_{est} - F_k|}{g_k})\}
\]

where \( F_{est} \) is a user-supplied estimate of the function value at the solution. If \( F_{est} \) is not specified, the software always chooses the unit step length for \( \alpha_k \). Subsequent step length estimates are computed using cubic interpolation with safeguards.

A quasi-Newton method can be used to compute the search direction \( p_k \) by updating the inverse of the approximate Hessian \( H_k \) and computing

\[
p = -H_k^{-1} g_{k+1}
\]

The updating formula for the approximate inverse is given by

\[
H_{k+1} = H_k - \frac{\beta}{\gamma} (H_k y_k s + s y_k H_k) + \gamma \beta (1 + \frac{1}{\gamma}) s s
\]
where \( y = g - g \) and \( s = x - x = (\alpha) p \).

The method used by E04DGF to obtain the search direction is based upon computing \( p = -H g \) where \( H \) is a matrix obtained by updating the identity matrix with a limited number of quasi-Newton corrections. The storage of an \( n \times n \) matrix is avoided by storing only the vectors that define the rank two corrections—hence the term limited-memory quasi-Newton method. The precise method depends upon the number of updating vectors stored. For example, the direction obtained with the 'one-step' limited memory update is given by (1) using (2) with \( H \) equal to the identity matrix, viz.

\[
T \begin{pmatrix} g \\ \alpha \end{pmatrix} - T \begin{pmatrix} g \\ \alpha \end{pmatrix}
\]

\[
p = -g + \frac{1}{k+1} (s \cdot g + y \cdot g - s) - \frac{1}{k+1} (1+\frac{y}{s}) \cdot g
\]

\[
T \begin{pmatrix} g \\ \alpha \end{pmatrix} - T \begin{pmatrix} g \\ \alpha \end{pmatrix}
\]

E04DGF uses a two-step method described in detail in Gill and Murray [1] in which restarts and pre-conditioning are incorporated. Using a limited-memory quasi-Newton formula, such as the one above, guarantees \( p \) to be a descent direction if all the inner products \( y \) are positive for all vectors \( y \) and \( s \) used in the updating formula.

The termination criterion of E04DGF is as follows:

Let \( (\tau) \) specify a parameter that indicates the number of correct figures desired in \( F \) \((\tau) \) is equivalent to Optimality Tolerance in the optional parameter list, see Section 5.1). If the following three conditions are satisfied

\[
(i) \quad F - F < (\tau) (1+|F|)
\]

\[
k-1 \quad F \quad k
\]

\[
------
\]
(ii) \( ||x - x^k|| < \frac{\tau}{1 + ||x||} \)

(iii) \( ||g|| \leq 3 \frac{\tau}{1 + |F|} \) or \( ||g|| < \epsilon \),

where \( \epsilon \) is the absolute error associated with computing the objective function

then the algorithm is considered to have converged. For a full discussion on termination criteria see Gill et al [2] Chapter 8.

4. References


5. Parameters

1: \( N \) -- INTEGER Input

   On entry: the number \( n \) of variables. Constraint: \( N \geq 1 \).

2: OBJFUN -- SUBROUTINE, supplied by the user.

   External Procedure

   OBJFUN must calculate the objective function \( F(x) \) and its gradient for a specified \( n \) element vector \( x \).

   Its specification is:

   ```
   SUBROUTINE OBJFUN (MODE, N, X, OBJF, OBJGRD,
   1 NSTATE, IUSER, USER)
   INTEGER MODE, N, NSTATE, IUSER(*)
   DOUBLE PRECISION X(N), OBJF, OBJGRD(N), USER(*)
   ```

1: MODE -- INTEGER Input/Output

   MODE is a flag that the user may set within OBJFUN to indicate a failure in the evaluation of the objective function. On entry: MODE is always non-negative. On exit: if MODE is negative the execution of E04DGF is terminated with IFAIL set to MODE.

2: \( N \) -- INTEGER Input

   On entry: the number \( n \) of variables.

3: \( X(N) \) -- DOUBLE PRECISION array Input
On entry: the point x at which the objective function is required.

4: OBJF -- DOUBLE PRECISION Output
On exit: the value of the objective function F at the current point x.

5: OBJGRD(N) -- DOUBLE PRECISION array Output
ddF
On exit: OBJGRD(i) must contain the value of ---- at ddx
          the point x, for i=1,2,...,n.

6: NSTATE -- INTEGER Input
On entry: NSTATE will be 1 on the first call of OBJFUN by E04DGF, and is 0 for all subsequent calls. Thus, if the user wishes, NSTATE may be tested within OBJFUN in order to perform certain calculations once only. For example the user may read data or initialise COMMON blocks when NSTATE = 1.

7: IUSER(*) -- INTEGER array User Workspace

8: USER(*) -- DOUBLE PRECISION array User Workspace
OBJFUN is called from E04DGF with the parameters IUSER and USER as supplied to E04DGF. The user is free to use arrays IUSER and USER to supply information to OBJFUN as an alternative to using COMMON.

OBJFUN must be declared as EXTERNAL in the (sub)program from which E04DGF is called. Parameters denoted as Input must not be changed by this procedure.

3: ITER -- INTEGER Output
On exit: the number of iterations performed.

4: OBJF -- DOUBLE PRECISION Output
On exit: the value of the objective function F(x) at the final iterate.

5: OBJGRD(N) -- DOUBLE PRECISION array Output
On exit: the objective gradient at the final iterate.

6: X(N) -- DOUBLE PRECISION array Input/Output
On entry: an initial estimate of the solution. On exit: the final estimate of the solution.

7: IWORK(N+1) -- INTEGER array Workspace

8: WORK(13*N) -- DOUBLE PRECISION array Workspace
9: IUSER(*) -- INTEGER array User Workspace
Note: the dimension of the array IUSER must be at least 1.
This array is not used by E04DGF, but is passed directly to
routine OBJFUN and may be used to supply information to
OBJFUN.

10: USER(*) -- DOUBLE PRECISION array User Workspace
Note: the dimension of the array USER must be at least 1.
This array is not used by E04DGF, but is passed directly to
routine OBJFUN and may be used to supply information to
OBJFUN.

11: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are
unfamiliar with this parameter should refer to the Essential
Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or
gives a warning (see Section 6).

For this routine, because the values of output parameters
may be useful even if IFAIL /=0 on exit, users are
recommended to set IFAIL to -1 before entry. It is then
essential to test the value of IFAIL on exit.

5.1. Optional Input Parameters

Several optional parameters in E04DGF define choices in the
behaviour of the routine. In order to reduce the number of formal
parameters of E04DGF these optional parameters have associated
default values (see Section 5.1.3) that are appropriate for most
problems. Therefore the user need only specify those optional
parameters whose values are to be different from their default
values.

The remainder of this section can be skipped by users who wish to
use the default values for all optional parameters. A complete
list of optional parameters and their default values is given in
Section 5.1.3.

5.1.1. Specification of the Optional Parameters

Optional parameters may be specified by calling one, or both, of
E04DJF and E04DKF prior to a call to E04DGF.

E04DJF reads options from an external options file, with Begin
and End as the first and last lines respectively and each
intermediate line defining a single optional parameter. For
example,

    Begin
    Print Level = 1
    End

The call

    CALL E04DJF(IOPTNS, INFORM)

can then be used to read the file on unit IOPTNS. INFORM will be zero on successful exit. E04DJF should be consulted for a full description of this method of supplying optional parameters.

E04DKF can be called to supply options directly, one call being necessary for each optional parameter.

For example,

    CALL E04DKF('Print level = 1')

E04DKF should be consulted for a full description of this method of supplying optional parameters.

All optional parameters not specified by the user are set to their default values. Optional parameters specified by the user are unaltered by E04DGF (unless they define invalid values) and so remain in effect for subsequent calls to E04DGF, unless altered by the user.

5.1.2. Description of the Optional Parameters

The following list (in alphabetical order) gives the valid options. For each option, we give the keyword, any essential optional qualifiers, the default value, and the definition. The minimum valid abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote INTEGER and real values required with certain options. The number (epsilon) is a generic notation for machine precision, and (epsilon) denotes the relative precision of the objective function (the optional parameter Function Precision; see below).

Defaults

This special keyword may be used to reset the default values
following a call to E04DGF.

Estimated Optimal Function Value \( r \)

(Axiom parameter ES)

This value of \( r \) specifies the user-supplied guess of the optimum objective function value. This value is used by E04DGF to calculate an initial step length (see Section 3). If the value of \( r \) is not specified by the user (the default), then this has the effect of setting the initial step length to unity. It should be noted that for badly scaled functions a unit step along the steepest descent direction will often compute the function at very large values of \( x \).

Function Precision \( r \) Default = (epsilon)

(Axiom parameter FU)

The parameter defines (epsilon), which is intended to be a measure of the accuracy with which the problem function \( F \) can be computed. The value of (epsilon) should reflect the relative precision of \( 1 + |F(x)| \); i.e. (epsilon) acts as a relative precision when \(|F| \) is large, and as an absolute precision when \(|F| \) is small. For example, if \( F(x) \) is typically of order 1000 and the first six significant digits are known to be correct, an appropriate value for (epsilon) would be 1.0E-6. In contrast, if \( F(x) \) is typically of order 10 and the first six significant digits are known to be correct, an appropriate value for (epsilon) would be 1.0E-10. The choice of (epsilon) can be quite complicated for badly scaled problems; see Chapter 8 of Gill and Murray [2], for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However when the accuracy of the computed function values is known to be significantly worse than full precision, the value of (epsilon) should be large enough so that E04DGF will not attempt to distinguish between function values that differ by less than the error inherent in the calculation. If 0 <= r <= (epsilon), where (epsilon) is the machine precision then the default value is used.

Iteration Limit \( i \) Default = max(50, 5n)
Iters
Itns

(Axiom parameter IT)

The value $i$ ($i \geq 0$) specifies the maximum number of iterations allowed before termination. If $i < 0$ the default value is used. See Section 8 for further information.

Linesearch Tolerance $r$ Default = 0.9

(Axiom parameter LIN)

The value $r$ ($0 \leq r < 1$) controls the accuracy with which the step ($\alpha$) taken during each iteration approximates a minimum of the function along the search direction (the smaller the value of $r$, the more accurate the linesearch). The default value $r=0.9$ requests an inaccurate search, and is appropriate for most problems. A more accurate search may be appropriate when it is desirable to reduce the number of iterations - for example, if the objective function is cheap to evaluate.

List Default = List
Nolist

(Axiom parameter LIST)

Normally each optional parameter specification is printed as it is supplied. Nolist may be used to suppress the printing and List may be used to restore printing.

Maximum Step Length $r$ Default = 10

(Axiom parameter MA)

The value $r$ ($r > 0$) defines the maximum allowable step length for the line search. If $r \leq 0$ the default value is used.

Optimality Tolerance $r$ Default = $\epsilon$

(Axiom parameter OP)

$\epsilon$

The parameter $r$ ($\epsilon \leq r < 1$) specifies the accuracy to which the user wishes the final iterate to approximate a solution of the problem. Broadly speaking, $r$ indicates the number of correct
figures desired in the objective function at the solution. For example, if \( r \) is 10 and E04DGF terminates successfully, the final value of \( F \) should have approximately six correct figures. E04DGF will terminate successfully if the iterative sequence of \( x \)-values is judged to have converged and the final point satisfies the termination criteria (see Section 3, where (tau) represents \( F \) Optimality Tolerance).

Print Level \( i \) Default = 10

(Axiom parameter PR)

The value \( i \) controls the amount of printout produced by E04DGF. The following levels of printing are available.

- \( i = 0 \) No output.
- \( i = 1 \) The final solution.
- \( i = 5 \) One line of output for each iteration.
- \( i = 10 \) The final solution and one line of output for each iteration.

Start Objective Check at Variable \( i \) Default = 1

(Axiom parameter STA)

Stop Objective Check at Variable \( i \) Default = \( n \)

(Axiom parameter STO)

These keywords take effect only if Verify Level > 0 (see below). They may be used to control the verification of gradient elements computed by subroutine OBJFUN. For example if the first 30 variables appear linearly in the objective, so that the corresponding gradient elements are constant, then it is reasonable to specify Start Objective Check at Variable 31.

Verify Level \( i \) Default = 0

Verify No

Verify Level -1

Verify Level 0
Verify

Verify

Verify Yes

Verify Objective Gradients

Verify Gradients

Verify Level 1

(Axiom parameter VE)

These keywords refer to finite-difference checks on the gradient elements computed by the user-provided subroutine \texttt{OBJFUN}. It is possible to set Verify Level in several ways, as indicated above. For example, the gradients will be verified if Verify, Verify Yes, Verify Gradients, Verify Objective Gradients or Verify Level = 1 is specified.

If i<0 then no checking will be performed. If i>0 then the gradients will be verified at the user-supplied point. If i=0 only a 'cheap' test will be performed, requiring one call to \texttt{OBJFUN}. If i=1, a more reliable (but more expensive) check will be made on individual gradient components, within the ranges specified by the Start and Stop keywords as described above. A result of the form OK or BAD? is printed by \texttt{E04DGF} to indicate whether or not each component appears to be correct.

5.1.3. Optional parameter checklist and default values

For easy reference, the following sample list shows all valid keywords and their default values. The default options Function Precision and Optimality Tolerance depend upon (epsilon), the machine precision.

<table>
<thead>
<tr>
<th>Optional Parameters</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated Optimal Function Value</td>
<td>0.9</td>
</tr>
<tr>
<td>Function precision (epsilon)</td>
<td>max(50,5n)</td>
</tr>
<tr>
<td>Linesearch Tolerance</td>
<td>0.9</td>
</tr>
<tr>
<td>Maximum Step Length</td>
<td>10</td>
</tr>
</tbody>
</table>
List/Nolist List

Optimality Tolerance (epsilon) 0.8

Print Level 10

Start Objective Check at Variable 1

Stop Objective Check at Variable n

Verify Level 0

5.2. Description of Printed Output

The level of printed output from E04DGF is controlled by the user (see the description of Print Level in Section 5.1).

When Print Level >= 5, the following line of output is produced at each iteration.

Itn is the iteration count.

Step is the step (alpha) taken along the computed search direction. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.

Nfun is the cumulated number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the gradients by finite differences are not included. Nfun is printed as a guide to the amount of work required for the linesearch. E04DGF will perform at most 16 function evaluations per iteration.

Objective is the value of the objective function.

Norm G is the Euclidean norm of the gradient of the objective function.

Norm X is the Euclidean norm of x.

Norm \((x(k-1)-x(k))\) is the Euclidean norm of \(x_{k-1} - x_k\).

When Print Level = 1 or Print Level >= 10 then the solution at
the end of execution of E04DGF is printed out.

The following describes the printout for each variable:

Variable gives the name (VARBL) and index j (j = 1 to n) of the variable

Value is the value of the variable at the final iterate

Gradient Value is the value of the gradient of the objective function with respect to the jth variable at the final iterate

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

On exit from E04DGF, IFAIL should be tested. If Print Level > 0 then a short description of IFAIL is printed.

Errors and diagnostics indicated by IFAIL from E04DGF are as follows:

IFAIL< 0
A negative value of IFAIL indicates an exit from E04DGF because the user set MODE negative in routine OBJFUN. The value of IFAIL will be the same as the user's setting of MODE.

IFAIL= 1
Not used by this routine.

IFAIL= 2
Not used by this routine.

IFAIL= 3
The maximum number of iterations has been performed. If the algorithm appears to be making progress the iterations value may be too small (see Section 5.1.2) so the user should increase iterations and rerun E04DGF. If the algorithm seems to be 'bogged down', the user should check for incorrect gradients or ill-conditioning as described below under IFAIL = 6.

IFAIL= 4
The computed upper bound on the step length taken during the linesearch was too small. A rerun with an increased value of
the Maximum Step Length (\(\rho\) say) may be successful unless
\[10 \quad (\rho) \geq 10 \quad \text{(the default value), in which case the current point cannot be improved upon.}\]

**IFAIL= 5**
Not used by this routine.

**IFAIL= 6**
A sufficient decrease in the function value could not be attained during the final linesearch. If the subroutine `OBJFUN` computes the function and gradients correctly, then this may occur because an overly stringent accuracy has been requested, i.e., Optimality Tolerance is too small or if the minimum lies close to a step length of zero. In this case the user should apply the four tests described in Section 3 to determine whether or not the final solution is acceptable (the user will need to set Print Level \(\geq 5\)). For a discussion of attainable accuracy see Gill and Murray [2].

If many iterations have occurred in which essentially no progress has been made or `E04DGF` has failed to move from the initial point, subroutine `OBJFUN` may be incorrect. The user should refer to the comments below under IFAIL = 7 and check the gradients using the Verify parameter. Unfortunately, there may be small errors in the objective gradients that cannot be detected by the verification process. Finite-difference approximations to first derivatives are catastrophically affected by even small inaccuracies.

**IFAIL= 7**
Large errors were found in the derivatives of the objective function. This value of IFAIL will occur if the verification process indicated that at least one gradient component had no correct figures. The user should refer to the printed output to determine which elements are suspected to be in error.

As a first step, the user should check that the code for the objective values is correct - for example, by computing the function at a point where the correct value is known. However, care should be taken that the chosen point fully tests the evaluation of the function. It is remarkable how often the values \(x=0\) or \(x=1\) are used to test function evaluation procedures, and how often the special properties of these numbers make the test meaningless.

Special care should be used in this test if computation of the objective function involves subsidiary data communicated in COMMON storage. Although the first evaluation of the
function may be correct, subsequent calculations may be in error because some of the subsidiary data has accidentally been overwritten.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single-precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

IFAIL= 8
The gradient (g) at the starting point is too small. The T value g g is less than (epsilon) |F(x)|, where (epsilon) m o m is the machine precision.

The problem should be rerun at a different starting point.

IFAIL= 9
On entry N < 1.

7. Accuracy

On successful exit the accuracy of the solution will be as defined by the optional parameter Optimality Tolerance.

8. Further Comments

Problems whose Hessian matrices at the solution contain sets of clustered eigenvalues are likely to be minimized in significantly fewer than n iterations. Problems without this property may require anything between n and 5n iterations, with approximately 2n iterations being a common figure for moderately difficult problems.

9. Example

To find a minimum of the function

\[ F(x) = e^{(4x_1 + 2x_2 + 4x_3 + 2x_4 + 1)}. \]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E04 -- Minimizing or Maximizing a Function
E04DJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

To supply optional parameters to E04DGF from an external file.

2. Specification

SUBROUTINE E04DJF (IOPTNS, INFORM)
INTEGER IOPTNS, INFORM

3. Description

E04DJF may be used to supply values for optional parameters to E04DGF. E04DJF reads an external file and each line of the file defines a single optional parameter. It is only necessary to supply values for those parameters whose values are to be different from their default values.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

Print level = 1

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran
77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

The file containing the options must start with begin and must finish with end. An example of a valid options file is:

```
Begin * Example options file
  Print level = 10
End
```

Normally each line of the file is printed as it is read, on the current advisory message unit (see X04ABF), but printing may be suppressed using the keyword nolist. To suppress printing of begin, nolist must be the first option supplied as in the file:

```
Begin
  Nolist
  Print level = 10
End
```

Printing will automatically be turned on again after a call to E04DGF and may be turned on again at any time by the user by using the keyword list.

Optional parameter settings are preserved following a call to E04DGF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values prior to a subsequent call to E04DGF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the routine document for E04DGF.

4. References

None.

5. Parameters

1: IOPTNS -- INTEGER
   Input
   On entry: IOPTNS must be the unit number of the options file. Constraint: $0 \leq IOPTNS \leq 99$.
   
2: INFORM -- INTEGER
   Output
   On exit: INFORM will be zero if an options file with the
correct structure has been read. Otherwise INFORM will be
positive. Positive values of INFORM indicate that an options
file may not have been successfully read as follows:
INFORM = 1
    IOPTNS is not in the range \([0,99]\).

INFORM = 2
    begin was found, but end-of-file was found before end
    was found.

INFORM = 3
    end-of-file was found before begin was found.

6. Error Indicators and Warnings

If a line is not recognised as a valid option, then a warning
message is output on the current advisory message unit (see
X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04DKF may also be used to supply optional parameters to E04DGF.

9. Example

See the example for E04DGF.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function
E04DGF
E04DKF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

To supply individual optional parameters to E04DGF.

2. Specification

SUBROUTINE E04DKF (STRING)
    CHARACTER*(*) STRING
3. Description

E04DKF may be used to supply values for optional parameters to E04DGF. It is only necessary to call E04DKF for those parameters whose values are to be different from their default values. One call to E04DKF sets one parameter value.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

   Print Level = 1

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77’s I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

Normally, each user-specified option is printed as it is defined, on the current advisory message unit (see X04ABF), but this printing may be suppressed using the keyword nolist Thus the statement

   CALL E04DKF (‘Nolist’)

suppresses printing of this and subsequent options. Printing will automatically be turned on again after a call to E04DGF, and may be turned on again at any time by the user, by using the keyword list.

Optional parameter settings are preserved following a call to E04DGF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values by the statement,

   CALL E04DKF (‘Defaults’)

prior to a subsequent call to E04DGF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the routine document for E04DGF.

4. References

None.

5. Parameters

1: STRING -- CHARACTER(*) Input
   On entry: STRING must be a single valid option string. See Section 3 above, and Section 5.1 of the routine document for E04DGF.

6. Error Indicators and Warnings

If the parameter STRING is not recognised as a valid option string, then a warning message is output on the current advisory message unit (see X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04DJF may also be used to supply optional parameters to E04DGF.

9. Example

See the example for E04DGF.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E04 -- Minimizing or Maximizing a Function
E04FDF
E04FDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04FDF is an easy-to-use algorithm for finding an unconstrained minimum of a sum of squares of m nonlinear functions in n
variables (m >= n). No derivatives are required.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

```fortran
SUBROUTINE E04FDF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
INTEGER M, N, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION X(N), FSUMSQ, W(LW)
```

3. Description

This routine is essentially identical to the subroutine LSNDN1 in the National Physical Laboratory Algorithms Library. It is applicable to problems of the form

\[
\min_{x} F(x) = \sum_{i=1}^{m} f_i(x)
\]

where \( x = (x_1, x_2, \ldots, x_n) \) and \( m \geq n \). (The functions \( f_i(x) \) are often referred to as 'residuals'.) The user must supply a subroutine LSFUN1 to evaluate functions \( f_i(x) \) at any point \( x \).

From a starting point supplied by the user, a sequence of points is generated which is intended to converge to a local minimum of the sum of squares. These points are generated using estimates of the curvature of \( F(x) \).

4. References


5. Parameters

1: M -- INTEGER Input

2: N -- INTEGER Input

On entry: the number m of residuals \( f_i(x) \), and the number n of variables, \( x \). Constraint: 1 <= N <= M.
3: X(N) -- DOUBLE PRECISION array  
   Input/Output  
   On entry: X(j) must be set to a guess at the jth component  
   of the position of the minimum, for j=1,2,...,N.  
   On exit:  
   the lowest point found during the calculations.  
   Thus, if  
   IFAIL = 0 on exit, X(j) is the jth component of the position  
   of the minimum.  

4: FSUMSQ -- DOUBLE PRECISION  
   Output  
   On exit: the value of the sum of squares, F(x),  
   corresponding to the final point stored in X.  

5: IW(LIW) -- INTEGER array  
   Workspace  

6: LIW -- INTEGER  
   Input  
   On entry: the length of IW as declared in the (sub)program  
   from which E04FDF has been called.  
   Constraint: LIW >= 1.  

7: W(LW) -- DOUBLE PRECISION array  
   Workspace  

8: LW -- INTEGER  
   Input  
   On entry: the length of W as declared in the (sub)program  
   from which E04FDF is called.  
   Constraints:  
   LW >= N*(7 + N + 2*M + (N-1)/2) + 3*M, if N > 1,  
   LW >= 9 + 5*M, if N = 1.  

9: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1.  
   Users who are  
   unfamiliar with this parameter should refer to the Essential  
   Introduction for details.  

   On exit: IFAIL = 0 unless the routine detects an error or  
   gives a warning (see Section 6).  

   For this routine, because the values of output parameters  
   may be useful even if IFAIL /=0 on exit, users are  
   recommended to set IFAIL to -1 before entry.  
   It is then  
   essential to test the value of IFAIL on exit.  

5.1. Optional Parameters  

LSFUN1 -- SUBROUTINE, supplied by the user.  
   External Procedure  
   This routine must be supplied by the user to calculate the  
   vector of values f(x) at any point x.  
   Since the routine is  
   not a parameter to E04FDF, it must be called LSFUN1.  
   It  
   should be tested separately before being used in conjunction
with E04FDF (see the Chapter Introduction).

Its specification is:

```fortran
SUBROUTINE LSFUN1 (M, N, XC, FVECC)
INTEGER M, N
DOUBLE PRECISION XC(N), FVECC(M)
```

1: M -- INTEGER  
   Input
2: N -- INTEGER  
   Input
   On entry: the numbers m and n of residuals and
   variables, respectively.
3: XC(N) -- DOUBLE PRECISION array  
   Input
   On entry: the point x at which the values of the f
   \( f_i \) are required.
4: FVECC(M) -- DOUBLE PRECISION array  
   Output
   On exit: FVECC(i) must contain the value of \( f_i \) at the
   point x, for \( i=1,2,\ldots,m \).

LSFUN1 must be declared as EXTERNAL in the (sub)program
from which E04FDF is called. Parameters denoted as
Input must not be changed by this procedure.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry N < 1,
   or M < N,
   or LIW < 1,
   or LW < N*(7 + N + 2*M + (N-1)/2) + 3*M, when N > 1,
   or LW < 9 + 5*M, when N = 1.

IFAIL= 2
   There have been 400*n calls of LSFUN1, yet the algorithm
does not seem to have converged. This may be due to an
awkward function or to a poor starting point, so it is worth
restarting E04FDF from the final point held in X.
IFAIL = 3
The final point does not satisfy the conditions for acceptance as a minimum, but no lower point could be found.

IFAIL = 4
An auxiliary routine has been unable to complete a singular value decomposition in a reasonable number of sub-iterations.

IFAIL = 5

IFAIL = 6

IFAIL = 7

IFAIL = 8
There is some doubt about whether the point x found by E04FDF is a minimum of F(x). The degree of confidence in the result decreases as IFAIL increases. Thus when IFAIL = 5, it is probable that the final x gives a good estimate of the position of a minimum, but when IFAIL = 8 it is very unlikely that the routine has found a minimum.

If the user is not satisfied with the result (e.g. because IFAIL lies between 3 and 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. Repeated failure may indicate some defect in the formulation of the problem.

7. Accuracy

If the problem is reasonably well scaled and a successful exit is made, then, for a computer with a mantissa of t decimals, one would expect to get about t/2-1 decimals accuracy in the components of x and between t-1 (if F(x) is of order 1 at the minimum) and 2t-2 (if F(x) is close to zero at the minimum) decimals accuracy in F(x).

8. Further Comments

The number of iterations required depends on the number of variables, the number of residuals and their behaviour, and the distance of the starting point from the solution. The number of multiplications performed per iteration of E04FDF varies, but for 2 3
m >> n is approximately n*m +O(n ). In addition, each iteration makes at least n+1 calls of LSFUN1. So, unless the residuals can be evaluated very quickly, the run time will be dominated by the
time spent in LSFUN1.

Ideally, the problem should be scaled so that the minimum value of the sum of squares is in the range (0,1), and so that at points a unit distance away from the solution the sum of squares is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04FDF will take less computer time.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance-covariance matrix of the estimated regression coefficients can be computed by a subsequent call to E04YCF, using information returned in segments of the workspace array W. See E04YCF for further details.

9. Example

To find least-squares estimates of $x_1$, $x_2$ and $x_3$ in the model

$$y = x_1 + \frac{1}{x_1 x_2 + x_3}$$

using the 15 sets of data given in the following table.

<table>
<thead>
<tr>
<th>y</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.14</td>
<td>1.0</td>
<td>15.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.18</td>
<td>2.0</td>
<td>14.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0.22</td>
<td>3.0</td>
<td>13.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0.25</td>
<td>4.0</td>
<td>12.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.29</td>
<td>5.0</td>
<td>11.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.32</td>
<td>6.0</td>
<td>10.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.35</td>
<td>7.0</td>
<td>9.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.39</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>0.37</td>
<td>9.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.58</td>
<td>10.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.73</td>
<td>11.0</td>
<td>5.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.96</td>
<td>12.0</td>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>1.34</td>
<td>13.0</td>
<td>3.0</td>
<td>3.0</td>
</tr>
<tr>
<td>2.10</td>
<td>14.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>4.39</td>
<td>15.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>
The program uses \((0.5, 1.0, 1.5)\) as the initial guess at the position of the minimum.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

\begin{verbatim}
E04 -- Minimizing or Maximizing a Function
E04GCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04GCF is an easy-to-use quasi-Newton algorithm for finding an unconstrained minimum of a sum of squares of \(m\) nonlinear functions in \(n\) variables (\(m \geq n\)). First derivatives are required.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

\begin{verbatim}
SUBROUTINE E04GCF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
INTEGER M, N, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION X(N), FSUMSQ, W(LW)
\end{verbatim}

3. Description

This routine is essentially identical to the subroutine LSFDQ2 in the National Physical Laboratory Algorithms Library. It is applicable to problems of the form

\[
\text{minimize } \sum_{i=1}^{m} f_i(x) \quad (m \geq n)
\]

where \(x=(x_1, x_2, \ldots, x_n)\) and \(m \geq n\). (The functions \(f_i(x)\) are often referred to as 'residuals'.) The user must supply a subroutine LSFUN2 to evaluate the residuals and their first derivatives at
any point $x$.

Before attempting to minimize the sum of squares, the algorithm
checks LSFUN2 for consistency. Then, from a starting point
supplied by the user, a sequence of points is generated which is
intended to converge to a local minimum of the sum of squares.
These points are generated using estimates of the curvature of
$F(x)$.

4. References

the Nonlinear Least-squares Problem. SIAM J. Numer. Anal. 15
977--992.

5. Parameters

1: M -- INTEGER Input

2: N -- INTEGER Input
   On entry: the number $m$ of residuals $f(x)$, and the number $n$
   $i$
of variables, $x$. Constraint: $1 \leq N \leq M$.
   $j$

3: X(N) -- DOUBLE PRECISION array Input/Output
   On entry: $X(j)$ must be set to a guess at the $j$th component
   of the position of the minimum, for $j=1,2,...,n$. The routine
   checks the first derivatives calculated by LSFUN2 at the
   starting point, and so is more likely to detect an error in
   the user's routine if the initial $X(j)$ are non-zero and
   mutually distinct. On exit: the lowest point found during
   the calculations. Thus, if IFAIL = 0 on exit, $X(j)$ is the $j$
th component of the position of the minimum.

4: FSUMSQ -- DOUBLE PRECISION Output
   On exit: the value of the sum of squares, $F(x)$,
corresponding to the final point stored in X.

5: IW(LIW) -- INTEGER array Workspace

6: LIW -- INTEGER Input
   On entry: the length of IW as declared in the (sub)program
   from which E04GCF is called. Constraint: LIW $\geq 1$.

7: W(LW) -- DOUBLE PRECISION array Workspace

8: LW -- INTEGER Input
   On entry: the length of W as declared in the (sub)program
   from which E04GCF is called. Constraints:
\\L W & \geq & 2N(4 + N + M) + 3M, \text{ if } N > 1, \\
\L W & \geq & 11 + 5M, \text{ if } N = 1.

9: \textbf{IFAIL} -- INTEGER \quad \text{Input/Output} \\
On entry: IFAIL must be set to 0, -1 or 1. Users who are 
unfamiliar with this parameter should refer to the Essential 
Introduction for details. 

On exit: IFAIL = 0 unless the routine detects an error or 
gives a warning (see Section 6). 

For this routine, because the values of output parameters 
may be useful even if IFAIL /=0 on exit, users are 
recommended to set IFAIL to -1 before entry. It is then 
essential to test the value of IFAIL on exit.

5.1. Optional Parameters 

\textbf{LSFUN2} -- SUBROUTINE, supplied by the user. 

\textbf{External Procedure} 

This routine must be supplied by the user to calculate the 
vector of values \( f(x) \) and the Jacobian matrix of first 
\( i \) 
\( \frac{df}{dx} \) 
\( i \) 
\( j \) 
derivatives \quad \text{at any point } x. \text{ Since the routine is not a} 
\( \frac{d}{dx} \) 
\( j \) 
\text{parameter to E04GCF, it must be called LSFUN2. It should be} 
tested separately before being used in conjunction with 
E04GCF (see the Chapter Introduction). 

Its specification is: 

\textbf{SUBROUTINE LSFUN2} \( (M, N, XC, FVECC, FJACC, LJC) \) 
\textbf{INTEGER} \( M, N, LJC \) 
\textbf{DOUBLE PRECISION} \( XC(N), FVECC(M), FJACC(LJC,N) \) 

Important: The dimension declaration for FJACC must 
contain the variable LJC, not an integer constant.

1: \textbf{M} -- INTEGER \quad \text{Input} 

2: \textbf{N} -- INTEGER \quad \text{Input} 

\text{On entry: the numbers } m \text{ and } n \text{ of residuals and} 
\text{variables, respectively.} 

3: \textbf{XC(N)} -- DOUBLE PRECISION array \quad \text{Input} 

\text{On entry: the point } x \text{ at which the values of the } f 
\( i \)
and the \( \frac{\partial}{\partial x} \) are required.

4: FVECC(M) -- DOUBLE PRECISION array  
   Output  
   On exit: FVECC(i) must contain the value of \( f \) at the  
   point \( x \), for \( i=1,2,\ldots,m \).

5: FJACC(LJC,N) -- DOUBLE PRECISION array  
   Output  
   On exit: FJACC(i,j) must contain the value of \( \frac{\partial^2 f}{\partial x_i \partial x_j} \) at  
   the point \( x \), for \( i=1,2,\ldots,m; j=1,2,\ldots,n \).

6: LJC -- INTEGER  
   Input  
   On entry: the first dimension of the array FJACC.


6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are  
output on the current error message unit (as defined by X04AAF).

IFAIL= 1  
On entry \( N < 1 \),  
or \( M < N \),  
or \( LIW < 1 \),  
or \( LW < 2*N*(4 + N + M) + 3*M \), when \( N > 1 \),  
or \( LW < 9 + 5*M \), when \( N = 1 \).

IFAIL= 2  
There have been 50\( n \) calls of LSFUN2, yet the algorithm does  
not seem to have converged. This may be due to an awkward  
function or to a poor starting point, so it is worth  
restarting E04GCF from the final point held in \( X \).

IFAIL= 3
The final point does not satisfy the conditions for acceptance as a minimum, but no lower point could be found.

IFAIL = 4
An auxiliary routine has been unable to complete a singular value decomposition in a reasonable number of sub-iterations.

IFAIL = 5

IFAIL = 6

IFAIL = 7

IFAIL = 8
There is some doubt about whether the point X found by E04GCF is a minimum of F(x). The degree of confidence in the result decreases as IFAIL increases. Thus, when IFAIL = 5, it is probable that the final X gives a good estimate of the position of a minimum, but when IFAIL = 8 it is very unlikely that the routine has found a minimum.

IFAIL = 9
It is very likely that the user has made an error in forming
\[ \frac{\partial f}{\partial x} \]
the derivatives \( \frac{\partial f}{\partial x} \) in LSFUN2.

If the user is not satisfied with the result (e.g. because IFAIL lies between 3 and 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. Repeated failure may indicate some defect in the formulation of the problem.

7. Accuracy

If the problem is reasonably well scaled and a successful exit is made then, for a computer with a mantissa of \( t \) decimals, one would expect to get \( t/2-1 \) decimals accuracy in the components of \( x \) and between \( t-1 \) (if \( F(x) \) is of order 1 at the minimum) and \( 2t-2 \) (if \( F(x) \) is close to zero at the minimum) decimals accuracy in \( F(x) \).

8. Further Comments

The number of iterations required depends on the number of variables, the number of residuals and their behaviour, and the
distance of the starting point from the solution. The number of multiplications performed per iteration of E04GCF varies, but for $m \gg n$ is approximately $n^m + O(n)$. In addition, each iteration makes at least one call of LSFUN2. So, unless the residuals and their derivatives can be evaluated very quickly, the run time will be dominated by the time spent in LSFUN2.

Ideally the problem should be scaled so that the minimum value of the sum of squares is in the range $(0,1)$ and so that at points a unit distance away from the solution the sum of squares is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04GCF will take less computer time.

When the sum of squares represents the goodness of fit of a nonlinear model to observed data, elements of the variance-covariance matrix of the estimated regression coefficients can be computed by a subsequent call to E04YCF, using information returned in segments of the workspace array $W$. See E04YCF for further details.

9. Example

To find the least-squares estimates of $x_1$, $x_2$ and $x_3$ in the model

$$
y = x_1 + \frac{x_2}{x_1 + x_3}
$$

using the 15 sets of data given in the following table.

<table>
<thead>
<tr>
<th>$y$</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.14</td>
<td>1.0</td>
<td>15.0</td>
<td>1.0</td>
</tr>
<tr>
<td>0.18</td>
<td>2.0</td>
<td>14.0</td>
<td>2.0</td>
</tr>
<tr>
<td>0.22</td>
<td>3.0</td>
<td>13.0</td>
<td>3.0</td>
</tr>
<tr>
<td>0.25</td>
<td>4.0</td>
<td>12.0</td>
<td>4.0</td>
</tr>
<tr>
<td>0.29</td>
<td>5.0</td>
<td>11.0</td>
<td>5.0</td>
</tr>
<tr>
<td>0.32</td>
<td>6.0</td>
<td>10.0</td>
<td>6.0</td>
</tr>
<tr>
<td>0.35</td>
<td>7.0</td>
<td>9.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.39</td>
<td>8.0</td>
<td>8.0</td>
<td>8.0</td>
</tr>
<tr>
<td>0.37</td>
<td>9.0</td>
<td>7.0</td>
<td>7.0</td>
</tr>
<tr>
<td>0.58</td>
<td>10.0</td>
<td>6.0</td>
<td>6.0</td>
</tr>
</tbody>
</table>
The program uses (0.5, 1.0, 1.5) as the initial guess at the position of the minimum.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

E04 -- Minimizing or Maximizing a Function
E04JAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04JAF is an easy-to-use quasi-Newton algorithm for finding a minimum of a function \( F(x_1, x_2, \ldots, x_n) \), subject to fixed upper and lower bounds of the independent variables \( x_1, x_2, \ldots, x_n \), using function values only.

It is intended for functions which are continuous and which have continuous first and second derivatives (although it will usually work even if the derivatives have occasional discontinuities).

2. Specification

```fortran
SUBROUTINE E04JAF (N, IBOUND, BL, BU, X, F, IW, LIW, W, LW, IFAIL)
INTEGER N, IBOUND, IW(LIW), LIW, LW, IFAIL
DOUBLE PRECISION BL(N), BU(N), X(N), F, W(LW)
```

3. Description

This routine is applicable to problems of the form:

\[
\text{Minimize } F(x_1, x_2, \ldots, x_n) \text{ subject to } l_j \leq x_j \leq u_j, \quad j=1, 2, \ldots, n
\]
when derivatives of \( F(x) \) are unavailable.

Special provision is made for problems which actually have no bounds on the \( x \), problems which have only non-negativity bounds and problems in which \( l = l \ldots = l \) and \( u = u \ldots = u \). The user must supply a subroutine \( \text{FUNCT1} \) to calculate the value of \( F(x) \) at any point \( x \).

From a starting point supplied by the user there is generated, on the basis of estimates of the gradient and the curvature of \( F(x) \), a sequence of feasible points which is intended to converge to a local minimum of the constrained function. An attempt is made to verify that the final point is a minimum.

4. References


5. Parameters

1: \( N \) -- INTEGER Input
   On entry: the number \( n \) of independent variables.
   Constraint: \( N \geq 1 \).

2: \( IBOUND \) -- INTEGER Input
   On entry: indicates whether the facility for dealing with bounds of special forms is to be used.
   It must be set to one of the following values:
   \( IBOUND = 0 \)
   if the user will be supplying all the \( l \) and \( u \) \( j \) \( j \) individually.
   \( IBOUND = 1 \)
   if there are no bounds on any \( x \).
   \( j \)
   \( IBOUND = 2 \)
   if all the bounds are of the form \( 0 \leq x \).
   \( j \)
   \( IBOUND = 3 \)
   if \( l = l \ldots = l \) and \( u = u \ldots = u \).
3: BL(N) -- DOUBLE PRECISION array
   On entry: the lower bounds \( l_j \).

If IBOUND is set to 0, the user must set \( BL(j) \) to \( l_j \), for \( j=1,2,...,n \). (If a lower bound is not specified for a particular \( x \), the corresponding \( BL(j) \) should be set to -10.)

If IBOUND is set to 3, the user must set \( BL(1) \) to \( l \); E04JAF will then set the remaining elements of \( BL \) equal to \( BL(1) \).

On exit: the lower bounds actually used by E04JAF.

4: BU(N) -- DOUBLE PRECISION array
   On entry: the upper bounds \( u_j \).

If IBOUND is set to 0, the user must set \( BU(j) \) to \( u_j \), for \( j=1,2,...,n \). (If an upper bound is not specified for a particular \( x \), the corresponding \( BU(j) \) should be set to 10.)

If IBOUND is set to 3, the user must set \( BU(1) \) to \( u \); E04JAF will then set the remaining elements of \( BU \) equal to \( BU(1) \).

On exit: the upper bounds actually used by E04JAF.

5: X(N) -- DOUBLE PRECISION array
   On entry: \( X(j) \) must be set to an estimate of the \( j \)th component of the position of the minimum, for \( j=1,2,...,n \).

On exit: the lowest point found during the calculations. Thus, if IFAIL = 0 on exit, \( X(j) \) is the \( j \)th component of the position of the minimum.

6: F -- DOUBLE PRECISION
   On exit: the value of \( F(x) \) corresponding to the final point stored in \( X \).

7: IW(LIW) -- INTEGER array
   Workspace

8: LIW -- INTEGER
   On entry: the length of IW as declared in the (sub)program from which E04JAF is called. Constraint: \( LIW >= N + 2 \).

9: W(LW) -- DOUBLE PRECISION array
   Workspace
10: LW -- INTEGER
On entry: the length of W as declared in the (sub)program from which E04JAF is called. Constraint: LW >= max(N*(N-1)/2+12*N,13).

11: IFAIL -- INTEGER
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit. To suppress the output of an error message when soft failure occurs, set IFAIL to 1.

5.1. Optional Parameters

FUNCT1 -- SUBROUTINE, supplied by the user.

External Procedure

This routine must be supplied by the user to calculate the value of the function F(x) at any point x. Since this routine is not a parameter to E04JAF, it must be called FUNCT1. It should be tested separately before being used in conjunction with E04JAF (see the Chapter Introduction).

Its specification is:

SUBROUTINE FUNCT1 (N, XC, FC)
 INTEGER N
 DOUBLE PRECISION XC(N), FC

1: N -- INTEGER
On entry: the number n of variables.

2: XC(N) -- DOUBLE PRECISION array
On entry: the point x at which the function value is required.

3: FC -- DOUBLE PRECISION
On exit: the value of the function F at the current point x.

FUNCT1 must be declared as EXTERNAL in the (sub)program from which E04JAF is called. Parameters denoted as Input must not be changed by this procedure.
6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
On entry N < 1,
or IBOUND < 0,
or IBOUND > 3,
or IBOUND = 0 and BL(j) > BU(j) for some j,
or IBOUND = 3 and BL(1) > BU(1),
or LIW < N + 2,
or LW<max(13,12*N+N*(N-1)/2).

IFAIL= 2
There have been 400*n function evaluations, yet the algorithm does not seem to be converging. The calculations can be restarted from the final point held in X. The error may also indicate that F(x) has no minimum.

IFAIL= 3
The conditions for a minimum have not all been met but a lower point could not be found and the algorithm has failed.

IFAIL= 4
An overflow has occurred during the computation. This is an unlikely failure, but if it occurs the user should restart at the latest point given in X.

IFAIL= 5

IFAIL= 6

IFAIL= 7

IFAIL= 8
There is some doubt about whether the point x found by E04JAF is a minimum. The degree of confidence in the result decreases as IFAIL increases. Thus, when IFAIL = 5 it is probable that the final x gives a good estimate of the position of a minimum, but when IFAIL = 8 it is very unlikely that the routine has found a minimum.

IFAIL= 9
In the search for a minimum, the modulus of one of the six variables has become very large (\(10^6\)). This indicates that there is a mistake in FUNCT1, that the user's problem has no finite solution, or that the problem needs rescaling (see Section 8).

If the user is dissatisfied with the result (e.g., because IFAIL = 5, 6, 7 or 8), it is worth restarting the calculations from a different starting point (not the point at which the failure occurred) in order to avoid the region which caused the failure. If persistent trouble occurs and the gradient can be calculated, it may be advisable to change to a routine which uses gradients (see the Chapter Introduction).

7. Accuracy

When a successful exit is made then, for a computer with a mantissa of \(t\) decimals, one would expect to get about \(t/2-1\) decimals accuracy in \(x\) and about \(t-1\) decimals accuracy in \(F\), provided the problem is reasonably well scaled.

8. Further Comments

The number of iterations required depends on the number of variables, the behaviour of \(F(x)\) and the distance of the starting point from the solution. The number of operations performed in an iteration of E04JAF is roughly proportional to \(n^2\). In addition, each iteration makes at least \(m+1\) calls of FUNCT1, where \(m\) is the number of variables not fixed on bounds. So, unless \(F(x)\) can be evaluated very quickly, the run time will be dominated by the time spent in FUNCT1.

Ideally the problem should be scaled so that at the solution the value of \(F(x)\) and the corresponding values of \(x_1, x_2, \ldots, x_n\) are each in the range (-1,+1), and so that at points a unit distance away from the solution, \(F\) is approximately a unit value greater than at the minimum. It is unlikely that the user will be able to follow these recommendations very closely, but it is worth trying (by guesswork), as sensible scaling will reduce the difficulty of the minimization problem, so that E04JAF will take less computer time.

9. Example

To minimize

\[
\begin{align*}
2^2 & 2^4 & 4^4
\end{align*}
\]
CHAPTER 15. CHAPTER N

\[ F = \sum_{i=1}^{4} (x_i + 10x_1) + 5(x_2 - x) + (x_3 - 2x_4) + 10(x_4 - x_1) \]

subject to

1. \[ 1 \leq x_1 \leq 3 \]
2. \[ -2 \leq x_2 \leq 0 \]
3. \[ 1 \leq x_3 \leq 3, \]
4. \[ \]

starting from the initial guess \((3, -1, 0, 1)\).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E04 -- Minimizing or Maximizing a Function E04MBF
E04MBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04MBF is an easy-to-use routine for solving linear programming problems, or for finding a feasible point for such problems. It is not intended for large sparse problems.

2. Specification

SUBROUTINE E04MBF (ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1 A, BL, BU, CVEC, LINOBJ, X, ISTATE,
2 OBJLP, CLAMDA, IWORK, LIWORK, WORK,
3 LWORK, IFAIL)
INTEGER ITMAX, MSGLVL, N, NCLIN, NCTOTL, NROWA,
1 ISTATE(NCTOTL), IWORK(LIWORK), LIWORK,
2 LWORK, IFAIL
DOUBLE PRECISION A(NROWA,N), BL(NCTOTL), BU(NCTOTL), CVEC
1 (N), X(N), OBJLP, CLAMDA(NCTOTL), WORK
2 (LIWORK)
LOGICAL LINOBJ
3. Description

E04MBF solves linear programming (LP) problems of the form

\[ \begin{align*}
& \text{Minimize } c^T x \\
& \text{subject to } l_A \leq Ax \leq u_A \quad (\text{LP})
\end{align*} \]

where \( c \) is an \( n \) element vector and \( A \) is an \( m \) by \( n \) matrix i.e., there are \( n \) variables and \( m \) general linear constraints. \( m \) may be zero in which case the LP problem is subject only to bounds on the variables. Notice that upper and lower bounds are specified for all the variables and constraints. This form allows full generality in specifying other types of constraints. For example the \( i \)th constraint may be specified as equality by setting \( l_i = u_i \).

If certain bounds are not present the associated elements of \( l \) or \( u \) can be set to special values that will be treated as -\( \infty \) or +\( \infty \).

The routine allows the linear objective function to be omitted in which case a feasible point for the set of constraints is sought.

The user must supply an initial estimate of the solution.

Users who wish to exercise additional control and users with problems whose solution would benefit from additional flexibility should consider using the comprehensive routine E04NAF.

4. References


5. Parameters

1: ITMAX -- INTEGER Input
On entry: an upper bound on the number of iterations to be taken. If ITMAX is not positive, then the value 50 is used in place of ITMAX.

2: MSGLVL -- INTEGER Input
On entry: indicates whether or not printout is required at the final solution. When printing occurs the output is on the advisory message channel (see X04ABF). A description of
the printed output is given in Section 5.1. The level of
printing is determined as follows:
MSGLVL < 0
No printing.
MSGLVL = 0
Printing only if an input parameter is incorrect, or
if the problem is so ill-conditioned that subsequent
overflow is likely. This setting is strongly
recommended in preference to MSGLVL < 0.
MSGLVL = 1
Printing at the solution.
MSGLVL > 1
Values greater than 1 should normally be used only at
the direction of NAG; such values may generate large
amounts of printed output.

3: N -- INTEGER Input

4: NCLIN -- INTEGER Input
On entry: the number of general linear constraints in the
problem. Constraint: NCLIN >= 0.

5: NCTOTL -- INTEGER Input
On entry: the value (N+NCLIN).

6: NROWA -- INTEGER Input
On entry: the first dimension of the array A as declared in the
(sub)program from which E04MBF is called.
Constraint: NROWA >= max(1,NCLIN).

7: A(NROWA,N) -- DOUBLE PRECISION array Input
On entry: the leading NCLIN by n part of A must contain the
NCLIN general constraints, with the coefficients of the ith
constraint in the ith row of A. If NCLIN = 0, then A is not
referenced.

8: BL(NCTOTL) -- DOUBLE PRECISION array Input
On entry: the first n elements of BL must contain the lower
bounds on the n variables, and when NCLIN > 0, the next
NCLIN elements of BL must contain the lower bounds on the
NCLIN general linear constraints. To specify a non-existent
lower bound (l = -infty), set BL(j) = -1.0E+20.

9: BU(NCTOTL) -- DOUBLE PRECISION array Input
On entry: the first \( n \) elements of \( BU \) must contain the upper bounds on the \( n \) variables, and when \( NCLIN > 0 \), the next \( NCLIN \) elements of \( BU \) must contain the upper bounds on the \( NCLIN \) general linear constraints. To specify a non-existent upper bound (\( u = +\infty \)), set \( BU(j) = 1.0E+20 \). Constraint: 
\[ j \]
\[ BL(j) \leq BU(j), \text{ for } j = 1, 2, \ldots, NCTOTL. \]

10: \( CVEC(N) \) -- DOUBLE PRECISION array  
Input  
On entry: with \( LINOBJ = .TRUE. \), \( CVEC \) must contain the coefficients of the objective function. If \( LINOBJ = .FALSE. \), then \( CVEC \) is not referenced.

11: \( LINOBJ \) -- LOGICAL  
Input  
On entry: indicates whether or not a linear objective function is present. If \( LINOBJ = .TRUE. \), then the full LP problem is solved, but if \( LINOBJ = .FALSE. \), only a feasible point is found and the array \( CVEC \) is not referenced.

12: \( X(N) \) -- DOUBLE PRECISION array  
Input/Output  
On entry: an estimate of the solution, or of a feasible point. Even when \( LINOBJ = .TRUE. \), it is not necessary for the point supplied in \( X \) to be feasible. In the absence of better information all elements of \( X \) may be set to zero. On exit: the solution to the LP problem when \( LINOBJ = .TRUE. \), or a feasible point when \( LINOBJ = .FALSE. \).

When no feasible point exists (see IFAIL = 1 in Section 6) then \( X \) contains the point for which the sum of the infeasibilities is a minimum. On return with IFAIL = 2, 3 or 4, \( X \) contains the point at which E04MBF terminated.

13: \( ISTATE(NCTOTL) \) -- INTEGER array  
Output  
On exit: with IFAIL < 5, ISTATE indicates the status of every constraint at the final point. The first \( n \) elements of ISTATE refer to the upper and lower bounds on the variables and when \( NCLIN > 0 \) the next \( NCLIN \) elements refer to the general constraints.

Their meaning is:
\( ISTATE(j) \) Meaning

-2 The constraint violates its lower bound. This value cannot occur for any element of ISTATE when a feasible point has been found.

-1 The constraint violates its upper bound. This value cannot occur for any element of ISTATE when a feasible point has been found.
0 The constraint is not in the working set (is not active) at the final point. Usually this means that the constraint lies strictly between its bounds.

1 This inequality constraint is in the working set (is active) at its lower bound.

2 This inequality constraint is in the working set (is active) at its upper bound.

3 This constraint is included in the working set (is active) as an equality. This value can only occur when BL(j) = BU(j).

14: OBJLP -- DOUBLE PRECISION
   Output
   On exit: when LINOBJ = .TRUE., then on successful exit, OBJLP contains the value of the objective function at the solution, and on exit with IFAIL = 2, 3 or 4, OBJLP contains the value of the objective function at the point returned in X.
   When LINOBJ = .FALSE., then on successful exit OBJLP will be zero and on return with IFAIL = 1, OBJLP contains the minimum sum of the infeasibilities corresponding to the point returned in X.

15: CLAMDA(NCTOTL) -- DOUBLE PRECISION array
   Output
   On exit: when LINOBJ = .TRUE., then on successful exit, or on exit with IFAIL = 2, 3, or 4, CLAMDA contains the Lagrange multipliers (reduced costs) for each constraint with respect to the working set. The first n components of CLAMDA contain the multipliers for the bound constraints on the variables and the remaining NCLIN components contain the multipliers for the general linear constraints.
   If ISTATE(j) = 0 so that the jth constraint is not in the working set then CLAMDA(j) is zero. If X is optimal and ISTATE(j) = 1, then CLAMDA(j) should be non-negative, and if ISTATE(j) = 2, then CLAMDA(j) should be non-positive.
   When LINOBJ = .FALSE., all NCTOTL elements of CLAMDA are returned as zero.

16: IWORK(LIWORK) -- INTEGER array
    Workspace

17: LIWORK -- INTEGER
    Input
    On entry: the length of the array IWORK as declared in the (sub)program from which E04MBF is called. Constraint: LIWORK>=2*N.
18: WORK(LWORK) -- DOUBLE PRECISION array  Workspace

19: LWORK -- INTEGER  Input
On entry: the length of the array WORK as declared in the (sub)program from which E04MBF is called. Constraints:
when N <= NCLIN then

\[ LWORK \geq 2N + 6N + 4NCLIN + NROWA; \]

when 0 <= NCLIN < N then

\[ LWORK \geq 2(NCLIN+1) + 4NCLIN + 6N + NROWA. \]

20: IFAIL -- INTEGER  Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit. To suppress the output of an error message when soft failure occurs, set IFAIL to 1.

5.1. Description of the Printed Output

When MSGLVL = 1, then E04MBF will produce output on the advisory message channel (see X04ABF), giving information on the final point. The following describes the printout associated with each variable.

<table>
<thead>
<tr>
<th>Output</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAREL</td>
<td>The name (V) and index j, for j=1,2,...,n, of the variable.</td>
</tr>
<tr>
<td>STATE</td>
<td>The state of the variable. (FR if neither bound is in the working set, EQ for a fixed variable, LL if on its lower bound, UL if on its upper bound and TB if held on a temporary bound.) If the value of the variable lies outside the upper or lower bound then STATE will be ++ or -- respectively.</td>
</tr>
<tr>
<td>VALUE</td>
<td>The value of the variable at the final iteration.</td>
</tr>
</tbody>
</table>
LOWER BOUND  The lower bound specified for the variable.

UPPER BOUND  The upper bound specified for the variable.

LAGR MULT   The value of the Lagrange multiplier for the associated bound.

RESIDUAL   The difference between the value of the variable and the nearer of its bounds.

For each of the general constraints the printout is as above with refers to the jth element of Ax, except that VARBL is replaced by:

LNCON   The name (L) and index j, for j = 1,2,...,NCLIN of the constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

Note: when MSGLVL=1 a short description of the error is printed.

IFAIL= 1
No feasible point could be found. Moving violated constraints so that they are satisfied at the point returned in X gives the minimum moves necessary to make the LP problem feasible.

IFAIL= 2
The solution to the LP problem is unbounded.

IFAIL= 3
A total of 50 changes were made to the working set without altering x. Cycling is probably occurring. The user should consider using E04NAF with MSGLVL >= 5 to monitor constraint additions and deletions in order to determine whether or not cycling is taking place.

IFAIL= 4
The limit on the number of iterations has been reached.
Increase ITMAX or consider using E04NAF to monitor progress.

IFAIL= 5
An input parameter is invalid. Unless MSGLVL < 0 a message will be printed.

IFAILOverflow
If the printed output before the overflow occurred contains a warning about serious ill-conditioning in the working set
when adding the jth constraint, then either the user should try using E04NAF and experiment with the magnitude of FEATOL (j) in that routine, or the offending linearly dependent constraint (with index j) should be removed from the problem.

7. Accuracy

The routine implements a numerically stable active set strategy and returns solutions that are as accurate as the condition of the LP problem warrants on the machine.

8. Further Comments

The time taken by each iteration is approximately proportional to

\[ \min(n, n_{\text{NCLIN}}) \].

Sensible scaling of the problem is likely to reduce the number of iterations required and make the problem less sensitive to perturbations in the data, thus improving the condition of the LP problem. In the absence of better information it is usually sensible to make the Euclidean lengths of each constraint of comparable magnitude. See Gill et al [1] for further information and advice.

Note that the routine allows constraints to be violated by an absolute tolerance equal to the machine precision (see X02AJF(*))

9. Example

To minimize the function

\[ -0.02x -0.2x -0.2x -0.2x -0.2x +0.04x +0.04x \]

subject to the bounds

\[ -0.01 \leq x \leq 0.01 \]
\[ -0.1 \leq x \leq 0.15, \]
\[ -0.01 \leq x \leq 0.03, \]
\[ -0.04 \leq x \leq 0.02, \]
\[ -0.1 \leq x \leq 0.05, \]
\[ -0.01 \leq x \]

1 2 3 4 5 6 7
-0.01 <= x
  7

and the general constraints

\[ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 = -0.13 \]
\[ 0.15x_1 + 0.04x_2 + 0.02x_3 + 0.04x_4 + 0.02x_5 + 0.01x_6 + 0.03x_7 <= -0.0049 \]
\[ 0.03x_1 + 0.05x_2 + 0.08x_3 + 0.02x_4 + 0.06x_5 + 0.01x_6 <= -0.0064 \]
\[ 0.02x_1 + 0.04x_2 + 0.01x_3 + 0.02x_4 + 0.02x_5 <= -0.0037 \]
\[ 0.02x_1 + 0.03x_2 + 0.01x_3 <= -0.0012 \]
\[ -0.0992 <= 0.70x_1 + 0.75x_2 + 0.80x_3 + 0.75x_4 + 0.80x_5 + 0.97x_6 \]
\[ -0.003 <= 0.02x_1 + 0.06x_2 + 0.08x_3 + 0.12x_4 + 0.02x_5 + 0.01x_6 + 0.97x_7 <= 0.002 \]

The initial point, which is infeasible, is

\[ x = (-0.01, -0.03, 0.0, -0.01, -0.1, 0.02, 0.01) \]
intended for large sparse problems.

2. Specification

```fortran
SUBROUTINE E04NAF (ITMAX, MSGVL, N, NCLIN, NCTOTL, NROWA,
    1 NROWH, NCOLH, BIGBND, A, BL, BU, CVEC,
    2 FEATOL, HESS, QPHESS, COLD, LP, ORTHOG,
    3 X, ISTATE, ITER, OBJ, CLAMDA, IWORK,
    4 LIWORK, WORK, LWORK, IFAIL)
INTEGER ITMAX, MSGVL, N, NCLIN, NCTOTL, NROWA,
    1 NROWH, NCOLH, ISTATE(NCTOTL), ITER, IWORK
    2 (LIWORK), LIWORK, LWORK, IFAIL
DOUBLE PRECISION BIGBND, A(NROWA,N), BL(NCTOTL),
    1 BU(NCTOTL), CVEC(N), FEATOL(NCTOTL), HESS
    2 (NROWH,NCOLH), X(N), OBJ, CLAMDA(NCTOTL),
    3 WORK(LWORK)
LOGICAL COLD, LP, ORTHOG
EXTERNAL QPHESS
```

3. Description

E04NAF is essentially identical to the subroutine SOL/QPSOL described in Gill et al [1].

E04NAF is designed to solve the quadratic programming (QP) problem - the minimization of a quadratic function subject to a set of linear constraints on the variables. The problem is assumed to be stated in the following form:

$$
\text{Minimize } c^T x + \frac{1}{2} x^T H x \quad \text{subject to } l \leq (A x) \leq u,
$$

where $c$ is a constant $n$-vector and $H$ is a constant $n \times n$ symmetric matrix; note that $H$ is the Hessian matrix (matrix of second partial derivatives) of the quadratic objective function. The matrix $A$ is $m \times n$, where $m$ may be zero; $A$ is treated as a dense matrix.

The constraints involving $A$ will be called the general constraints. Note that upper and lower bounds are specified for all the variables and for all the general constraints. The form of (1) allows full generality in specifying other types of constraints. In particular, an equality constraint is specified by setting $l_i = u_i$. If certain bounds are not present, the associated elements of $l$ or $u$ can be set to special values that will be treated as $-\infty$ or $+\infty$.

The user must supply an initial estimate of the solution to (1),
and a subroutine that computes the product $Hx$ for any given vector $x$. If $H$ is positive-definite or positive semi-definite, E04NAF will obtain a global minimum; otherwise, the solution obtained will be a local minimum (which may or may not be a global minimum). If $H$ is defined as the zero matrix, E04NAF will solve the resulting linear programming (LP) problem; however, this can be accomplished more efficiently by setting a logical variable in the call of the routine (see the parameter LP in Section 5).

E04NAF allows the user to provide the indices of the constraints that are believed to be exactly satisfied at the solution. This facility, known as a warm start, can lead to significant savings in computational effort when solving a sequence of related problems.

The method has two distinct phases. In the first (the LP phase), an iterative procedure is carried out to determine a feasible point. In this context, feasibility is defined by a user-provided array FEATOL; the $j$th constraint is considered satisfied if its violation does not exceed FEATOL($j$). The second phase (the QP phase) generates a sequence of feasible iterates in order to minimize the quadratic objective function. In both phases, a subset of the constraints - called the working set - is used to define the search direction at each iteration; typically, the working set includes constraints that are satisfied to within the corresponding tolerances in the FEATOL array.

We now briefly describe a typical iteration in the QP phase. Let $x_k$ denote the estimate of the solution at the $k$th iteration; the next iterate is defined by

$$x_{k+1} = x_k + \alpha_k p_k$$

where $p_k$ is an $n$-dimensional search direction and $\alpha_k$ is a scalar step length. Assume that the working (active) set contains $t$ linearly independent constraints, and let $C_k$ denote the matrix of coefficients of the bounds and general constraints in the current working set.

Let $Z_k$ denote a matrix whose columns form a basis for the null space of $C_k$, so that $C_k Z_k = 0$. (Note that $Z_k$ has $n$ columns, where $n = n-t$.) The vector $Z_k (c + Hx_k)$ is called the projected gradient.
at \( x \). If the projected gradient is zero at \( x \) (i.e., \( x \) is a
constrained stationary point in the subspace defined by \( Z \)),
Lagrange multipliers \((\lambda)\) are defined as the solution of the
compatible overdetermined system
\[
T
C (\lambda) = c + Hx
\]
The Lagrange multiplier \((\lambda)\) corresponding to an inequality
constraint in the working set is said to be optimal if
\((\lambda) \leq 0\) when the associated constraint is at its upper bound,
or if \((\lambda) \geq 0\) when the associated constraint is at its lower
bound. If a multiplier is non-optimal, the objective function can
be reduced by deleting the corresponding constraint (with index
\(J_{\text{DEL}}\), see Section 5.1) from the working set.

If the projected gradient at \( x \) is non-zero, the search direction
\( p \) is defined as
\[
p = Z p
\]
where \( p \) is an \( n \)-vector. In effect, the constraints in the
working set are treated as equalities, by constraining \( p \) to lie
within the subspace of vectors orthogonal to the rows of \( C \). This
definition ensures that \( C p = 0 \), and hence the values of the
constraints in the working set are not altered by any move along
\( p \).

The vector \( p \) is obtained by solving the equations
\[
T
Z H Z p = - Z (c + Hx)
\]
\((The \text{ matrix } Z H Z \text{ is called the projected Hessian matrix.)} \) If the
If the projected Hessian is positive-definite and $x + \kappa \eta$ is feasible, \( \kappa \) will be taken as unity. In this case, the projected gradient at $x$ will be zero (see NORM ZTG in Section 5.1), and Lagrange multipliers can be computed (see Gill et al [2]). Otherwise, \( \kappa \) is set to the step to the 'nearest' constraint (with index JADD, see Section 5.1), which is added to the working set at the next iteration.

The matrix $Z$ is obtained from the TQ factorization of $C$, in which $C$ is represented as

$$ C Q = \begin{bmatrix} 0 & T \end{bmatrix} $$

where $T$ is reverse-triangular. It follows from (5) that $Z$ may be taken as the first $n$ columns of $Q$. If the projected Hessian is positive-definite, (3) is solved using the Cholesky factorization

$$ T^T \eta = R \eta $$

where $R$ is upper triangular. These factorizations are updated as constraints enter or leave the working set (see Gill et al [2] for further details).

An important feature of E04NAF is the treatment of indefiniteness in the projected Hessian. If the projected Hessian is positive-definite, it may become indefinite only when a constraint is deleted from the working set. In this case, a temporary modification (of magnitude HESS MOD, see Section 5.1) is added to the last diagonal element of the Cholesky factor. Once a modification has occurred, no further constraints are deleted from the working set until enough constraints have been added so
that the projected Hessian is again positive-definite. If equation (1) has a finite solution, a move along the direction obtained by solving (4) with the modified Cholesky factor must encounter a constraint that is not already in the working set.

In order to resolve indefiniteness in this way, we must ensure that the projected Hessian is positive-definite at the first iterate in the QP phase. Given the \( n \times n \) projected Hessian, a step-wise Cholesky factorization is performed with symmetric interchanges (and corresponding rearrangement of the columns of \( Z \)), terminating if the next step would cause the matrix to become indefinite. This determines the largest possible positive-definite principal sub-matrix of the (permuted) projected Hessian. If \( n \) steps of the Cholesky factorization have been successfully completed, the relevant projected Hessian is an \( n \times n \) positive-definite matrix \( Z^T H Z \), where \( Z \) comprises the first \( n \) columns of \( Z \). The quadratic function will subsequently be minimized within subspaces of reduced dimension until the full projected Hessian is positive-definite.

If a linear program is being solved and there are fewer general constraints than variables, the method moves from one vertex to another while minimizing the objective function. When necessary, an initial vertex is defined by temporarily fixing some of the variables at their initial values.

Several strategies are used to control ill-conditioning in the working set. One such strategy is associated with the FEATOL array. Allowing the \( j \)th constraint to be violated by as much as \( \text{FEATOL}(j) \) often provides a choice of constraints that could be added to the working set. When a choice exists, the decision is based on the conditioning of the working set. Negative steps are occasionally permitted, since \( x \) may violate the constraint to be added.

4. References


5. Parameters

1: ITMAX -- INTEGER
   Input
   On entry: an upper bound on the number of iterations to be taken during the LP phase or the QP phase. If ITMAX is not positive, then the value 50 is used in place of ITMAX.

2: MSGlvl -- INTEGER
   Input
   On entry: MSGlvl must indicate the amount of intermediate output desired (see Section 5.1 for a description of the printed output). All output is written to the current advisory message unit (see X04ABF). For MSGlvl >= 10, each level includes the printout for all lower levels.
   Value Definition
   <0 No printing.
   0 Printing only if an input parameter is incorrect, or if the working set is so ill-conditioned that subsequent overflow is likely. This setting is strongly recommended in preference to MSGlvl < 0.
   1 The final solution only.
   5 One brief line of output for each constraint addition or deletion (no printout of the final solution).
   >=10 The final solution and one brief line of output for each constraint addition or deletion.
   >=15 At each iteration, X, ISTATE, and the indices of the free variables (i.e., the variables not currently held on a bound).
   >=20 At each iteration, the Lagrange multiplier estimates and the general constraint values.
   >=30 At each iteration, the diagonal elements of the matrix T associated with the TQ factorization of the working set, and the diagonal elements of the Cholesky factor R of the projected Hessian.
   >=80 Debug printout.
The arrays CVEC and HESS.

3: N -- INTEGER Input

4: NCLIN -- INTEGER Input
   On entry: the number of general linear constraints in the
   problem. Constraint: NCLIN >= 0.

5: NCTOTL -- INTEGER Input
   On entry: the value (N+NCLIN).

6: NROWA -- INTEGER Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which E04NAF is called.
   Constraint: NROWA >= max(1,NCLIN).

7: NROWH -- INTEGER Input
   On entry: the first dimension of the array HESS as declared
   in the (sub)program from which E04NAF is called.
   Constraint: NROWH >= 1.

8: NCOLH -- INTEGER Input
   On entry: the column dimension of the array HESS as declared
   in the (sub)program from which E04NAF is called.
   Constraint: NCOLH >= 1.

9: BIGBND -- DOUBLE PRECISION Input
   On entry: BIGBND must denote an 'infinite' component of l
   and u. Any upper bound greater than or equal to BIGBND will
   be regarded as plus infinity, and a lower bound less than or
   equal to -BIGBND will be regarded as minus infinity.
   Constraint: BIGBND > 0.0.

10: A(NROWA,N) -- DOUBLE PRECISION array Input
    On entry: the leading NCLIN by n part of A must contain the
    NCLIN general constraints, with the ith constraint in the i
    th row of A. If NCLIN = 0, then A is not referenced.

11: BL(NCTOTL) -- DOUBLE PRECISION array Input
    On entry: the lower bounds for all the constraints, in the
    following order. The first n elements of BL must contain the
    lower bounds on the variables. If NCLIN > 0, the next NCLIN
    elements of BL must contain the lower bounds for the general
    linear constraints. To specify a non-existent lower bound
    (i.e., l = -infty), the value used must satisfy BL(j)<=-j
    BIGBND To specify the jth constraint as an equality, the
user must set BL(j) = BU(j). Constraint: BL(j) <= BU(j),
j=1,2,...,NCTOTL.

12:   BU(NCTOTL) -- DOUBLE PRECISION array   Input
On entry: the upper bounds for all the constraints, in the
following order. The first n elements of BU must contain the
upper bounds on the variables. If NCLIN > 0, the next NCLIN
elements of BU must contain the upper bounds for the general
linear constraints. To specify a non-existent upper bound
(i.e., \( u = +\infty \)), the value used must satisfy \( BU(j) \geq BIGBND \).

13:   CVEC(N) -- DOUBLE PRECISION array   Input
On entry: the coefficients of the linear term of the
objective function (the vector c in equation (1)).

14:   FEATOL(NCTOTL) -- DOUBLE PRECISION array   Input
On entry: a set of positive tolerances that define the
maximum permissible absolute violation in each constraint in
order for a point to be considered feasible, i.e., if the
violation in constraint j is less than \( FEATOL(j) \), the point
is considered to be feasible with respect to the jth
constraint. The ordering of the elements of FEATOL is the
same as that described above for BL.

The elements of FEATOL should not be too small and a warning
message will be printed on the current advisory message
channel if any element of FEATOL is less than the machine
precision (see X02AJF(*)). As the elements of FEATOL
increase, the algorithm is less likely to encounter
difficulties with ill-conditioning and degeneracy. However,
larger values of \( FEATOL(j) \) mean that constraint j could be
violated by a significant amount. It is recommended that
\( FEATOL(j) \) be set to a value equal to the largest acceptable
violation for constraint j. For example, if the data
defining the constraints are of order unity and are correct
to about 6 decimal digits, it would be appropriate to choose
\(-6\)
\( FEATOL(j) \) as 10 for all relevant j. Often the square root
of the machine precision is a reasonable choice if the
constraint is well scaled.

15:   HESS(NROWH,NCOLH) -- DOUBLE PRECISION array   Input
On entry: HESS may be used to store the Hessian matrix H of
equation (1) if desired. HESS is accessed only by the
subroutine QPHESS and is not accessed if LP = .TRUE.. Refer
to the specification of QPHESS (below) for further details.
of how HESS may be used to pass data to QPHESS.

16: QPHESS -- SUBROUTINE, supplied by the user.

External Procedure

QPHESS must define the product of the Hessian matrix $H$ and a vector $x$. The elements of $H$ need not be defined explicitly. QPHESS is not accessed if LP is set to .TRUE. and in this case QPHESS may be the dummy routine E04NAN. (E04NAN is included in the NAG Foundation Library and so need not be supplied by the user. Its name may be implementation-dependent: see the Users' Note for your implementation for details.)

Its specification is:

```fortran
SUBROUTINE QPHESS (N, NROWH, NCOLH, JTHCOL,
                  1 HESS, X, HX)
INTEGER N, NROWH, NCOLH, JTHCOL
DOUBLE PRECISION HESS(NROWH,NCOLH), X(N), HX(N)
```

1: $N$ -- INTEGER  
   Input  
   On entry: the number $n$ of variables.

2: $NROWH$ -- INTEGER  
   Input  
   On entry: the row dimension of the array $HESS$.

3: $NCOLH$ -- INTEGER  
   Input  
   On entry: the column dimension of the array $HESS$.

4: $JTHCOL$ -- INTEGER  
   Input  
   The input parameter $JTHCOL$ is included to allow flexibility for the user in the special situation when $x$ is the $j$th co-ordinate vector (i.e., the $j$th column of the identity matrix). This may be of interest because the product $Hx$ is then the $j$th column of $H$, which can sometimes be computed very efficiently. The user may code QPHESS to take advantage of this case. On entry: if $JTHCOL = j$, where $j > 0$, $HX$ must contain column $JTHCOL$ of $H$, and hence special code may be included in QPHESS to test $JTHCOL$ if desired. However, special code is not necessary, since the vector $x$ always contains column $JTHCOL$ of the identity matrix whenever QPHESS is called with $JTHCOL > 0$.

5: $HESS(NROWH,NCOLH)$ -- DOUBLE PRECISION array  
   Input  
   On entry: the Hessian matrix $H$.

In some cases, it may be desirable to use a one-dimensional array to transmit data or workspace to QPHESS; $HESS$ should then be declared with dimension
(NROWH) in the (sub)program from which E04NAF is called and the parameter NCOLH must be 1.

In other situations, it may be desirable to compute Hx without accessing HESS - for example, if H is sparse or has special structure. (This is illustrated in the subroutine QPHESS1 in the example program in Section 9.) The parameters HESS, NROWH and NCOLH may then refer to any convenient array.

When MSGlvl = 99, the (possibly undefined) contents of HESS will be printed, except if NROWH and NCOLH are both 1. Also printed are the results of calling QPHESS with JTHCOL = 1,2,...,n.

6: X(N) -- DOUBLE PRECISION array  
   On entry: the vector x.

7: HX(N) -- DOUBLE PRECISION array  
   On exit: HX must contain the product Hx. QPHESS must be declared as EXTERNAL in the (sub)program from which E04NAF is called. Parameters denoted as Input must not be changed by this procedure.

17: COLD -- LOGICAL  
   On entry: COLD must indicate whether the user has specified an initial estimate of the active set of constraints. If COLD is set to .TRUE., the initial working set is determined by E04NAF. If COLD is set to .FALSE. (a 'warm start'), the user must define the ISTATE array which gives the status of each constraint with respect to the working set. E04NAF will override the user's specification of ISTATE if necessary, so that a poor choice of working set will not cause a fatal error.

The warm start option is particularly useful when E04NAF is called repeatedly to solve related problems.

18: LP -- LOGICAL  
   On entry: if LP = .FALSE., E04NAF will solve the specified quadratic programming problem. If LP = .TRUE., E04NAF will treat H as zero and solve the resulting linear programming problem; in this case, the parameters HESS and QPHESS will not be referenced.

19: ORTHOG -- LOGICAL  
   On entry: ORTHOG must indicate whether orthogonal transformations are to be used in computing and updating the TQ factorization of the working set

\[ A Q = (0 T), \]
where A is a sub-matrix of A and T is reverse-triangular.

If ORTHOG = .TRUE., the TQ factorization is computed using Householder reflections and plane rotations, and the matrix Q is orthogonal. If ORTHOG = .FALSE., stabilized elementary transformations are used to maintain the factorization, and Q is not orthogonal. A rule of thumb in making the choice is that orthogonal transformations require more work, but provide greater numerical stability. Thus, we recommend setting ORTHOG to .TRUE. if the problem is reasonably small or the active set is ill-conditioned. Otherwise, setting ORTHOG to .FALSE. will often lead to a reduction in solution time with negligible loss of reliability.

20: X(N) -- DOUBLE PRECISION array Input/Output
On entry: an estimate of the solution. In the absence of better information all elements of X may be set to zero. On exit: from E04NAF, X contains the best estimate of the solution.

21: ISTATE(NCTOTL) -- INTEGER array Input/Output
On entry: with COLD as .FALSE., ISTATE must indicate the status of every constraint with respect to the working set. The ordering of ISTATE is as follows: the first n elements of ISTATE refer to the upper and lower bounds on the variables and elements n+1 through n + NCLIN refer to the upper and lower bounds on Ax. The significance of each possible value of ISTATE(j) is as follows:

<table>
<thead>
<tr>
<th>ISTATE(j)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2</td>
<td>The constraint violates its lower bound by more than FEATOL(j). This value of ISTATE cannot occur after a feasible point has been found.</td>
</tr>
<tr>
<td>-1</td>
<td>The constraint violates its upper bound by more than FEATOL(j). This value of ISTATE cannot occur after a feasible point has been found.</td>
</tr>
<tr>
<td>0</td>
<td>The constraint is not in the working set. Usually, this means that the constraint lies strictly between its bounds.</td>
</tr>
<tr>
<td>1</td>
<td>This inequality constraint is included in the working set at its lower bound. The value of the constraint is within FEATOL(j) of its lower bound.</td>
</tr>
<tr>
<td>2</td>
<td>This inequality constraint is included in the working set at its upper bound. The value of the constraint is within FEATOL(j) of its upper bound.</td>
</tr>
</tbody>
</table>
The constraint is included in the working set as an equality. This value of ISTATE can occur only when BL(j) = BU(j). The corresponding constraint is within FEATOL(j) of its required value. If COLD = .TRUE., ISTATE need not be set by the user. However, when COLD = .FALSE., every element of ISTATE must be set to one of the values given above to define a suggested initial working set (which will be changed by E04NAF if necessary). The most likely values are:

<table>
<thead>
<tr>
<th>ISTATE(j)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The corresponding constraint should not be in the initial working set.</td>
</tr>
<tr>
<td>1</td>
<td>The constraint should be in the initial working set at its lower bound.</td>
</tr>
<tr>
<td>2</td>
<td>The constraint should be in the initial working set at its upper bound.</td>
</tr>
<tr>
<td>3</td>
<td>The constraint should be in the initial working set as an equality. This value must not be specified unless BL(j) = BU(j). The values 1, 2 or 3 all have the same effect when BL(j) = BU(j).</td>
</tr>
</tbody>
</table>

Note that if E04NAF has been called previously with the same values of N and NCLIN, ISTATE already contains satisfactory values. On exit: when E04NAF exits with IFAIL set to 0, 1 or 3, the values in the array ISTATE indicate the status of the constraints in the active set at the solution. Otherwise, ISTATE indicates the composition of the working set at the final iterate.

22: ITER -- INTEGER
On exit: the number of iterations performed in either the LP phase or the QP phase, whichever was last entered.

Note that ITER is reset to zero after the LP phase.

23: OBJ -- DOUBLE PRECISION
On exit: the value of the quadratic objective function at x if x is feasible (IFAIL <= 5), or the sum of infeasibilities at x otherwise (6 <= IFAIL <= 8).

24: CLAMDA(NCTOTL) -- DOUBLE PRECISION array
On exit: the values of the Lagrange multiplier for each constraint with respect to the current working set. The ordering of CLAMDA is as follows; the first n components contain the multipliers for the bound constraints on the variables, and the remaining components contain the
multipliers for the general linear constraints. If ISTATE(j) = 0 (i.e., constraint j is not in the working set), CLAMDA(j) is zero. If x is optimal and ISTATE(j) = 1, CLAMDA(j) should be non-negative; if ISTATE(j) = 2, CLAMDA(j) should be non-positive.

25: IWORK(LIWORK) -- INTEGER array Workspace

26: LIWORK -- INTEGER Input
On entry:
the dimension of the array IWORK as declared in the (sub)program from which E04NAF is called.
Constraint: LIWORK>=2*N.

27: WORK(LWORK) -- DOUBLE PRECISION array Workspace

28: LWORK -- INTEGER Input
On entry:
the dimension of the array WORK as declared in the (sub)program from which E04NAF is called.
Constraint if LP = .FALSE. or NCLIN >= N then
nts:

2
LWORK>=2*N +4*N*NCLIN+NROWA.

if LP = .TRUE. and NCLIN < N then
2
LWORK>=2*(NCLIN+1) +4*N+2*NCLIN+NROWA.

If MSGLVL > 0, the amount of workspace provided and the amount of workspace required are output on the current advisory message unit (as defined by X04ABF). As an alternative to computing LWORK from the formula given above, the user may prefer to obtain an appropriate value from the output of a preliminary run with a positive value of MSGLVL and LWORK set to 1 (E04NAF will then terminate with IFAIL = 9).

29: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit. To suppress the output of an error message when soft failure occurs, set IFAIL to 1.
IFAIL contains zero on exit if \( x \) is a strong local minimum. i.e., the projected gradient is negligible, the Lagrange multipliers are optimal, and the projected Hessian is positive-definite. In some cases, a zero value of IFAIL means that \( x \) is a global minimum (e.g. when the Hessian matrix is positive-definite).

5.1. Description of the Printed Output

When MSGLVL \( \geq 5 \), a line of output is produced for every change in the working set (thus, several lines may be printed during a single iteration).

To aid interpretation of the printed results, we mention the convention for numbering the constraints: indices 1 through to \( n \) refer to the bounds on the variables, and when NCLIN > 0 indices \( n+1 \) through to \( n + NCLIN \) refer to the general constraints. When the status of a constraint changes, the index of the constraint is printed, along with the designation L (lower bound), U (upper bound) or E (equality).

In the LP phase, the printout includes the following:

\begin{itemize}
  \item ITN is the iteration count.
  \item JDEL is the index of the constraint deleted from the working set. If JDEL is zero, no constraint was deleted.
  \item JADD is the index of the constraint added to the working set. If JADD is zero, no constraint was added.
  \item STEP is the step taken along the computed search direction.
  \item CONDT is a lower bound on the condition number of the matrix of predicted active constraints.
  \item NUMINF is the number of violated constraints (infeasibilities).
  \item SUMINF is a weighted sum of the magnitudes of the constraint violations.
  \item LPOBJ is the value of the linear objective function \( c^T x \). It is printed only if LP = .TRUE..
During the QP phase, the printout includes the following:

**ITN**  
is the iteration count (reset to zero after the LP phase).

**JDEL**  
is the index of the constraint deleted from the working set. If JDEL is zero, no constraint was deleted.

**JADD**  
is the index of the constraint added to the working set. If JADD is zero, no constraint was added.

**STEP**  
is the step (alpha) taken along the direction of $k$ search (if STEP is 1.0, the current point is a minimum in the subspace defined by the current working set).

**NHESS**  
is the number of calls to subroutine QPHESS.

**OBJECTIVE**  
is the value of the quadratic objective function.

**NCOLZ**  
is the number of columns of $Z$ (see Section 3). In general, it is the dimension of the subspace in which the quadratic is currently being minimized.

**NORM GFREE**  
is the Euclidean norm of the gradient of the objective function with respect to the free variables, i.e. variables not currently held at a bound (NORM GFREE is not printed if ORTHOG = .FALSE.). In some cases, the objective function and gradient are updated rather than recomputed. If so, this entry will be -- to indicate that the gradient with respect to the free variables has not been computed.

**NORM QTG**  
is a weighted norm of the gradient of the objective function with respect to the free variables (NORM QTG is not printed if ORTHOG = .TRUE.). In some cases, the objective function and gradient are updated rather than recomputed. If so, this entry will be -- to indicate that the gradient with respect to the free variables has not been computed.

**NORM ZTG**  
is the Euclidean norm of the projected gradient (see Section 3).

**COND T**  
is a lower bound on the condition number of the...
matrix of constraints in the working set.

COND ZHZ is a lower bound on the condition number of the projected Hessian matrix.

HESS MOD is the correction added to the diagonal of the projected Hessian to ensure that a satisfactory Cholesky factorization exists (see Section 3). When the projected Hessian is sufficiently positive-definite, HESS MOD will be zero.

When MSGLVL = 1 or MSGLVL >= 10, the summary printout at the end of execution of E04NAF includes a listing of the status of every constraint. Note that default names are assigned to all variables and constraints.

The following describes the printout for each variable.

VARBL is the name (V) and index j, j=1,2,...,n, of the variable.

STATE gives the state of the variable (FR if neither bound is in the working set, EQ if a fixed variable, LL if on its lower bound, UL if on its upper bound, TB if held on a temporary bound). If VALUE lies outside the upper or lower bounds by more than FEATOL(j), STATE will be ++ or -- respectively.

VALUE is the value of the variable at the final iteration.

LOWER BOUND is the lower bound specified for the variable.

UPPER BOUND is the upper bound specified for the variable.

LAGR MULT is the value of the Lagrange multiplier for the associated bound constraint. This will be zero if STATE is FR. If x is optimal and STATE is LL, the multiplier should be non-negative; if STATE is UL, the multiplier should be non-positive.

RESIDUAL is the difference between the variable and the nearer of its bounds BL(j) and BU(j).

For each of the general constraints the printout is as above with refers to the jth element of Ax, except that VARBL is replaced by LNCON The name (L) and index j, j=1,2,...,NCLIN, of the constraint.
6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
x is a weak local minimum (the projected gradient is negligible, the Lagrange multipliers are optimal, but the projected Hessian is only semi-definite). This means that the solution is not unique.

IFAIL= 2
The solution appears to be unbounded, i.e., the quadratic function is unbounded below in the feasible region. This value of IFAIL occurs when a step of infinity would have to be taken in order to continue the algorithm.

IFAIL= 3
x appears to be a local minimum, but optimality cannot be verified because some of the Lagrange multipliers are very small in magnitude.

E04NAF has probably found a solution. However, the presence of very small Lagrange multipliers means that the predicted active set may be incorrect, or that x may be only a constrained stationary point rather than a local minimum. The method in E04NAF is not guaranteed to find the correct active set when there are very small multipliers. E04NAF attempts to delete constraints with zero multipliers, but this does not necessarily resolve the issue. The determination of the correct active set is a combinatorial problem that may require an extremely large amount of time. The occurrence of small multipliers often (but not always) indicates that there are redundant constraints.

IFAIL= 4
The iterates of the QP phase could be cycling, since a total of 50 changes were made to the working set without altering x.

This value will occur if 50 iterations are performed in the QP phase without changing x. The user should check the printed output for a repeated pattern of constraint deletions and additions. If a sequence of constraint changes is being repeated, the iterates are probably cycling. (E04NAF does not contain a method that is guaranteed to avoid cycling, which would be combinatorial in nature.) Cycling may occur in two circumstances: at a constrained stationary point where there are some small or zero Lagrange multipliers (see the discussion of IFAIL = 3); or at a point
(usually a vertex) where the constraints that are satisfied exactly are nearly linearly dependent. In the latter case, the user has the option of identifying the offending dependent constraints and removing them from the problem, or restarting the run with larger values of FEATOL for nearly dependent constraints. If E04NAF terminates with IFAIL = 4, but no suspicious pattern of constraint changes can be observed, it may be worthwhile to restart with the final x (with or without the warm start option).

IFAIL = 5
The limit of ITMAX iterations was reached in the QP phase before normal termination occurred.

The value of ITMAX may be too small. If the method appears to be making progress (e.g. the objective function is being satisfactorily reduced), increase ITMAX and rerun E04NAF (possibly using the warm start facility to specify the initial working set). If ITMAX is already large, but some of the constraints could be nearly linearly dependent, check the output for a repeated pattern of constraints entering and leaving the working set. (Near-dependencies are often indicated by wide variations in size in the diagonal elements of the T matrix, which will be printed if MSGlvl >= 30.) In this case, the algorithm could be cycling (see the comments for IFAIL = 4).

IFAIL = 6
The LP phase terminated without finding a feasible point, and hence it is not possible to satisfy all the constraints to within the tolerances specified by the FEATOL array. In this case, the final iterate will reveal values for which there will be a feasible point (e.g. a feasible point will exist if the feasibility tolerance for each violated constraint exceeds its RESIDUAL at the final point). The modified problem (with altered values in FEATOL) may then be solved using a warm start.

The user should check that there are no constraint redundancies. If the data for the jth constraint are accurate only to the absolute precision (delta), the user should ensure that the value of FEATOL(j) is greater than (delta). For example, if all elements of A are of order unity and are accurate only to three decimal places, every -3 component of FEATOL should be at least 10 .

IFAIL = 7
The iterates may be cycling during the LP phase; see the comments above under IFAIL = 4.
IFAIL= 8
The limit of ITMAX iterations was reached during the LP
phase. See comments above under IFAIL = 5.

IFAIL= 9
An input parameter is invalid.

Overflow
If the printed output before the overflow error contains a
warning about serious ill-conditioning in the working set
when adding the jth constraint, it may be possible to avoid
the difficulty by increasing the magnitude of FEATOL(j) and
rerunning the program. If the message recurs even after this
change, the offending linearly dependent constraint (with
index j) must be removed from the problem. If a warning
message did not precede the fatal overflow, the user should
contact NAG.

7. Accuracy
The routine implements a numerically stable active set strategy
and returns solutions that are as accurate as the condition of
the QP problem warrants on the machine.

8. Further Comments
The number of iterations depends upon factors such as the number
of variables and the distances of the starting point from the
solution. The number of operations performed per iteration is

\[ 2 \]

roughly proportional to (NFREE), where NFREE (NFREE<\(n\)) is the
number of variables fixed on their upper or lower bounds.

Sensible scaling of the problem is likely to reduce the number of
iterations required and make the problem less sensitive to
perturbations in the data, thus improving the condition of the QP
problem. See the Chapter Introduction and Gill et al [1] for
further information and advice.

9. Example

\[ T \begin{bmatrix} 1 \\ T \end{bmatrix} \]

To minimize the function \( c^T x + -x^T H x \), where

\[ c=\begin{bmatrix} -0.02, -0.2, -0.2, -0.2, -0.2, 0.04, 0.04 \end{bmatrix} \]

\[ \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \]
subject to the bounds

1. Purpose

E04UCF is designed to minimize an arbitrary smooth function subject to constraints, which may include simple bounds on the variables, linear constraints and smooth nonlinear constraints. (E04UCF may be used for unconstrained, bound-constrained and linearly constrained optimization.) The user must provide subroutines that define the objective and constraint functions and as many of their first partial derivatives as possible. Unspecified derivatives are approximated by finite differences. All matrices are treated as dense, and hence E04UCF is not intended for large sparse problems.

E04UCF uses a sequential quadratic programming (SQP) algorithm in which the search direction is the solution of a quadratic programming (QP) problem. The algorithm treats bounds, linear constraints and nonlinear constraints separately.

2. Specification

```fortran
SUBROUTINE E04UCF (N, NCLIN, NCNLN, NROWA, NROWJ, NROWR,
  1   A, BL, BU, CONFUN, OBJFUN, ITER,
  2   ISTATE, C, CJAC, CLAMDA, OBJF, OBJGRD,
  3   R, X, IWORK, LIWORK, WORK, LWORK,
```
3. Description

E04UCF is designed to solve the nonlinear programming problem --
the minimization of a smooth nonlinear function subject to a set
of constraints on the variables. The problem is assumed to be
stated in the following form:

\[
\begin{align*}
\text{Minimize} & \quad F(x) \quad \text{subject to} \\
\text{subject to} & \quad \begin{cases}
\quad 1 \leq \{A x\} \leq u,
\quad \{L\}
\quad \{c(x)\}
\end{cases}
\end{align*}
\]

where \(F(x)\), the objective function, is a nonlinear function, \(A\)
is an \(n \times n\) constant matrix, and \(c(x)\) is an \(n\) element vector
of nonlinear constraint functions. (The matrix \(A\) and the vector
\(c(x)\) may be empty.) The objective function and the constraint
functions are assumed to be smooth, i.e., at least twice-
continuously differentiable. (The method of E04UCF will usually
solve (1) if there are only isolated discontinuities away from
the solution.)

This routine is essentially identical to the subroutine SOL/NPSOL
described in Gill et al [8].

Note that upper and lower bounds are specified for all the
variables and for all the constraints.

An equality constraint can be specified by setting \(l = u\). If
certain bounds are not present, the associated elements of \( l \) or \( u \) can be set to special values that will be treated as -\( \infty \) or +\( \infty \).

If there are no nonlinear constraints in (1) and \( F \) is linear or quadratic then one of E04MBF, E04NAF or E04NCF(*) will generally be more efficient. If the problem is large and sparse the MINOS package (see Murtagh and Saunders [13]) should be used, since E04UCF treats all matrices as dense.

The user must supply an initial estimate of the solution to (1), together with subroutines that define \( F(x) \), \( c(x) \) and as many first partial derivatives as possible; unspecified derivatives are approximated by finite differences.

The objective function is defined by subroutine OBJFUN, and the nonlinear constraints are defined by subroutine CONFUN. On every call, these subroutines must return appropriate values of the objective and nonlinear constraints. The user should also provide the available partial derivatives. Any unspecified derivatives are approximated by finite differences; see Section 5.1 for a discussion of the optional parameter Derivative Level. Just before either OBJFUN or CONFUN is called, each element of the current gradient array OBJGRD or CJAC is initialised to a special value. On exit, any element that retains the value is estimated by finite differences. Note that if there are nonlinear constraints, then the first call to CONFUN will precede the first call to OBJFUN.

For maximum reliability, it is preferable for the user to provide all partial derivatives (see Chapter 8 of Gill et al [10], for a detailed discussion). If all gradients cannot be provided, it is similarly advisable to provide as many as possible. While developing the subroutines OBJFUN and CONFUN, the optional parameter Verify (see Section 5.1) should be used to check the calculation of any known gradients.

E04UCF implements a sequential quadratic programming (SQP) method. The document for E04NCF(*) should be consulted in conjunction with this document.

In the rest of this section we briefly summarize the main features of the method of E04UCF. Where possible, explicit reference is made to the names of variables that are parameters of subroutines E04UCF or appear in the printed output (see Section 5.2).

At a solution of (1), some of the constraints will be active, i.e., satisfied exactly. An active simple bound constraint
implies that the corresponding variable is fixed at its bound, and hence the variables are partitioned into fixed and free variables. Let \( C \) denote the \( m \times n \) matrix of gradients of the active general linear and nonlinear constraints. The number of fixed variables will be denoted by \( n_F \), with \( n = n_F + n_{FR} \) the number of free variables. The subscripts \('FX'\) and \('FR'\) on a vector or matrix will denote the vector or matrix composed of the components corresponding to fixed or free variables.

A point \( x \) is a first-order Kuhn-Tucker point for (1) (see, e.g., Powell [14]) if the following conditions hold:

(i) \( x \) is feasible;

(ii) there exist vectors \((x_i)\) and \((\lambda)\) (the Lagrange multiplier vectors for the bound and general constraints) such that

\[
T \quad g = C (\lambda) + (x_i), \tag{2}
\]

where \( g \) is the gradient of \( F \) evaluated at \( x \), and \((x_i)_j = 0\) if the \( j \)th variable is free.

(iii) The Lagrange multiplier corresponding to an inequality constraint active at its lower bound must be non-negative, and non-positive for an inequality constraint active at its upper bound.

Let \( Z \) denote a matrix whose columns form a basis for the set of vectors orthogonal to the rows of \( C \); i.e., \( C Z = 0 \). An equivalent statement of the condition (2) in terms of \( Z \) is

\[
T \quad Z g_{FP} = 0. \tag{3}
\]

The vector \( Z g_{FP} \) is termed the projected gradient of \( F \) at \( x \).

Certain additional conditions must be satisfied in order for a first-order Kuhn-Tucker point to be a solution of (1) (see, e.g., Powell [14]).

The method of E04UCF is a sequential quadratic programming (SQP) method. For an overview of SQP methods, see, for example, Fletcher [5], Gill et al [10] and Powell [15].

The basic structure of E04UCF involves major and minor
iterations. The major iterations generate a sequence of iterates $\{x_k\}$ that converge to $x$, a first-order Kuhn-Tucker point of (1).

At a typical major iteration, the new iterate $x$ is defined by

$$
    x = x + (\alpha)p
$$

(3)

where $x$ is the current iterate, the non-negative scalar $(\alpha)$ is the step length, and $p$ is the search direction. (For simplicity, we shall always consider a typical iteration and avoid reference to the index of the iteration.) Also associated with each major iteration are estimates of the Lagrange multipliers and a prediction of the active set.

The search direction $p$ in (3) is the solution of a quadratic programming subproblem of the form

$$
    \begin{array}{c}
    \text{Minimize} & g^T p^+ - p^T Hp, \\ 
    \text{subject to} & l^B \leq \{A p\} \leq u, \\
    \end{array}
$$

(4)

where $g$ is the gradient of $F$ at $x$, the matrix $H$ is a positive-definite quasi-Newton approximation to the Hessian of the Lagrangian function (see Section 8.3), and $A$ is the Jacobian matrix of $c$ evaluated at $x$. (Finite-difference estimates may be used for $g$ and $A$; see the optional parameter Derivative Level in Section 5.1.) Let $l$ in (1) be partitioned into three sections: $l^B$, $l^L$, and $l^N$, corresponding to the bound, linear and nonlinear constraints. The vector $l$ in (4) is similarly partitioned, and is defined as

$$
\begin{align*}
\bar{l}^B &= l^B - x, \\
\bar{l}^L &= l^L - A x, \\
\bar{l}^N &= l^N - c,
\end{align*}
$$

where $c$ is the vector of nonlinear constraints evaluated at $x$.

The vector $u$ is defined in an analogous fashion.

The estimated Lagrange multipliers at each major iteration are the Lagrange multipliers from the subproblem (4) (and similarly...
for the predicted active set). (The numbers of bounds, general linear and nonlinear constraints in the QP active set are the quantities Bnd, Lin and Nln in the printed output of E04UCF.) In E04UCF, (4) is solved using E04NCF(*). Since solving a quadratic program as an iterative procedure, the minor iterations of E04UCF are the iterations of E04NCF(*). (More details about solving the subproblem are given in Section 8.1.)

Certain matrices associated with the QP subproblem are relevant in the major iterations. Let the subscripts 'FX' and 'FR' refer to the predicted fixed and free variables, and let C denote the m by n matrix of gradients of the general linear and nonlinear constraints in the predicted active set. First, we have available the TQ factorization of C:

\[
C_{FR} Q_{FR} = (0 \ T), \tag{5}
\]

where \( T \) is a nonsingular m by m reverse-triangular matrix (i.e., \( t_{ij} = 0 \) if \( i+j<m \)), and the non-singular n by n matrix \( Q \) is the product of orthogonal transformations (see Gill et al [6]). Second, we have the upper-triangular Cholesky factor \( R \) of the transformed and re-ordered Hessian matrix

\[
R_{FR} R_{FR} = Q_{FR} H Q_{FR}, \tag{6}
\]

where \( H \) is the Hessian matrix \( H \) with rows and columns permuted so that the free variables are first, and \( Q \) is the n by n matrix

\[
Q = \begin{pmatrix}
(Q_{FX}) \\
(FR)
\end{pmatrix}, \tag{7}
\]

with \( I \) the identity matrix of order n. If the columns of \( Q \) are partitioned so that

\[
Q_{FR} = (Z_{FR}) \tag{7}
\]

the n (n == n - m) columns of \( Z \) form a basis for the null space of

\[
T_{FR} C_{FR}. \tag{8}
\]

The matrix \( Z \) is used to compute the projected gradient \( Z g \).
at the current iterate. (The values $N_z$, Norm $g_f$ and Norm $g_z$
printed by E04UCF give $n$ and the norms of $g$ and $Z g_f$.)

A theoretical characteristic of SQP methods is that the predicted active set from the QP subproblem (4) is identical to the correct active set in a neighbourhood of $x$. In E04UCF, this feature is exploited by using the QP active set from the previous iteration as a prediction of the active set for the next QP subproblem, which leads in practice to optimality of the subproblems in only one iteration as the solution is approached. Separate treatment of bound and linear constraints in E04UCF also saves computation in factorizing $C$ and $H$.

Once $p$ has been computed, the major iteration proceeds by determining a step length (alpha) that produces a 'sufficient decrease' in an augmented Lagrangian merit function (see Section 8.2). Finally, the approximation to the transformed Hessian matrix $H$ is updated using a modified BFGS quasi-Newton update (see Section 8.3) to incorporate new curvature information obtained in the move from $x$ to $\bar{x}$.

On entry to E04UCF, an iterative procedure from E04NCF(*) is executed, starting with the user-provided initial point, to find a point that is feasible with respect to the bounds and linear constraints (using the tolerance specified by Linear Feasibility Tolerance see Section 5.1). If no feasible point exists for the bound and linear constraints, (1) has no solution and E04UCF terminates. Otherwise, the problem functions will thereafter be evaluated only at points that are feasible with respect to the bounds and linear constraints. The only exception involves variables whose bounds differ by an amount comparable to the finite-difference interval (see the discussion of Difference Interval in Section 5.1). In contrast to the bounds and linear constraints, it must be emphasised that the nonlinear constraints will not generally be satisfied until an optimal point is reached.

Facilities are provided to check whether the user-provided gradients appear to be correct (see the optional parameter Verify in Section 5.1). In general, the check is provided at the first point that is feasible with respect to the linear constraints and bounds. However, the user may request that the check be performed at the initial point.
In summary, the method of E04UCF first determines a point that satisfies the bound and linear constraints. Thereafter, each iteration includes:

(a) the solution of a quadratic programming subproblem;

(b) a linesearch with an augmented Lagrangian merit function; and

(c) a quasi-Newton update of the approximate Hessian of the Lagrangian function.

These three procedures are described in more detail in Section 8.

4. References


5. Parameters

1: N -- INTEGER          Input
   On entry: the number, n, of variables in the problem.
   Constraint: N > 0.

2: NCLIN -- INTEGER       Input
   On entry: the number, n, of general linear constraints in
   the problem. Constraint: NCLIN >= 0.

3: NCNLN -- INTEGER       Input
   On entry: the number, n, of nonlinear constraints in the
   problem. Constraint: NCNLN >= 0.

4: NROWA -- INTEGER        Input
   On entry:
   the first dimension of the array A as declared in the
   (sub)program from which E04UCF is called.

    Theoretical Properties of an Augmented Lagrangian Merit
    Function. Report SOL 86-6R. Department of Operations
    Research, Stanford University.


    Programming Codes. Lecture Notes in Economics and

    Basic Linear Algebra Subprograms for Fortran Usage. ACM
    Trans. Math. Softw. 5 308--325.

    Report SOL 83-20. Department of Operations Research,
    Stanford University.

    Optimization. Numerical Methods for Constrained
    Optimization. (ed P E Gill and W Murray) Academic Press. 1--
    28.

    Optimization. Mathematical Programming: The State of the
    Art. (ed A Bachem, M Groetschel and B Korte) Springer-
    Verlag. 288--311.
Constraint: NROWA >= max(1,NCLIN).

5: NROWJ -- INTEGER Input
On entry:
the first dimension of the array CJAC as declared in the
(sub)program from which E04UCF is called.
Constraint: NROWJ >= max(1,NCNLN).

6: NROWR -- INTEGER Input
On entry:
the first dimension of the array R as declared in the
(sub)program from which E04UCF is called.
Constraint: NROWR >= N.

7: A(NROWA,*) -- DOUBLE PRECISION array Input
The second dimension of the array A must be >= N for NCLIN > 0. On entry: the ith row of the array A must contain the ith row of the matrix A of general linear constraints in (1).
That is, the ith row contains the coefficients of the ith general linear constraint, for i = 1,2,...,NCLIN.
If NCLIN = 0 then the array A is not referenced.

8: BL(N+NCLIN+NCNLN) -- DOUBLE PRECISION array Input
On entry: the lower bounds for all the constraints, in the following order. The first n elements of BL must contain the lower bounds on the variables. If NCLIN > 0, the next n elements of BL must contain the lower bounds on the general linear constraints. If NCNLN > 0, the next n elements of BL must contain the lower bounds for the nonlinear constraints. To specify a non-existent lower bound (i.e., l = -infty), the value used must satisfy BL(j) <= -BIGBND, where BIGBND is the value of the optional parameter Infinite Bound Size whose default value is 10 (see Section 5.1). To specify the jth constraint as an equality, the user must set BL(j) = BU(j) = (beta), say, where |(beta)| < BIGBND. Constraint: BL(j) <= BU(j), for j = 1,...,N+NCLIN+NCNLN.

9: BU(N+NCLIN+NCNLN) -- DOUBLE PRECISION array Input
On entry: the upper bounds for all the constraints in the following order. The first n elements of BU must contain the upper bounds on the variables. If NCLIN > 0, the next n elements of BU must contain the upper bounds on the general linear constraints. If NCNLN > 0, the next n elements of BU
must contain the upper bounds for the nonlinear constraints. To specify a non-existent upper bound (i.e., \( u = +\infty \)), the value used must satisfy \( B(j) \geq BIGBND \), where \( BIGBND \) is the value of the optional parameter Infinite Bound Size, whose default value is 10 (see Section 5.1). To specify the jth constraint as an equality, the user must set \( B(j) = BL(j) = (\beta) \), say, where \( |(\beta)| < BIGBND \). Constraint: \( B(j) \geq BL(j) \), for \( j=1,2,\ldots,N+NCLIN+NCNLN \).

10: \textbf{CONFUN} -- SUBROUTINE, supplied by the user. 

External Procedure

\texttt{CONFUN} must calculate the vector \( c(x) \) of nonlinear constraint functions and (optionally) its Jacobian for a specified \( n \) element vector \( x \). If there are no nonlinear constraints (\( NCNLN=0 \)), \texttt{CONFUN} will never be called by \texttt{E04UCF} and \texttt{CONFUN} may be the dummy routine \texttt{E04UDM}. (\texttt{E04UDM} is included in the NAG Foundation Library and so need not be supplied by the user. Its name may be implementation-dependent: see the Users’ Note for your implementation for details.) If there are nonlinear constraints, the first call to \texttt{CONFUN} will occur before the first call to \texttt{OBJFUN}.

Its specification is:

\begin{verbatim}
SUBROUTINE CONFUN (MODE, NCNLN, N, NROWJ, NEEDC, 
1   X, C, CJAC, NSTATE, IUSER, 
2   USER)
INTEGER MODE, NCNLN, N, NROWJ, NEEDC 
1   (NCNLN), NSTATE, IUSER(*) 
DOUBLE PRECISION X(N), C(NCNLN), CJAC(NROWJ,N), 
1   USER(*)
\end{verbatim}

1: MODE -- INTEGER

Input/Output

On entry: \texttt{MODE} indicates the values that must be assigned during each call of \texttt{CONFUN}. \texttt{MODE} will always have the value 2 if all elements of the Jacobian are available, i.e., if Derivative Level is either 2 or 3 (see Section 5.1). If some elements of \texttt{CJAC} are unspecified, \texttt{E04UCF} will call \texttt{CONFUN} with \texttt{MODE} = 0, 1, or 2:

If \texttt{MODE} = 2, only the elements of \texttt{C} corresponding to positive values of \texttt{NEEDC} must be set (and similarly for the available components of the rows of \texttt{CJAC}).

If \texttt{MODE} = 1, the available components of the rows of \texttt{CJAC} corresponding to positive values in \texttt{NEEDC} must be
set. Other rows of CJAC and the array C will be ignored.

If MODE = 0, the components of C corresponding to positive values in NEEDC must be set. Other components and the array CJAC are ignored. On exit: MODE may be set to a negative value if the user wishes to terminate the solution to the current problem. If MODE is negative on exit from CONFUN then E04UCF will terminate with IFAIL set to MODE.

2: NCNLN -- INTEGER Input
   On entry: the number, n, of nonlinear constraints.

N

3: N -- INTEGER Input
   On entry: the number, n, of variables.

4: NROWJ -- INTEGER Input
   On entry: the first dimension of the array CJAC.

5: NEEDC(NCNLN) -- INTEGER array Input
   On entry: the indices of the elements of C or CJAC that must be evaluated by CONFUN. If NEEDC(i)>0 then the ith element of C and/or the ith row of CJAC (see parameter MODE above) must be evaluated at x.

6: X(N) -- DOUBLE PRECISION array Input
   On entry: the vector x of variables at which the constraint functions are to be evaluated.

7: C(NCNLN) -- DOUBLE PRECISION array Output
   On exit: if NEEDC(i)>0 and MODE = 0 or 2, C(i) must contain the value of the ith constraint at x. The remaining components of C, corresponding to the non-positive elements of NEEDC, are ignored.

8: CJAC(NROWJ,N) -- DOUBLE PRECISION array Output
   On exit: if NEEDC(i)>0 and MODE = 1 or 2, the ith row of CJAC must contain the available components of the vector (nabla)c given by

   \[ (\text{nabla}c) = \left( \frac{\partial c_1}{\partial x_1}, \frac{\partial c_1}{\partial x_2}, \ldots, \frac{\partial c_1}{\partial x_n} \right) \]

   \[ \left( \frac{\partial c_2}{\partial x_1}, \frac{\partial c_2}{\partial x_2}, \ldots, \frac{\partial c_2}{\partial x_n} \right) \]

   \[ \vdots \]

   \[ \left( \frac{\partial c_n}{\partial x_1}, \frac{\partial c_n}{\partial x_2}, \ldots, \frac{\partial c_n}{\partial x_n} \right) \]

   where \(-\)- is the partial derivative of the ith
constraint with respect to the $j$th variable, evaluated at the point $x$. See also the parameter NSTATE below. The remaining rows of CJAC, corresponding to non-positive elements of NEEDC, are ignored.

If all constraint gradients (Jacobian elements) are known (i.e., Derivative Level = 2 or 3; see Section 5.1) any constant elements may be assigned to CJAC one time only at the start of the optimization. An element of CJAC that is not subsequently assigned in CONFUN will retain its initial value throughout. Constant elements may be loaded into CJAC either before the call to E04UCF or during the first call to CONFUN (signalled by the value NSTATE = 1). The ability to preload constants is useful when many Jacobian elements are identically zero, in which case CJAC may be initialised to zero and non-zero elements may be reset by CONFUN.

Note that constant non-zero elements do affect the values of the constraints. Thus, if CJAC$(i,j)$ is set to a constant value, it need not be reset in subsequent calls to CONFUN, but the value CJAC$(i,j)\times X(j)$ must nonetheless be added to C$(i)$.

It must be emphasized that, if Derivative Level < 2, unassigned elements of CJAC are not treated as constant; they are estimated by finite differences, at non-trivial expense. If the user does not supply a value for Difference Interval (see Section 5.1), an interval for each component of $x$ is computed automatically at the start of the optimization. The automatic procedure can usually identify constant elements of CJAC, which are then computed once only by finite differences.

9: NSTATE -- INTEGER
   On entry: if NSTATE = 1 then E04UCF is calling CONFUN for the first time. This parameter setting allows the user to save computation time if certain data must be read or calculated only once.

10: IUSER(*) -- INTEGER array
    User Workspace

11: USER(*) -- DOUBLE PRECISION array
    User Workspace
    CONFUN is called from E04UCF with the parameters IUSER and USER as supplied to E04UCF. The user is free to use the arrays IUSER and USER to supply information to CONFUN as an alternative to using COMMON.
CONFUN must be declared as EXTERNAL in the (sub)program from which E04UCF is called. Parameters denoted as Input must not be changed by this procedure.

11: OBJFUN -- SUBROUTINE, supplied by the user.

External Procedure

OBJFUN must calculate the objective function \( F(x) \) and (optionally) the gradient \( g(x) \) for a specified \( n \) element vector \( x \).

Its specification is:

```
SUBROUTINE OBJFUN (MODE, N, X, OBJF, OBJGRD, NSTATE, IUSER, USER)
INTEGER MODE, N, NSTATE, IUSER(*)
DOUBLE PRECISION X(N), OBJF, OBJGRD(N), USER(*)
```

1: MODE -- INTEGER Input/Output

On entry: MODE indicates the values that must be assigned during each call of OBJFUN.

MODE will always have the value 2 if all components of the objective gradient are specified by the user, i.e., if Derivative Level is either 1 or 3. If some gradient elements are unspecified, E04UCF will call OBJFUN with MODE = 0, 1 or 2.

If MODE = 2, compute OBJF and the available components of OBJGRD.

If MODE = 1, compute all available components of OBJGRD; OBJF is not required.

If MODE = 0, only OBJF needs to be computed; OBJGRD is ignored.

On exit: MODE may be set to a negative value if the user wishes to terminate the solution to the current problem. If MODE is negative on exit from OBJFUN, then E04UCF will terminate with IFAIL set to MODE.

2: N -- INTEGER Input

On entry: the number, \( n \), of variables.

3: X(N) -- DOUBLE PRECISION array Input

On entry: the vector \( x \) of variables at which the objective function is to be evaluated.

4: OBJF -- DOUBLE PRECISION Output

On exit: if MODE = 0 or 2, OBJF must be set to the value of the objective function at \( x \).
5: OBJGRD(N) -- DOUBLE PRECISION array
On exit: if MODE = 1 or 2, OBJGRD must return the
available components of the gradient evaluated at x.

6: NSTATE -- INTEGER
Input
On entry: if NSTATE = 1 then E04UCF is calling OBJFUN
for the first time. This parameter setting allows the
user to save computation time if certain data must be
read or calculated only once.

7: IUSER(*) -- INTEGER array
User Workspace

8: USER(*) -- DOUBLE PRECISION array
User Workspace
OBJFUN is called from E04UCF with the parameters IUSER
and USER as supplied to E04UCF. The user is free to use
the arrays IUSER and USER to supply information to
OBJFUN as an alternative to using COMMON.
OBJFUN must be declared as EXTERNAL in the (sub)program
from which E04UCF is called. Parameters denoted as
Input must not be changed by this procedure.

12: ITER -- INTEGER
Output
On exit: the number of iterations performed.

13: ISTATE(N+NCLIN+NCNLN) -- INTEGER array
Input/Output
On entry: ISTATE need not be initialised if E04UCF is called
with (the default) Cold Start option. The ordering of ISTATE
is as follows. The first n elements of ISTATE refer to the
upper and lower bounds on the variables, elements n+1
through n+n refer to the upper and lower bounds on A x, and
L
elements n+n +1 through n+n +n refer to the upper and lower
L L N
bounds on c(x). When a Warm Start option is chosen, the
elements of ISTATE corresponding to the bounds and linear
constraints define the initial working set for the procedure
that finds a feasible point for the linear constraints and
bounds. The active set at the conclusion of this procedure
and the elements of ISTATE corresponding to nonlinear
constraints then define the initial working set for the
first QP subproblem. Possible values for ISTATE(j) are:

ISTATE(j) Meaning
0 The corresponding constraint is not in the initial
QP working set.
1 This inequality constraint should be in the
working set at its lower bound.
2 This inequality constraint should be in the working set at its upper bound.

3 This equality constraint should be in the initial working set. This value must not be specified unless $BL(j) = BU(j)$. The values 1, 2 or 3 all have the same effect when $BL(j) = BU(j)$.

Note that if E04UCF has been called previously with the same values of $N$, NCLIN and NCNLN, ISTATE already contains satisfactory values. If necessary, E04UCF will override the user's specification of ISTATE so that a poor choice will not cause the algorithm to fail. On exit: with IFAIL = 0 or 1, the values in the array ISTATE correspond to the active set of the final QP subproblem, and are a prediction of the status of the constraints at the solution of the problem. Otherwise, ISTATE indicates the composition of the QP working set at the final iterate. The significance of each possible value of ISTATE(j) is as follows:

-2 This constraint violates its lower bound by more than the appropriate feasibility tolerance (see the optional parameters LinearFeasibility Tolerance and Nonlinear Feasibility Tolerance in Section 5.1). This value can occur only when no feasible point can be found for a QP subproblem.

-1 This constraint violates its upper bound by more than the appropriate feasibility tolerance (see the optional parameters LinearFeasibility Tolerance and Nonlinear Feasibility Tolerance in Section 5.1). This value can occur only when no feasible point can be found for a QP subproblem.

0 The constraint is satisfied to within the feasibility tolerance, but is not in the working set.

1 This inequality constraint is included in the QP working set at its upper bound.

2 This inequality constraint is included in the QP working set at its upper bound.

3 This constraint is included in the QP working set as an equality. This value of ISTATE can occur only when $BL(j) = BU(j)$.

14: C(*) -- DOUBLE PRECISION array Output

Note: the dimension of the array C must be at least $\max(1,NCNLN)$.

On exit: if $NCNLN > 0$, C(i) contains the value of the ith
nonlinear constraint function $c_i$ at the final iterate, for $i = 1, 2, \ldots, NCNLN$. If $NCNLN = 0$, then the array $C$ is not referenced.

15: CJAC(NROWJ,*): DOUBLE PRECISION array  Input/Output
Note: the second dimension of the array CJAC must be at least $N$ for $NCNLN > 0$ and 1 otherwise On entry: in general, CJAC need not be initialised before the call to E04UCF. However, if Derivative Level $= 3$, the user may optionally set the constant elements of CJAC (see parameter NSTATE in the description of CONFUN). Such constant elements need not be re-assigned on subsequent calls to CONFUN. If $NCNLN = 0$, then the array CJAC is not referenced. On exit: if $NCNLN > 0$, CJAC contains the Jacobian matrix of the nonlinear constraint functions at the final iterate, i.e., $CJAC(i,j)$ contains the partial derivative of the $i$th constraint function with respect to the $j$th variable, for $i = 1, 2, \ldots, NCNLN; j = 1, 2, \ldots, N$. (See the discussion of parameter CJAC under CONFUN.)

16: CLAMDA(N+NCLIN+NCNLN): DOUBLE PRECISION array  Input/Output
On entry: CLAMDA need not be initialised if E04UCF is called with the (default) Cold Start option. With the Warm Start option, CLAMDA must contain a multiplier estimate for each nonlinear constraint with a sign that matches the status of the constraint specified by the ISTATE array (as above). The ordering of CLAMDA is as follows; the first $n$ elements contain the multipliers for the bound constraints on the variables, elements $n+1$ through $n+n$ contain the multipliers for the general linear constraints, and elements $n+n +1$ through $n+n +n$ contain the multipliers for the nonlinear constraints. If the $j$th constraint is defined as 'inactive' by the initial value of the ISTATE array, CLAMDA($j$) should be zero; if the $j$th constraint is an inequality active at its lower bound, CLAMDA($j$) should be non-negative; if the $j$th constraint is an inequality active at its upper bound, CLAMDA($j$) should be non-positive. On exit: the values of the QP multipliers from the last QP subproblem. CLAMDA($j$) should be non-negative if ISTATE($j$) = 1 and non-positive if ISTATE($j$) = 2.

17: OBJF: DOUBLE PRECISION  Output
On exit: the value of the objective function, $F(x)$, at the final iterate.

18: OBJGRD(N): DOUBLE PRECISION array  Output
On exit: the gradient (or its finite-difference approximation) of the objective function at the final iterate.

19: R(NROWR,N) -- DOUBLE PRECISION array Input/Output
On entry: R need not be initialised if E04UCF is called with a Cold Start option (the default), and will be taken as the identity. With a Warm Start R must contain the upper-triangular Cholesky factor R of the initial approximation of the Hessian of the Lagrangian function, with the variables in the natural order. Elements not in the upper-triangular part of R are assumed to be zero and need not be assigned.
On exit: if Hessian = No, (the default; see Section 5.1), R
contains the upper-triangular Cholesky factor R of Q HQ, an estimate of the transformed and re-ordered Hessian of the Lagrangian at x (see (6) in Section 3). If Hessian = Yes, R contains the upper-triangular Cholesky factor R of H, the approximate (untransformed) Hessian of the Lagrangian, with the variables in the natural order.

20: X(N) -- DOUBLE PRECISION array Input/Output
On entry: an initial estimate of the solution. On exit: the final estimate of the solution.

21: IWORK(LIWORK) -- INTEGER array Workspace

22: LIWORK -- INTEGER Input
On entry: the dimension of the array IWORK as declared in the (sub)program from which E04UCF is called.
Constraints: LIWORK>=3*N+NCLIN+N*NCNLN.

23: WORK(LWORK) -- DOUBLE PRECISION array Workspace

24: LWORK -- INTEGER Input
On entry: the dimension of the array WORK as declared in the (sub)program from which E04UCF is called.
Constraints:
if NCLIN = NCNLN = 0 then
LWORK >=20*N
if NCNLN = 0 and NCLIN > 0 then
2
LWORK >=2*N +20*N+11*NCLIN
if NCNLN > 0 and NCLIN >= 0 then
2
LWORK>=2*N +N*NCLIN+20*N+NCNLN+20*N+ 11*NCNLIN+21*NCNLN
If Major Print Level > 0, the required amounts of workspace are output on the current advisory message channel (see X04ABF). As an alternative to computing LIWORK and LWORK from the formulas given above, the user may prefer to obtain appropriate values from the output of a preliminary run with a positive value of Major Print Level and LIWORK and LWORK set to 1. (E04UCF will then terminate with IFAIL = 9.)

25: IUSER(*) -- INTEGER array User Workspace
Note: the dimension of the array IUSER must be at least 1. IUSER is not used by E04UCF, but is passed directly to routines CONFUN and OBJFUN and may be used to pass information to those routines.

26: USER(*) -- DOUBLE PRECISION array User Workspace
Note: the dimension of the array USER must be at least 1. USER is not used by E04UCF, but is passed directly to routines CONFUN and OBJFUN and may be used to pass information to those routines.

27: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.

On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

E04UCF returns with IFAIL = 0 if the iterates have converged to a point x that satisfies the first-order Kuhn-Tucker conditions to the accuracy requested by the optional parameter Optimality Tolerance (see Section 5.1), i.e., the projected gradient and active constraint residuals are negligible at x.

The user should check whether the following four conditions are satisfied:
(i) the final value of Norm Gz is significantly less than that at the starting point;
(ii) during the final major iterations, the values of Step and ItQP are both one;
(iii) the last few values of both Norm Gz and Norm C become
small at a fast linear rate;

(iv) Cond Hz is small.

If all these conditions hold, x is almost certainly a local minimum of (1). (See Section 9 for a specific example.)

5.1. Optional Input Parameters

Several optional parameters in E04UCF define choices in the behaviour of the routine. In order to reduce the number of formal parameters of E04UCF these optional parameters have associated default values (see Section 5.1.3) that are appropriate for most problems. Therefore the user need only specify those optional parameters whose values are to be different from their default values.

The remainder of this section can be skipped by users who wish to use the default values for all optional parameters. A complete list of optional parameters and their default values is given in Section 5.1.3

5.1.1. Specification of the optional parameters

Optional parameters may be specified by calling one, or both, of E04UDF and E04UEF prior to a call to E04UCF.

E04UDF reads options from an external options file, with Begin and End as the first and last lines respectively and each intermediate line defining a single optional parameter. For example,

```
Begin
  Print Level = 1
End
```

The call

```
CALL E04UDF (IOPTNS, INFORM)
```

can then be used to read the file on unit IOPTNS. INFORM will be zero on successful exit. E04UDF should be consulted for a full description of this method of supplying optional parameters.

E04UEF can be called directly to supply options, one call being necessary for each optional parameter. For example,

```
CALL E04UEF ('Print level = 1')
```

E04UEF should be consulted for a full description of this method of supplying optional parameters.
All optional parameters not specified by the user are set to their default values. Optional parameters specified by the user are unaltered by E04UCF (unless they define invalid values) and so remain in effect for subsequent calls to E04UCF, unless altered by the user.

5.1.2. Description of the optional parameters

The following list (in alphabetical order) gives the valid options. For each option, we give the keyword, any essential optional qualifiers, the default value, and the definition. The minimum valid abbreviation of each keyword is underlined. If no characters of an optional qualifier are underlined, the qualifier may be omitted. The letter a denotes a phrase (character string) that qualifies an option. The letters i and r denote INTEGER and DOUBLE PRECISION values required with certain options. The number (epsilon) is a generic notation for machine precision (see X02AJF(*)), and (epsilon) denotes the relative precision of the objective function (the optional parameter Function Precision see below).

Central Difference Interval r Default values are computed

If the algorithm switches to central differences because the forward-difference approximation is not sufficiently accurate, the value of r is used as the difference interval for every component of x. The use of finite-differences is discussed further below under the optional parameter Difference Interval.

Cold Start Default = Cold Start

Warm Start

(AXIOM parameter STA, warm start when .TRUE.)

This option controls the specification of the initial working set in both the procedure for finding a feasible point for the linear constraints and bounds, and in the first QP subproblem thereafter. With a Cold Start, the first working set is chosen by E04UCF based on the values of the variables and constraints at the initial point. Broadly speaking, the initial working set will include equality constraints and bounds or inequality constraints that violate or 'nearly' satisfy their bounds (within Crash Tolerance; see below). With a Warm Start, the user must set the ISTATE array and define CLAMDA and R as discussed in Section 5. ISTATE values associated with bounds and linear constraints determine the initial working set of the procedure to find a feasible point with respect to the bounds and linear constraints.
ISTATE values associated with nonlinear constraints determine the initial working set of the first QP subproblem after such a feasible point has been found. E04UCF will override the user's specification of ISTATE if necessary, so that a poor choice of the working set will not cause a fatal error. A warm start will be advantageous if a good estimate of the initial working set is available - for example, when E04UCF is called repeatedly to solve related problems.

Crash Tolerance $r$ Default = 0.01

(AXIOM parameter CRA)

This value is used in conjunction with the optional parameter Cold Start (the default value). When making a cold start, the QP algorithm in E04UCF must select an initial working set. When $r>=0$, the initial working set will include (if possible) bounds or general inequality constraints that lie within $r$ of their bounds. In particular, a constraint of the form $a^T x >= l$ will be included if $|a^T x - l| <= r(1+|l|)$. If $r<0$ or $r>1$, the default value is used.

Defaults

This special keyword may be used to reset the default values following a call to E04UCF.

Derivative Level $i$ Default = 3

(AXIOM parameter DER)

This parameter indicates which derivatives are provided by the user in subroutines OBJFUN and CONFUN. The possible choices for $i$ are the following.

1. Meaning
2. All objective and constraint gradients are provided by the user.
3. All of the Jacobian is provided, but some components of the objective gradient are not specified by the user.
4. All elements of the objective gradient are known, but some elements of the Jacobian matrix are not specified by the user.
Some elements of both the objective gradient and the Jacobian matrix are not specified by the user. The value $i=3$ should be used whenever possible, since E04UCF is more reliable and will usually be more efficient when all derivatives are exact.

If $i=0$ or 2, E04UCF will estimate the unspecified components of the objective gradient, using finite differences. The computation of finite-difference approximations usually increases the total run-time, since a call to OBJFUN is required for each unspecified element. Furthermore, less accuracy can be attained in the solution (see Chapter 8 of Gill et al [10], for a discussion of limiting accuracy).

If $i=0$ or 1, E04UCF will approximate unspecified elements of the Jacobian. One call to CONFUN is needed for each variable for which partial derivatives are not available. For example, if the Jacobian has the form

\[
\begin{pmatrix}
* & * & * \\
* & ? & ? \\
* & * & ? \\
* & * & * \\
\end{pmatrix}
\]

where '*' indicates an element provided by the user and '?' indicates an unspecified element, E04UCF will call CONFUN twice: once to estimate the missing element in column 2, and again to estimate the two missing elements in column 3. (Since columns 1 and 4 are known, they require no calls to CONFUN.)

At times, central differences are used rather than forward differences, in which case twice as many calls to OBJFUN and CONFUN are needed. (The switch to central differences is not under the user's control.)

**Difference Interval $r$**

Default values are computed

This option defines an interval used to estimate gradients by finite differences in the following circumstances:

(a) For verifying the objective and/or constraint gradients (see the description of Verify, below).

(b) For estimating unspecified elements of the objective gradient of the Jacobian matrix.
In general, a derivative with respect to the jth variable is approximated using the interval \((\delta_j)\), where \(\delta_j = r(1 + |x_j|)\) with \(x_j\) the first point feasible with respect to the bounds and linear constraints. If the functions are well scaled, the resulting derivative approximation should be accurate to \(O(r)\).

See Gill et al [10] for a discussion of the accuracy in finite-difference approximations.

If a difference interval is not specified by the user, a finite-difference interval will be computed automatically for each variable by a procedure that requires up to six calls of CONFUN and OBJFUN for each component. This option is recommended if the function is badly scaled or the user wishes to have E04UCF determine constant elements in the objective and constraint gradients (see the descriptions of CONFUN and OBJFUN in Section 5).

**Feasibility Tolerance** \(r\)

**Default** = \(\sqrt{\text{eps}}\)

The scalar \(r\) defines the maximum acceptable absolute violations in linear and nonlinear constraints at a 'feasible' point; i.e., a constraint is considered satisfied if its violation does not exceed \(r\). If \(r < (\text{eps})\) or \(r \geq 1\), the default value is used. Using this keyword sets both optional parameters Linear Feasibility Tolerance and Nonlinear Feasibility Tolerance to \(r\), if \((\text{eps}) < r < 1\). (Additional details are given below under the descriptions of these parameters.)

**Function Precision** \(r\)

**Default** = \(\text{eps}\)

This parameter defines \(\text{eps}\), which is intended to be a measure of the accuracy with which the problem functions \(f\) and \(c\) can be computed. If \(r < (\text{eps})\) or \(r \geq 1\), the default value is used. The value of \(\text{eps}\) should reflect the relative precision of \(1 + |F(x)|\); i.e., \(\text{eps}\) acts as a relative precision when \(|F|\) is large, and as an absolute precision when \(|F|\) is small. For example, if \(F(x)\) is typically of order 1000 and
the first six significant digits are known to be correct, an appropriate value for \((\epsilon)\) would be \(1.0\times 10^{-6}\). In contrast, if \(F(x)\) is typically of order 10 and the first six significant digits are known to be correct, an appropriate value for \((\epsilon)\) would be \(1.0\times 10^{-10}\). The choice of \((\epsilon)\) can be quite complicated for badly scaled problems; see Chapter 8 of Gill et al [10] for a discussion of scaling techniques. The default value is appropriate for most simple functions that are computed with full accuracy. However, when the accuracy of the computed function values is known to be significantly worse than full precision, the value of \((\epsilon)\) should be large enough so that E04UCF will not attempt to distinguish between function values that differ by less than the error inherent in the calculation.

Hessian No Default = No

Hessian Yes

(No AXIOM parameter - fixed as Yes)

This option controls the contents of the upper-triangular matrix \(R\) (see Section 5). E04UCF works exclusively with the transformed and re-ordered Hessian \(H\) (6), and hence extra computation is required to form the Hessian itself. If Hessian = No, \(R\) contains the Cholesky factor of the transformed and re-ordered Hessian. If Hessian = Yes the Cholesky factor of the approximate Hessian itself is formed and stored in \(R\). The user should select Hessian = Yes if a warm start will be used for the next call to E04UCF.

10

Infinite Bound Size \(r\) Default = 10

(AXIOM parameter INFB)

If \(r>0\), \(r\) defines the 'infinite' bound \(\text{BIGBND}\) in the definition of the problem constraints. Any upper bound greater than or equal to \(\text{BIGBND}\) will be regarded as plus infinity (and similarly for a lower bound less than or equal to \(-\text{BIGBND}\)). If \(r<=0\), the default value is used.

10

Infinite Step Size \(r\) Default = \(\max(\text{BIGBND},10)\)

(AXIOM parameter INFS)
If $r > 0$, $r$ specifies the magnitude of the change in variables that is treated as a step to an unbounded solution. If the change in $x$ during an iteration would exceed the value of Infinite Step Size, the objective function is considered to be unbounded below in the feasible region. If $r \leq 0$, the default value is used.

Iteration limit $i$ Default $= \max(50, 3(n + N) + 10n)$

See Major Iteration Limit below.

Linear Feasibility Tolerance $r$ Default $= \sqrt{\text{epsilon)}}$

(AXIOM parameter LINF)

Nonlinear Feasibility Tolerance $r_i$ Default $= \sqrt{\text{epsilon)}}$ if $i = 0.33$ 

(AXIOM parameter NONF)

Derivative Level $\geq 2$ and $(\text{epsilon)}$ otherwise

The scalars $r$ and $r_i$ define the maximum acceptable absolute violations in linear and nonlinear constraints at a 'feasible' point; i.e., a linear constraint is considered satisfied if its violation does not exceed $r$, and similarly for a nonlinear constraint and $r_i$. If $r < (\text{epsilon)}$ or $r_i \geq 1$, the default value is used, for $i = 1, 2$.

On entry to E04UCF, an iterative procedure is executed in order to find a point that satisfies the linear constraint and bounds on the variables to within the tolerance $r$. All subsequent iterates will satisfy the linear constraints to within the same tolerance (unless $r$ is comparable to the finite-difference interval).

For nonlinear constraints, the feasibility tolerance $r$ defines the largest constraint violation that is acceptable at an optimal point. Since nonlinear constraints are generally not satisfied until the final iterate, the value of Nonlinear Feasibility
Tolerance acts as a partial termination criterion for the iterative sequence generated by E04UCF (see the discussion of Optimality Tolerance).

These tolerances should reflect the precision of the corresponding constraints. For example, if the variables and the coefficients in the linear constraints are of order unity, and the latter are correct to about 6 decimal digits, it would be appropriate to specify $r$ as $10^{-6}$.

Linesearch Tolerance $r$ Default = 0.9

(AXIOM parameter LINT)

The value $r$ ($0 \leq r < 1$) controls the accuracy with which the step (alpha) taken during each iteration approximates a minimum of the merit function along the search direction (the smaller the value of $r$, the more accurate the linesearch). The default value $r=0.9$ requests an inaccurate search, and is appropriate for most problems, particularly those with any nonlinear constraints.

If there are no nonlinear constraints, a more accurate search may be appropriate when it is desirable to reduce the number of major iterations - for example, if the objective function is cheap to evaluate, or if a substantial number of gradients are unspecified.

List Default = List

Nolist

(AXIOM parameter LIST)

Normally each optional parameter specification is printed as it is supplied. Nolist may be used to suppress the printing and List may be used to restore printing.

Major Iteration Limit $i$ Default = $\max(50,3(n+n)+10n)$

Iteration Limit

Iters

Itns

(AXIOM parameter MAJI)
The value of $i$ specifies the maximum number of major iterations allowed before termination. Setting $i=0$ and Major Print Level> 0 means that the workspace needed will be computed and printed, but no iterations will be performed.

Major Print level $i$ Default = 10

Print Level

(AXIOM parameter MAJP)

The value of $i$ controls the amount of printout produced by the major iterations of E04UCF. (See also Minor Print level below.) The levels of printing are indicated below.

$i$ Output

0 No output.

1 The final solution only.

5 One line for each major iteration (no printout of the final solution).

$\geq 10$ The final solution and one line of output for each iteration.

$\geq 20$ At each major iteration, the objective function, the Euclidean norm of the nonlinear constraint violations, the values of the nonlinear constraints (the vector $c$), the values of the linear constraints (the vector $A x$), and the current values of the variables (the vector $x$).

$\geq 30$ At each major iteration, the diagonal elements of the matrix $T$ associated with the TQ factorization (5) of the QP working set, and the diagonal elements of $R$, the triangular factor of the transformed and re-ordered Hessian (6).

Minor Iteration Limit $i$ Default = max(50,3(n+n +n ))

L N

(AXIOM parameter MINI)

The value of $i$ specifies the maximum number of iterations for the optimality phase of each QP subproblem.

Minor Print Level $i$ Default = 0
The value of $i$ controls the amount of printout produced by the minor iterations of E04UCF, i.e., the iterations of the quadratic programming algorithm. (See also Major Print Level, above.) The following levels of printing are available.

- **$i = 0$**: No output.
- **$i = 1$**: The final QP solution.
- **$i = 5$**: One line of output for each minor iteration (no printout of the final QP solution).
- **$i \geq 10$**: The final QP solution and one brief line of output for each minor iteration.
- **$i \geq 20$**: At each minor iteration, the current estimates of the QP multipliers, the current estimate of the QP search direction, the QP constraint values, and the status of each QP constraint.
- **$i \geq 30$**: At each minor iteration, the diagonal elements of the matrix $T$ associated with the TQ factorization (5) of the QP working set, and the diagonal elements of the Cholesky factor $R$ of the transformed Hessian (6).

---

**Nonlinear Feasibility Tolerance $r$**

Default: $\sqrt{\text{epsilon}}$

See Linear Feasibility Tolerance, above.

**Optimality Tolerance $r$**

Default: $\text{epsilon}$

---

**AXIOM parameter MINP**

**AXIOM parameter OPT**

The parameter $r$ ($(\text{epsilon}) \leq r < 1$) specifies the accuracy to which the user wishes the final iterate to approximate a solution of the problem. Broadly speaking, $r$ indicates the number of correct figures desired in the objective function at the solution. For example, if $r$ is 10 and E04UCF terminates successfully, the final value of $F$ should have approximately six correct figures. If $r < (\text{epsilon})$ or $r \geq 1$ the default value is used.
E04UCF will terminate successfully if the iterative sequence of \( x \) -values is judged to have converged and the final point satisfies the first-order Kuhn-Tucker conditions (see Section 3). The sequence of iterates is considered to have converged at \( x \) if

\[
(\alpha) \|p\| \leq \sqrt{r(1+\|x\|)},
\]

where \( p \) is the search direction and \( (\alpha) \) the step length from (3). An iterate is considered to satisfy the first-order conditions for a minimum if

\[
\frac{\|Z g\|}{\|F(x)\|} \leq \sqrt{r(1+\max(1+|F(x)|,\|g\|))},
\]

and

\[
|\text{res}^j| \leq \text{ftol} \text{ for all } j,
\]

where \( Z g \) is the projected gradient (see Section 3), \( g \) is the gradient of \( F(x) \) with respect to the free variables, \( \text{res}^j \) is the violation of the \( j \)th active nonlinear constraint, and \( \text{ftol} \) is the Nonlinear Feasibility Tolerance.

Step Limit \( r \) Default = 2.0

(AXIOM parameter STE)

If \( r > 0 \), \( r \) specifies the maximum change in variables at the first step of the linesearch. In some cases, such as \( F(x) = ax \) or \( F(x) = bx \), even a moderate change in the components of \( x \) can lead to floating-point overflow. The parameter \( r \) is therefore used to encourage evaluation of the problem functions at meaningful points. Given any major iterate \( x \), the first point \( x \) at which \( F \) and \( c \) are evaluated during the linesearch is restricted so that

\[
\|x-x\| \leq r(1+\|x\|).
\]

The linesearch may go on and evaluate \( F \) and \( c \) at points further
from \( x \) if this will result in a lower value of the merit function. In this case, the character \( L \) is printed at the end of the optional line of printed output, (see Section 5.2). If \( L \) is printed for most of the iterations, \( r \) should be set to a larger value.

Wherever possible, upper and lower bounds on \( x \) should be used to prevent evaluation of nonlinear functions at wild values. The default value Step Limit = 2.0 should not affect progress on well-behaved functions, but values 0.1 or 0.01 may be helpful when rapidly varying functions are present. If a small value of Step Limit is selected, a good starting point may be required. An important application is to the class of nonlinear least-squares problems. If \( r \leq 0 \), the default value is used.

Start Objective Check At Variable \( k \) Default = 1

\[ \text{(AXIOM parameter STAO)} \]

Start Constraint Check At Variable \( k \) Default = 1

\[ \text{(AXIOM parameter STAC)} \]

Stop Objective Check At Variable \( l \) Default = \( n \)

\[ \text{(AXIOM parameter STOO)} \]

Stop Constraint Check At Variable \( l \) Default = \( n \)

\[ \text{(AXIOM parameter STOC)} \]

These keywords take effect only if Verify Level \( > 0 \) (see below). They may be used to control the verification of gradient elements computed by subroutines OBJFUN and CONFUN. For example, if the first 30 components of the objective gradient appeared to be correct in an earlier run, so that only component 31 remains questionable, it is reasonable to specify Start Objective Check At Variable 31. If the first 30 variables appear linearly in the objective, so that the corresponding gradient elements are constant, the above choice would also be appropriate.

Verify Level \( i \) Default = 0

Verify No

Verify Level - 1

Verify Level 0

Verify Objective Gradients
Verify Level 1
Verify Constraint Gradients
Verify Level 2
Verify
Verify Yes
Verify Gradients
Verify Level 3

(AXIOM parameter VE)

These keywords refer to finite-difference checks on the gradient elements computed by the user-provided subroutines OBJFUN and CONFUN. (Unspecified gradient components are not checked.) It is possible to specify Verify Levels 0-3 in several ways, as indicated above. For example, the nonlinear objective gradient (if any) will be verified if either Verify Objective Gradients or Verify Level 1 is specified. Similarly, the objective and the constraint gradients will be verified if Verify Yes or Verify Level 3 or Verify is specified.

If 0<=i<=3, gradients will be verified at the first point that satisfies the linear constraints and bounds. If i=0, only a 'cheap' test will be performed, requiring one call to OBJFUN and one call to CONFUN. If 1<=i<=3, a more reliable (but more expensive) check will be made on individual gradient components, within the ranges specified by the Start and Stop keywords described above. A result of the form OK or BAD? is printed by E04UCF to indicate whether or not each component appears to be correct.

If 10<=i<=13, the action is the same as for i - 10, except that it will take place at the user-specified initial value of x.

We suggest that Verify Level 3 be specified whenever a new function routine is being developed.

5.1.3. Optional parameter checklist and default values

For easy reference, the following list shows all the valid keywords and their default values. The symbol (epsilon) represents the machine precision (see X02AJF(*) ).

Optional Parameters Default Values
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Central difference interval</td>
<td>Computed automatically</td>
</tr>
<tr>
<td>Cold/Warm start</td>
<td>Cold start</td>
</tr>
<tr>
<td>Crash tolerance</td>
<td>0.01</td>
</tr>
<tr>
<td>Defaults</td>
<td></td>
</tr>
<tr>
<td>Derivative level</td>
<td>3</td>
</tr>
<tr>
<td>Difference interval</td>
<td>Computed automatically</td>
</tr>
<tr>
<td>Feasibility tolerance</td>
<td>$\sqrt{\epsilon}$</td>
</tr>
<tr>
<td>Function precision</td>
<td>0.9</td>
</tr>
<tr>
<td>Hessian</td>
<td>No</td>
</tr>
<tr>
<td>Infinite bound size</td>
<td>10</td>
</tr>
<tr>
<td>Infinite step size</td>
<td>10</td>
</tr>
<tr>
<td>Linear feasibility tolerance</td>
<td>$\sqrt{\epsilon}$</td>
</tr>
<tr>
<td>Linesearch tolerance</td>
<td>0.9</td>
</tr>
<tr>
<td>List/Nolist</td>
<td>List</td>
</tr>
<tr>
<td>Major iteration limit</td>
<td>$\max(50,3(n+n^2)+10n)$</td>
</tr>
<tr>
<td>Major print level</td>
<td>10</td>
</tr>
<tr>
<td>Minor iteration limit</td>
<td>$\max(50,3(n+n^2)+10n)$</td>
</tr>
<tr>
<td>Minor print level</td>
<td>0</td>
</tr>
<tr>
<td>Nonlinear feasibility</td>
<td>$\sqrt{\epsilon}$ if Derivative Level $\geq 2$</td>
</tr>
</tbody>
</table>
tolerance  
0.33 
otherwise (epsilon) 

Optimality tolerance  
0.8 
(event)

Step limit  
R 

Start objective check  
1 

Start constraint check  
1 

Stop objective check  
n 

Stop constraint check  
n 

Verify level  
0 

5.2. Description of Printed Output

The level of printed output from E04UCF is controlled by the user (see the description of Major Print Level and Minor Print Level in Section 5.1). If Minor Print Level > 0, output is obtained from the subroutines that solve the QP subproblem. For a detailed description of this information the reader should refer to E04NCF(*).

When Major Print Level >= 5, the following line of output is produced at every major iteration of E04UCF. In all cases, the values of the quantities printed are those in effect on completion of the given iteration.

Itn is the iteration count.

ItQP is the sum of the iterations required by the feasibility and optimality phases of the QP subproblem. Generally, ItQP will be 1 in the later iterations, since theoretical analysis predicts that the correct active set will be identified near the solution (see Section 3).

Note that ItQP may be greater than the Minor Iteration Limit if some iterations are required for the feasibility phase.

Step is the step (alpha) taken along the computed search direction. On reasonably well-behaved problems, the unit step will be taken as the solution is approached.
| **Nfun** | is the cumulative number of evaluations of the objective function needed for the linesearch. Evaluations needed for the estimation of the gradients by finite differences are not included. Nfun is printed as a guide to the amount of work required for the linesearch. |
| **Merit** | is the value of the augmented Lagrangian merit function (12) at the current iterate. This function will decrease at each iteration unless it was necessary to increase the penalty parameters (see Section 8.2). As the solution is approached, Merit will converge to the value of the objective function at the solution. If the QP subproblem does not have a feasible point (signified by I at the end of the current output line), the merit function is a large multiple of the constraint violations, weighted by the penalty parameters. During a sequence of major iterations with infeasible subproblems, the sequence of Merit values will decrease monotonically until either a feasible subproblem is obtained or E04UCF terminates with IFAIL = 3 (no feasible point could be found for the nonlinear constraints). |
| **Bnd** | is the number of simple bound constraints in the predicted active set. |
| **Lin** | is the number of general linear constraints in the predicted active set. |
| **Nln** | is the number of nonlinear constraints in the predicted active set (not printed if NCNLN is zero). |
| **Nz** | is the number of columns of Z (see Section 8.1). The value of Nz is the number of variables minus the number of constraints in the predicted active set; i.e., Nz = n-(Bnd + Lin + Nln). |
Norm $G_f$ is the Euclidean norm of $g$, the gradient of the objective function with respect to the free variables, i.e., variables not currently held at a bound.

$T$

Norm $G_z$ is $||Zg||$, the Euclidean norm of the projected gradient (see Section 8.1). Norm $G_z$ will be approximately zero in the neighbourhood of a solution.

Cond $H$ is a lower bound on the condition number of the Hessian approximation $H$.

Cond $H_z$ is a lower bound on the condition number of the projected Hessian approximation $H$ ($H = ZH Z^T$; see (6) and (12) in Sections 3 and 8.1). The larger this number, the more difficult the problem.

Cond $T$ is a lower bound on the condition number of the matrix of predicted active constraints.

Norm $C$ is the Euclidean norm of the residuals of constraints that are violated or in the predicted active set (not printed if NCNLN is zero). Norm $C$ will be approximately zero in the neighbourhood of a solution.

Penalty is the Euclidean norm of the vector of penalty parameters used in the augmented Lagrangian merit function (not printed if NCNLN is zero).

Conv is a three-letter indication of the status of the three convergence tests (8a)-(8c) defined in the description of the optional parameter Optimality Tolerance in Section 5.1 Each letter is T if the test is satisfied, and F otherwise. The three tests indicate whether:

(a) the sequence of iterates has converged;

(b) the projected gradient (Norm $G_z$) is sufficiently small; and

(c) the norm of the residuals of constraints in
the predicted active set (Norm C) is small enough.
If any of these indicators is F when E04UCF terminates with IFAIL = 0, the user should check the solution carefully.

M is printed if the Quasi-Newton update was modified to ensure that the Hessian approximation is positive-definite (see Section 8.3).

I is printed if the QP subproblem has no feasible point.

C is printed if central differences were used to compute the unspecified objective and constraint gradients. If the value of Step is zero, the switch to central differences was made because no lower point could be found in the linesearch. (In this case, the QP subproblem is resolved with the central-difference gradient and Jacobian.) If the value of Step is non-zero, central differences were computed because Norm Gz and Norm C imply that x is close to a Kuhn-Tucker point.

L is printed if the linesearch has produced a relative change in x greater than the value defined by the optional parameter Step Limit. If this output occurs frequently during later iterations of the run, Step Limit should be set to a larger value.

R is printed if the approximate Hessian has been refactorized. If the diagonal condition estimator of R indicates that the approximate Hessian is badly conditioned, the approximate Hessian is refactorized using column interchanges. If necessary, R is modified so that its diagonal condition estimator is bounded.

When Major Print Level = 1 or Major Print Level >= 10, the summary printout at the end of execution of E04UCF includes a listing of the status of every variable and constraint. Note that default names are assigned to all variables and constraints.

The following describes the printout for each variable.

Varbl gives the name (V) and index j=1,2,...,n of the variable.

State gives the state of the variable in the predicted
active set (FR if neither bound is in the active set, EQ if a fixed variable, LL if on its lower bound, UL if on its upper bound). If the variable is predicted to lie outside its upper or lower bound by more than the feasibility tolerance, State will be ++ or -- respectively. (The latter situation can occur only when there is no feasible point for the bounds and linear constraints.)

Value is the value of the variable at the final iteration.

Lower bound is the lower bound specified for the variable. (None indicates that BL(j)<=-BIGBND.)

Upper bound is the upper bound specified for the variable. (None indicates that BL(j)>=BIGBND.)

Lagr Mult is the value of the Lagrange-multiplier for the associated bound constraint. This will be zero if State is FR. If x is optimal, the multiplier should be non-negative if State is LL, and non-positive if State is UL.

Residual is the difference between the variable Value and the nearer of its bounds BL(j) and BU(j).

The printout for general constraints is the same as for variables, except for the following:

L Con is the name (L) and index i, for i = 1,2,...,NCLIN of a linear constraint.

N Con is the name (N) and index i, for i = 1,2,...,NCNLN of a nonlinear constraint.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

The input data for E04UCF should always be checked (even if E04UCF terminates with IFAIL=0).

Note that when Print Level>0, a short description of IFAIL is printed.

Errors and diagnostics indicated by IFAIL, together with some
recommendations for recovery are indicated below.

IFAIL = 1
The final iterate $x$ satisfies the first-order Kuhn-Tucker conditions to the accuracy requested, but the sequence of iterates has not yet converged. E04UCF was terminated because no further improvement could be made in the merit function.

This value of IFAIL may occur in several circumstances. The most common situation is that the user asks for a solution with accuracy that is not attainable with the given precision of the problem (as specified by Function Precision see Section 5). This condition will also occur if, by chance, an iterate is an 'exact' Kuhn-Tucker point, but the change in the variables was significant at the previous iteration. (This situation often happens when minimizing very simple functions, such as quadratics.)

If the four conditions listed in Section 5 for IFAIL = 0 are satisfied, $x$ is likely to be a solution of (1) even if IFAIL = 1.

IFAIL = 2
E04UCF has terminated without finding a feasible point for the linear constraints and bounds, which means that no feasible point exists for the given value of Linear Feasibility Tolerance (see Section 5.1). The user should check that there are no constraint redundancies. If the data for the constraints are accurate only to an absolute precision ($\sigma$), the user should ensure that the value of the optional parameter Linear Feasibility Tolerance is greater than ($\sigma$). For example, if all elements of $A$ are of order unity and are accurate to only three decimal places, Linear Feasibility Tolerance should be at least $10^{-3}$.

IFAIL = 3
No feasible point could be found for the nonlinear constraints. The problem may have no feasible solution. This means that there has been a sequence of QP subproblems for which no feasible point could be found (indicated by I at the end of each terse line of output). This behaviour will occur if there is no feasible point for the nonlinear constraints. (However, there is no general test that can determine whether a feasible point exists for a set of nonlinear constraints.) If the infeasible subproblems occur from the very first major iteration, it is highly likely that no feasible point exists. If infeasibilities occur when earlier subproblems have been feasible, small constraint
inconsistencies may be present. The user should check the validity of constraints with negative values of ISTATE. If the user is convinced that a feasible point does exist, E04UCF should be restarted at a different starting point.

IFAIL = 4

The limiting number of iterations (determined by the optional parameter Major Iteration Limit see Section 5.1) has been reached.

If the algorithm appears to be making progress, Major Iteration Limit may be too small. If so, increase its value and rerun E04UCF (possibly using the Warm Start option). If the algorithm seems to be 'bogged down', the user should check for incorrect gradients or ill-conditioning as described below under IFAIL = 6.

Note that ill-conditioning in the working set is sometimes resolved automatically by the algorithm, in which case performing additional iterations may be helpful. However, ill-conditioning in the Hessian approximation tends to persist once it has begun, so that allowing additional iterations without altering R is usually inadvisable. If the quasi-Newton update of the Hessian approximation was modified during the latter iterations (i.e., an M occurs at the end of each terse line), it may be worthwhile to try a warm start at the final point as suggested above.

IFAIL = 6

x does not satisfy the first-order Kuhn-Tucker conditions, and no improved point for the merit function could be found during the final line search.

A sufficient decrease in the merit function could not be attained during the final line search. This sometimes occurs because an overly stringent accuracy has been requested, i.e., Optimality Tolerance is too small. In this case the user should apply the four tests described under IFAIL = 0 above to determine whether or not the final solution is acceptable (see Gill et al [10], for a discussion of the attainable accuracy).

If many iterations have occurred in which essentially no progress has been made and E04UCF has failed completely to move from the initial point then subroutines OBJFUN or CONFUN may be incorrect. The user should refer to comments below under IFAIL = 7 and check the gradients using the Verify parameter. Unfortunately, there may be small errors in the objective and constraint gradients that cannot be detected by the verification process. Finite-difference
approximations to first derivatives are catastrophically affected by even small inaccuracies. An indication of this situation is a dramatic alteration in the iterates if the finite-difference interval is altered. One might also suspect this type of error if a switch is made to central differences even when Norm Gz and Norm C are large.

Another possibility is that the search direction has become inaccurate because of ill-conditioning in the Hessian approximation or the matrix of constraints in the working set; either form of ill-conditioning tends to be reflected in large values of ItQP (the number of iterations required to solve each QP subproblem).

If the condition estimate of the projected Hessian (Cond Hz) is extremely large, it may be worthwhile to rerun E04UCF from the final point with the Warm Start option. In this situation, ISTATE should be left unaltered and R should be reset to the identity matrix.

If the matrix of constraints in the working set is ill-conditioned (i.e., Cond T is extremely large), it may be helpful to run E04UCF with a relaxed value of the Feasibility Tolerance (Constraint dependencies are often indicated by wide variations in size in the diagonal elements of the matrix T, whose diagonals will be printed for Major Print Level >= 30).

IFAIL = 7
The user-provided derivatives of the objective function and/or nonlinear constraints appear to be incorrect.

Large errors were found in the derivatives of the objective function and/or nonlinear constraints. This value of IFAIL will occur if the verification process indicated that at least one gradient or Jacobian component had no correct figures. The user should refer to the printed output to determine which elements are suspected to be in error.

As a first-step, the user should check that the code for the objective and constraint values is correct - for example, by computing the function at a point where the correct value is known. However, care should be taken that the chosen point fully tests the evaluation of the function. It is remarkable how often the values x=0 or x=1 are used to test function evaluation procedures, and how often the special properties of these numbers make the test meaningless.

Special care should be used in this test if computation of the objective function involves subsidiary data communicated
in COMMON storage. Although the first evaluation of the function may be correct, subsequent calculations may be in error because some of the subsidiary data has accidently been overwritten.

Errors in programming the function may be quite subtle in that the function value is 'almost' correct. For example, the function may not be accurate to full precision because of the inaccurate calculation of a subsidiary quantity, or the limited accuracy of data upon which the function depends. A common error on machines where numerical calculations are usually performed in double precision is to include even one single-precision constant in the calculation of the function; since some compilers do not convert such constants to double precision, half the correct figures may be lost by such a seemingly trivial error.

IFAIL = 9

An input parameter is invalid. The user should refer to the printed output to determine which parameter must be redefined.

IFAILOverflow

If the printed output before the overflow error contains a warning about serious ill-conditioning in the working set when adding the jth constraint, it may be possible to avoid the difficulty by increasing the magnitude of the optional parameter Linear Feasibility Tolerance or Nonlinear Feasibility Tolerance, and rerunning the program. If the message recurs even after this change, the offending linearly dependent constraint (with index 'j') must be removed from the problem. If overflow occurs in one of the user-supplied routines (e.g. if the nonlinear functions involve exponentials or singularities), it may help to specify tighter bounds for some of the variables (i.e., reduce the gap between appropriate l and u).

7. Accuracy

If IFAIL = 0 on exit then the vector returned in the array X is an estimate of the solution to an accuracy of approximately Feasibility Tolerance (see Section 5.1), whose default value is 0.8 (epsilon), where (epsilon) is the machine precision (see X02AJF(*)).

8. Further Comments

In this section we give some further details of the method used
by E04UCF.

8.1. Solution of the Quadratic Programming Subproblem

The search direction $p$ is obtained by solving (4) using the method of E04NCF(*) (Gill et al [8]), which was specifically designed to be used within an SQP algorithm for nonlinear programming.

The method of E04UCF is a two-phase (primal) quadratic programming method. The two phases of the method are: finding an initial feasible point by minimizing the sum of infeasibilities (the feasibility phase), and minimizing the quadratic objective function within the feasible region (the optimality phase). The computations in both phases are performed by the same subroutines. The two-phase nature of the algorithm is reflected by changing the function being minimized from the sum of infeasibilities to the quadratic objective function.

In general, a quadratic program must be solved by iteration. Let $p$ denote the current estimate of the solution of (4); the new iterate $p$ is defined by

$$p = p + \alpha d,$$

where, as in (3), $\alpha$ is a non-negative step length and $d$ is a search direction.

At the beginning of each iteration of E04UCF, a working set is defined of constraints (general and bound) that are satisfied exactly. The vector $d$ is then constructed so that the values of constraints in the working set remain unaltered for any move along $d$. For a bound constraint in the working set, this property is achieved by setting the corresponding component of $d$ to zero, i.e., by fixing the variable at its bound. As before, the subscripts 'FX' and 'FR' denote selection of the components associated with the fixed and free variables.

Let $C$ denote the sub-matrix of rows of

$$
\begin{pmatrix}
(A) \\
(L) \\
(A) \\
(N)
\end{pmatrix}
$$

corresponding to general constraints in the working set. The general constraints in the working set will remain unaltered if
\[ C \, d = 0, \quad (10) \]
\[ \text{which is equivalent to defining } \, d \, \text{ as} \]
\[ d = Z \, \text{d} \quad (11) \]
\[ \text{for some vector } \, d \, \text{, where } \, Z \, \text{is the matrix associated with the TQ} \]
\[ \text{factorization (5) of } \, C. \]

The definition of \( d \) in (11) depends on whether the current \( p \) is \( z \)
feasible. If not, \( d \) is zero except for a component \( (\gamma) \) in \( z \)
the \( j \)th position, where \( j \) and \( (\gamma) \) are chosen so that the sum
of infeasibilities is decreasing along \( d \). (For further details,
see Gill et al [8].) In the feasible case, \( d \) satisfies the \( z \)
equations
\[ T \quad T \quad R \, R \, d = -Z \, q, \quad (12) \]
\[ z \quad z \quad z \quad \text{FR} \]
\[ \text{where } \, R \, \text{ is the Cholesky factor of } \, Z \, H \, Z \, \text{and } \, q \, \text{is the gradient} \]
\[ z \quad \text{FR} \]
\[ \text{of the quadratic objective function (} q = g + H p). \quad (\text{The vector } Z \, q \]
\[ \text{FR} \]
is the projected gradient of the QP.) With (12), \( P + d \) is the
minimizer of the quadratic objective function subject to treating
the constraints in the working set as equalities.

If the QP projected gradient is zero, the current point is a
constrained stationary point in the subspace defined by the
working set. During the feasibility phase, the projected gradient
will usually be zero only at a vertex (although it may vanish at
non-vertices in the presence of constraint dependencies). During
the optimality phase, a zero projected gradient implies that \( p \)
minimizes the quadratic objective function when the constraints
in the working set are treated as equalities. In either case,
Lagrange multipliers are computed. Given a positive constant
\( (\delta) \) of the order of the machine precision, the Lagrange
multiplier \( (\mu) \) corresponding to an inequality constraint in the
\( j \)
working set at its upper bound is said to be optimal if 
\( (\mu_j) \leq (\delta_j) \) when the jth constraint is at its upper bound, or 
\( \mu_j \geq -\delta_j \) when the associated constraint is at its lower
bound. If any multiplier is non-optimal, the current objective
function (either the true objective or the sum of 
infeasibilities) can be reduced by deleting the corresponding
constraint from the working set.

If optimal multipliers occur during the feasibility phase and the
sum of infeasibilities is non-zero, no feasible point exists. The
QP algorithm will then continue iterating to determine the
minimum sum of infeasibilities. At this point, the Lagrange
multiplier \( (\mu_j) \) will satisfy 
\[ -(1+\delta_j) \leq (\mu_j) \leq (\delta_j) \] for an
inequality constraint at its upper bound, and 
\[ -(\delta_j) \leq (\mu_j) \leq 1+\delta_j \] for an inequality at its lower bound.

The Lagrange multiplier for an equality constraint will satisfy 
\( |(\mu_j)| \leq 1+\delta_j \).

The choice of step length \( (\sigma) \) in the QP iteration (9) is
based on remaining feasible with respect to the satisfied
constraints. During the optimality phase, if \( p+d \) is feasible,
\( (\sigma) \) will be taken as unity. (In this case, the projected
gradient at \( p \) will be zero.) Otherwise, \( (\sigma) \) is set to
\( (\sigma) \), the step to the 'nearest' constraint, which is added to
the working set at the next iteration.

Each change in the working set leads to a simple change to \( C : \)
if the status of a general constraint changes, a row of \( C \) is
altered; if a bound constraint enters or leaves the working set,
a column of \( C \) changes. Explicit representations are recurred of
the matrices \( T, Q \) and \( R \), and of the vectors \( Q q \) and \( Q g \).

8.2. The Merit Function

After computing the search direction as described in Section 3,
each major iteration proceeds by determining a step length
\( (\alpha) \) in (3) that produces a 'sufficient decrease' in the
augmented Lagrangian merit function.
where \( x, (\lambda), \text{and } s \) vary during the line search. The summation terms in (13) involve only the nonlinear constraints. The vector \((\lambda)\) is an estimate of the Lagrange multipliers for the nonlinear constraints of (1). The non-negative slack variables \( \{s_i\} \) allow nonlinear inequality constraints to be treated without introducing discontinuities. The solution of the QP subproblem (4) provides a vector triple that serves as a direction of search for the three sets of variables. The non-negative vector \((\rho)\) of penalty parameters is initialised to zero at the beginning of the first major iteration. Thereafter, selected components are increased whenever necessary to ensure descent for the merit function. Thus, the sequence of norms of \((\rho)\) (the printed quantity Penalty, see Section 5.2) is generally non-decreasing, although each \((\rho)\) may be reduced a limited number of times.

The merit function (13) and its global convergence properties are described in Gill et al [9].

8.3. The Quasi-Newton Update

The matrix \( H \) in (4) is a positive-definite quasi-Newton approximation to the Hessian of the Lagrangian function. (For a review of quasi-Newton methods, see Dennis and Schnabel [3].) At the end of each major iteration, a new Hessian approximation \( \tilde{H} \) is defined as a rank-two modification of \( H \). In E04UCF, the BFGS quasi-Newton update is used:

\[
\tilde{H} = H - \frac{1}{\rho} \frac{\mathbf{s} \mathbf{s}^T}{\lambda} + \frac{\mathbf{y} \mathbf{y}^T}{\lambda},
\]

where \( \mathbf{s} = x - x \) (the change in \( x \)).
In E04UCF, H is required to be positive-definite. If H is positive-definite, H defined by (14) will be positive-definite if and only if y is positive (see, e.g. Dennis and More [1]). Ideally, y in (14) would be taken as y, the change in gradient of the Lagrangian function

\[ y = \nabla L(N) - \mu^T g(N) \]

where \((\mu)\) denotes the QP multipliers associated with the nonlinear constraints of the original problem. If y is not sufficiently positive, an attempt is made to perform the update with a vector y of the form

\[ y = y + \sum_{i=1}^m (\omega) (a(x)_i c(x)_i - a(x)c(x)_i) \]

where \((\omega) \geq 0\). If no such vector can be found, the update is performed with a scaled y; in this case, M is printed to indicate that the update is modified.

Rather than modifying H itself, the Cholesky factor of the transformed Hessian H (6) is updated, where Q is the matrix from (5) associated with the active set of the QP subproblem. The update (13) is equivalent to the following update to H:

\[
H = H - \sum_{i=1}^m (a(x)_i c(x)_i - a(x)c(x)_i) y^T y + \sum_{i=1}^m (a(x)_i c(x)_i - a(x)c(x)_i) s^T s
\]

where y = Q y, and s = Q s. This update may be expressed as a rank-
one update to R (see Dennis and Schnabel [2]).

9. Example

This section describes one version of the so-called 'hexagon' problem (a different formulation is given as Problem 108 in Hock and Schittkowski [11]). The problem is to determine the hexagon of maximum area such that no two of its vertices are more than one unit apart (the solution is not a regular hexagon).

All constraint types are included (bounds, linear, nonlinear), and the Hessian of the Lagrangian function is not positive-definite at the solution. The problem has nine variables, non-infinite bounds on seven of the variables, four general linear constraints, and fourteen nonlinear constraints.

The objective function is

\[ F(x) = -x_2 x_6 + x_2 x_7 - x_3 x_3 + x_3 x_5 + x_4 x_4 + x_6 x_8. \]

The bounds on the variables are

\[ x \geq 0, \quad -1 \leq x_1 \leq 1, \quad x_3 \geq 0, \quad x \geq 0, \quad x_7 \geq 0, \quad x_8 \leq 0, \quad x_9 \leq 0. \]

Thus,

\[ l = (0, -\infty, -\infty, -\infty, 0, 0, 0, -\infty, -\infty) \]

\[ u = (\infty, \infty, 1, \infty, \infty, \infty, \infty, 0, 0) \]

The general linear constraints are

\[ x_2 - x_6 \geq 0, \quad x_2 - x_7 \geq 0, \quad x_3 - x_3 \geq 0, \quad \text{and} \quad x_4 - x_4 \geq 0. \]

Hence,

\[ A = (0, 0, 0, 0, 0, 0, 0, 0, 0) \quad \text{and} \quad u = (\infty). \]

The nonlinear constraints are all of the form \( c_i(x) \leq 1, \) for
i=1,2,...,14; hence, all components of \( l \) are \(-\infty\), and all components of \( u \) are 1. The fourteen functions \( \{c_i(x)\} \) are

\[
\begin{align*}
2 & 2 \\
c_i(x) &= x + x \\
1 & 1 6 \\
2 & 2 \\
c_i(x) &= (x - x) + (x - x) \\
2 & 1 7 6 \\
2 & 2 \\
c_i(x) &= (x - x) + x \\
3 & 1 6 \\
2 & 2 \\
c_i(x) &= (x - x) + (x - x) \\
4 & 1 4 6 8 \\
2 & 2 \\
c_i(x) &= (x - x) + (x - x) \\
5 & 1 5 6 9 \\
2 & 2 \\
c_i(x) &= x + x \\
6 & 2 7 \\
2 & 2 \\
c_i(x) &= (x - x) + x \\
7 & 3 2 7 \\
2 & 2 \\
c_i(x) &= (x - x) + (x - x) \\
8 & 4 2 8 7 \\
2 & 2 \\
c_i(x) &= (x - x) + (x - x) \\
9 & 2 5 7 9 \\
2 & 2 \\
c_i(x) &= (x - x) + x \\
10 & 4 3 8 \\
2 & 2 \\
c_i(x) &= (x - x) + x \\
11 & 5 3 9 \\
2 & 2
\end{align*}
\]
c (x) = x + x ,
12  4  8

2  2
2


2  2


2

An optimal solution (to five figures) is

* x = (0.060947, 0.59765, 1.0, 0.59765, 0.060947, 0.34377, 0.5, T
-0.5, 0.34377) ,

* and F(x ) = -1.34996. (The optimal objective function is unique,
but is achieved for other values of x.) Five nonlinear

* constraints and one simple bound are active at x . The sample
solution output is given later in this section, following the
sample main program and problem definition.

Two calls are made to E04UCF in order to demonstrate some of its
features. For the first call, the starting point is:

T
x = (0.1, 0.125, 0.666666, 0.142857, 0.111111, 0.2, 0.25, -0.2, -0.25) .
0

All objective and constraint derivatives are specified in the
user-provided subroutines OBJFN1 and CONFN1, i.e., the default
option Derivative Level = 3 is used.

On completion of the first call to E04UCF, the optimal variables
are perturbed to produce the initial point for a second run in
which the problem functions are defined by the subroutines OBJFN2
and CONFN2. To illustrate one of the finite-difference options in
E04UCF, these routines are programmed so that the first six
components of the objective gradient and the constant elements of
the Jacobian matrix are not specified; hence, the option
Derivative Level = 0 is chosen. During computation of the finite-
difference intervals, the constant Jacobian elements are
identified and set, and E04UCF automatically increases the
derivative level to 2.
The second call to E04UCF illustrates the use of the Warm Start Level option to utilize the final active set, nonlinear multipliers and approximate Hessian from the first run. Note that Hessian = Yes was specified for the first run so that the array R would contain the Cholesky factor of the approximate Hessian of the Lagrangian.

The two calls to E04UCF illustrate the alternative methods of assigning default parameters. (There is no special significance in the order of these assignments; an options file may just as easily be used to modify parameters set by E04UEF.)

The results are typical of those obtained from E04UCF when solving well behaved (non-trivial) nonlinear problems. The approximate Hessian and working set remain relatively well-conditioned. Similarly the penalty parameters remain small and approximately constant. The numerical results illustrate much of the theoretically predicted behaviour of a quasi-Newton SQP method. As \( x \) approaches the solution, only one minor iteration is performed per major iteration, and the Norm Gz and Norm C columns exhibit the fast linear convergence rate mentioned in Sections 5 and 6. Note that the constraint violations converge earlier than the projected gradient. The final values of the project gradient norm and constraint norm reflect the limiting accuracy of the two quantities. It is possible to achieve almost full precision in the constraint norm but only half precision in the projected gradient norm. Note that the final accuracy in the nonlinear constraints is considerably better than the feasibility tolerance, because the constraint violations are being refined during the last few iterations while the algorithm is working to reduce the projected gradient norm. In this problem, the constraint values and Lagrange multipliers at the solution are 'well balanced', i.e., all the multipliers are approximately the same order of magnitude. The behaviour is typical of a well-scaled problem.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
E04 -- Minimizing or Maximizing a Function
E04UDF
E04UDF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

To supply optional parameters to E04UCF from an external file.

2. Specification

```fortran
SUBROUTINE E04UDF (IOPTNS, INFORM)
    INTEGER IOPTNS, INFORM
```

3. Description

E04UDF may be used to supply values for optional parameters to E04UCF. E04UDF reads an external file and each line of the file defines a single optional parameter. It is only necessary to supply values for those parameters whose values are to be different from their default values.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

```
Print level = 1
```

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77’s I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

The file containing the options must start with begin and must finish with end. An example of a valid options file is:

```
Begin * Example options file
Print level =10
End
```

Normally each line of the file is printed as it is read, on the
current advisory message unit (see X04ABF), but printing may be suppressed using the keyword nolist. To suppress printing of begin, nolist must be the first option supplied as in the file:

```plaintext
Begin
  Nolist
  Print level = 10
End
```

Printing will automatically be turned on again after a call to E04UCF and may be turned on again at any time by the user by using the keyword list.

Optional parameter settings are preserved following a call to E04UCF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values prior to a subsequent call to E04UCF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the document for E04UCF.

4. References

None.

5. Parameters

1: IOPTNS -- INTEGER
   Input
   On entry: IOPTNS must be the unit number of the options file. Constraint: 0 <= IOPTNS <= 99.

2: INFORM -- INTEGER
   Output
   On exit: INFORM will be zero, if an options file with the current structure has been read. Otherwise INFORM will be positive. Positive values of INFORM indicate that an options file may not have been successfully read as follows:
   INFORM = 1
     IOPTNS is not in the range [0,99].
   INFORM = 2
     begin was found, but end-of-file was found before end was found.
   INFORM = 3
     end-of-file was found before begin was found.

6. Error Indicators and Warnings

If a line is not recognised as a valid option, then a warning
message is output on the current advisory message unit (X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04UEF may also be used to supply optional parameters to E04UCF.

9. Example

See the example for E04UCF.

E04UEF may be used to supply values for optional parameters to E04UCF. It is only necessary to call E04UEF for those parameters whose values are to be different from their default values. One call to E04UEF sets one parameter value.

Each optional parameter is defined by a single character string of up to 72 characters, consisting of one or more items. The items associated with a given option must be separated by spaces, or equal signs (=). Alphabetic characters may be upper or lower case. The string

Print level = 1

is an example of a string used to set an optional parameter. For each option the string contains one or more of the following
items:

(a) A mandatory keyword.

(b) A phrase that qualifies the keyword.

(c) A number that specifies an INTEGER or real value. Such numbers may be up to 16 contiguous characters in Fortran 77's I, F, E or D formats, terminated by a space if this is not the last item on the line.

Blank strings and comments are ignored. A comment begins with an asterisk (*) and all subsequent characters in the string are regarded as part of the comment.

Normally, each user-specified option is printed as it is defined, on the current advisory message unit (see X04ABF), but this printing may be suppressed using the keyword nolist. Thus the statement

```
CALL E04UEF ('Nolist')
```

suppresses printing of this and subsequent options. Printing will automatically be turned on again after a call to E04UCF, and may be turned on again at any time by the user, by using the keyword list.

Optional parameter settings are preserved following a call to E04UCF, and so the keyword defaults is provided to allow the user to reset all the optional parameters to their default values by the statement,

```
CALL E04UEF ('Defaults')
```

prior to a subsequent call to E04UCF.

A complete list of optional parameters, their abbreviations, synonyms and default values is given in Section 5.1 of the document for E04UCF.

4. References

None.

5. Parameters

1: STRING -- CHARACTER(*) Input
   On entry: STRING must be a single valid option string. See Section 3 above and Section 5.1 of the routine document for E04UCF. On entry: STRING must be a single valid option
string. See Section 3 above and Section 5.1 of the routine document for E04UCF.

6. Error Indicators and Warnings

If the parameter STRING is not recognised as a valid option string, then a warning message is output on the current advisory message unit (X04ABF).

7. Accuracy

Not applicable.

8. Further Comments

E04UDF may also be used to supply optional parameters to E04UCF.

9. Example

See the example for E04UCF.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

E04 -- Minimizing or Maximizing a Function               E04YCF
E04YCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

E04YCF returns estimates of elements of the variance-covariance matrix of the estimated regression coefficients for a nonlinear least squares problem. The estimates are derived from the Jacobian of the function f(x) at the solution.

This routine may be used following any one of the nonlinear least-squares routines E04FCF(*), E04FDF, E04GBF(*), E04GCF, E04GDF(*), E04GEF(*), E04HEF(*), E04HFF(*).

2. Specification

SUBROUTINE E04YCF (JOB, M, N, FSUMSQ, S, V, LV, CJ, WORK, 1
                   IFAIL)
       INTEGER JOB, M, N, LV, IFAIL
       DOUBLE PRECISION FSUMSQ, S(N), V(LV,N), CJ(N), WORK(N)

3. Description
E04YCF is intended for use when the nonlinear least-squares
\( \mathbf{F}(\mathbf{x}) = \mathbf{f}(\mathbf{x}) \mathbf{f}(\mathbf{x}) \) function, represents the goodness of fit of a
nonlinear model to observed data. The routine assumes that the
Hessian of \( \mathbf{F}(\mathbf{x}) \), at the solution, can be adequately approximated
\( \mathbf{J}^T \mathbf{J} \) by \( 2 \mathbf{J}^T \mathbf{J} \), where \( \mathbf{J} \) is the Jacobian of \( \mathbf{f}(\mathbf{x}) \) at the solution. The
estimated variance-covariance matrix \( \mathbf{C} \) is then given by
\[
\mathbf{C} = (\sigma)^2 \left( \mathbf{J}^T \mathbf{J} \right)^{-1} \left( \mathbf{J}^T \mathbf{J} \right)_{\text{non-singular}},
\]
where \( (\sigma)^2 \) is the estimated variance of the residual at the
solution, \( \mathbf{x} \), given by
\[
(\sigma)^2 = \frac{2}{m-n} \mathbf{F}(\mathbf{x})
\]
\( m \) being the number of observations and \( n \) the number of variables.

The diagonal elements of \( \mathbf{C} \) are estimates of the variances of the
estimated regression coefficients. See the Chapter Introduction
E04 and Bard [1] and Wolberg [2] for further information on the
use of \( \mathbf{C} \).

When \( \mathbf{J}^T \mathbf{J} \) is singular then \( \mathbf{C} \) is taken to be
\[
\mathbf{C} = (\sigma)^2 \left( \mathbf{J}^T \mathbf{J} \right)^{-T},
\]
\( \left( \mathbf{J}^T \mathbf{J} \right)^{-T} \) where \( \mathbf{J}^T \mathbf{J} \) is the pseudo-inverse of \( \mathbf{J} \), but in this case the
parameter IFAIL is returned as non-zero as a warning to the user
that \( \mathbf{J} \) has linear dependencies in its columns. The assumed rank
of \( \mathbf{J} \) can be obtained from IFAIL.

The routine can be used to find either the diagonal elements of
\( \mathbf{C} \), or the elements of the \( j \)th column of \( \mathbf{C} \), or the whole of \( \mathbf{C} \).

E04YCF must be preceded by one of the nonlinear least-squares
routines mentioned in Section 1, and requires the parameters
FSUMSQ, S and V to be supplied by those routines. FSUMSQ is the
residual sum of squares $F(x)$, and $S$ and $V$ contain the singular values and right singular vectors respectively in the singular value decomposition of $J$. $S$ and $V$ are returned directly by the comprehensive routines E04FCF(*), E04GBF(*), E04GDF(*) and E04HEF(*), but are returned as part of the workspace parameter $W$ from the easy-to-use routines E04FDF, E04GCF, E04GEF(*) and E04HFF(*). In the case of E04FDF, $S$ starts at $W(NS)$, where

$$NS = 6 \times N + 2 \times M + M \times N + 1 + \max(1, N \times (N-1)/2)$$

and in the cases of the remaining easy-to-use routines, $S$ starts at $W(NS)$, where

$$NS = 7 \times N + 2 \times M + M \times N + N \times (N+1)/2 + 1 + \max(1, N \times (N-1)/2)$$

The parameter $V$ starts immediately following the elements of $S$, so that $V$ starts at $W(NV)$, where

$$NV = NS + N.$$

For all the easy-to-use routines the parameter $LV$ must be supplied as $N$. Thus a call to E04YCF following E04FDF can be illustrated as

```fortran
CALL E04FDF (M, N, X, FSUMSQ, IW, LIW, W, LW, IFAIL)
NS = 6 \times N + 2 \times M + M \times N + 1 + \max(1, N \times (N-1)/2)
NV = NS + N
CALL E04YCF (JOB, M, N, FSUMSQ, W(NS), W(NV), * N, CJ, WORK, IFAIL)
```

where the parameters $M$, $N$, $FSUMSQ$ and the $(n+n)$ elements $W(NS)$, $W(NS+1), \ldots$, $W(NV+N+N-1)$ must not be altered between the calls to E04FDF and E04YCF. The above illustration also holds for a call to E04YCF following a call to one of E04GCF, E04GEF(*), E04HFF(*) except that $NS$ must be computed as

$$NS = 7 \times N + 2 \times M + M \times N + (N \times (N+1)/2 + 1 + \max((1, N \times (N-1))/2)$$

4. References


5. Parameters
1: JOB -- INTEGER
   On entry: which elements of C are returned as follows:
   JOB = -1
       The n by n symmetric matrix C is returned.
   JOB = 0
       The diagonal elements of C are returned.
   JOB > 0
       The elements of column JOB of C are returned.
   Constraint: -1 <= JOB <= N.

2: M -- INTEGER
   On entry: the number m of observations (residuals f(x)).
   Constraint: M >= N.

3: N -- INTEGER
   On entry: the number n of variables (x). Constraint: 1 <=
   N <= M.

4: FSUMSQ -- DOUBLE PRECISION
   On entry: the sum of squares of the residuals, F(x), at the
   solution x, as returned by the nonlinear least-squares
   routine. Constraint: FSUMSQ >= 0.0.

5: S(N) -- DOUBLE PRECISION array
   On entry: the n singular values of the Jacobian as returned
   by the nonlinear least-squares routine. See Section 3 for
   information on supplying S following one of the easy-to-use
   routines.

6: V(LV,N) -- DOUBLE PRECISION array
   On entry: the n by n right-hand orthogonal matrix (the
   right singular vectors) of J as returned by the nonlinear
   least-squares routine. See Section 3 for information on
   supplying V following one of the easy-to-use routines. On
   exit: when JOB >= 0 then V is unchanged.

   When JOB = -1 then the leading n by n part of V is
   overwritten by the n by n matrix C. When E04YCF is called
   with JOB = -1 following an easy-to-use routine this means
   that C is returned, column by column, in the n elements of
2

W given by W(NV), W(NV+1), ..., W(NV+N-1). (See Section 3 for the definition of NV).

7: LV -- INTEGER
    Input
    On entry:
    the first dimension of the array V as declared in the
    (sub)program from which E04YCF is called.
    When V is passed in the workspace parameter W following one
    of the easy-to-use least-square routines, LV must be the
    value N.

8: CJ(N) -- DOUBLE PRECISION array
    Output
    On exit: with JOB = 0, CJ returns the n diagonal elements
    of C.
    With JOB = j>0, CJ returns the n elements of the jth column
    of C.
    When JOB = -1, CJ is not referenced.

9: WORK(N) -- DOUBLE PRECISION array
    Workspace
    When JOB = -1 or 0 then WORK is used as internal workspace.
    When JOB > 0, WORK is not referenced.

10: IFAIL -- INTEGER
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. Users who are
    unfamiliar with this parameter should refer to the Essential
    Introduction for details.
    On exit: IFAIL = 0 unless the routine detects an error or
    gives a warning (see Section 6).
    For this routine, because the values of output parameters
    may be useful even if IFAIL /=0 on exit, users are
    recommended to set IFAIL to -1 before entry. It is then
    essential to test the value of IFAIL on exit. To suppress
    the output of an error message when soft failure occurs, set
    IFAIL to 1.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

IFAIL= 1
    On entry JOB < -1,
    or JOB > N,
or \( N < 1 \),
or \( M < N \),
or \( \text{FSUMSQ} < 0.0 \).

**IFAIL = 2**
The singular values are all zero, so that at the solution the Jacobian matrix \( J \) has rank 0.

**IFAIL > 2**
At the solution the Jacobian matrix contains linear, or near linear, dependencies amongst its columns. In this case the required elements of \( C \) have still been computed based upon \( J \) having an assumed rank given by \( (\text{IFAIL}-2) \). The rank is computed by regarding singular values \( SV(j) \) that are not larger than \( 10*(\text{epsilon})*SV(1) \) as zero, where \( (\text{epsilon}) \) is the machine precision (see X02AJF(*)). Users who expect near linear dependencies at the solution and are happy with this tolerance in determining rank should call E04YCF with IFAIL = 1 in order to prevent termination by P01ABF(*). It is then essential to test the value of IFAIL on exit from E04YCF.

**IFAILOverflow**
If overflow occurs then either an element of \( C \) is very large, or the singular values or singular vectors have been incorrectly supplied.

7. **Accuracy**

The computed elements of \( C \) will be the exact covariances corresponding to a closely neighbouring Jacobian matrix \( J \).

8. **Further Comments**

When \( JOB = -1 \) the time taken by the routine is approximately \( 3 \) proportional to \( n \). When \( JOB \geq 0 \) the time taken by the routine is approximately proportional to \( n \).

9. **Example**

To estimate the variance-covariance matrix \( C \) for the least-squares estimates of \( x_1, x_2, \) and \( x_3 \) in the model

\[
\begin{align*}
t & \quad 1 \\
1 & \quad y = x_1 + ---------
\end{align*}
\]
using the 15 sets of data given in the following table:

\[
\begin{array}{cccc}
  y & t & t & t \\
  1 & 2 & 3 \\
  0.14 & 1.0 & 15.0 & 1.0 \\
  0.18 & 2.0 & 14.0 & 2.0 \\
  0.22 & 3.0 & 13.0 & 3.0 \\
  0.25 & 4.0 & 12.0 & 4.0 \\
  0.29 & 5.0 & 11.0 & 5.0 \\
  0.32 & 6.0 & 10.0 & 6.0 \\
  0.35 & 7.0 & 9.0 & 7.0 \\
  0.39 & 8.0 & 8.0 & 8.0 \\
  0.37 & 9.0 & 7.0 & 7.0 \\
  0.58 & 10.0 & 6.0 & 6.0 \\
  0.73 & 11.0 & 5.0 & 5.0 \\
  0.96 & 12.0 & 4.0 & 4.0 \\
  1.34 & 13.0 & 3.0 & 3.0 \\
  2.10 & 14.0 & 2.0 & 2.0 \\
  4.39 & 15.0 & 1.0 & 1.0 \\
\end{array}
\]

The program uses \((0.5, 1.0, 1.5)\) as the initial guess at the position of the minimum and computes the least-squares solution using E04FDF. See the routine document E04FDF for further information.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagOptimisationPackage (NAGE04)

Exports:
e04dgf  e04fdf  e04gcf  e04jaf  e04mbf
  e04naf  e04ucf  e04ycf

| package NAGE04 NagOptimisationPackage |

)abbrev package NAGE04 NagOptimisationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:09 1994
++ Description:
++ This package uses the NAG Library to perform optimization.
++ An optimization problem involves minimizing a function (called
++ the objective function) of several variables, possibly subject to
++ restrictions on the values of the variables defined by a set of
++ constraint functions. The routines in the NAG Foundation Library
++ are concerned with function minimization only, since the problem
++ of maximizing a given function can be transformed into a
++ minimization problem simply by multiplying the function by -1.

NagOptimisationPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
e04dgf : (Integer,DoubleFloat,DoubleFloat,Integer,_,
  DoubleFloat,Boolean,DoubleFloat,DoubleFloat,Integer,Integer,Integer,_
  Integer,Matrix DoubleFloat,Integer,_,
  Union(fn:FileName,fp:Asp49(OBJFUN))) -> Result
++ e04dgf(n,es,fu,it,lin,list,ma,op,pr,sta,sto,ve,x,ifail,objfun)
++ minimizes an unconstrained nonlinear function of several
++ variables using a pre-conditioned, limited memory quasi-Newton
++ conjugate gradient method. First derivatives are required. The
++ routine is intended for use on large scale problems.
++ See \downlink{Manual Page}{manpageXXe04dgf}.
e04fdf : (Integer,Integer,Integer,Integer,Integer,_) 
   Matrix DoubleFloat,Integer,_, 
   Union(fn:FileName,fp:Asp50(LSFUN1)) -> Result 
++ e04fdf(m,n,liw,lw,x,ifail,lsfun1) 
++ is an easy-to-use algorithm for finding an unconstrained 
++ minimum of a sum of squares of m nonlinear functions in n 
++ variables (m>=n). No derivatives are required. 
++ See \downlink{Manual Page}{manpageXXe04fdf}.
e04gcf : (Integer,Integer,Integer,Integer,Integer,_) 
   Matrix DoubleFloat,Integer,_, 
   Union(fn:FileName,fp:Asp19(LSFUN2)) -> Result 
++ e04gcf(m,n,liw,lw,x,ifail,lsfun2) 
++ is an easy-to-use quasi-Newton algorithm for finding an 
++ unconstrained minimum of m nonlinear 
++ functions in n variables (m>=n). First derivatives are required. 
++ See \downlink{Manual Page}{manpageXXe04gcf}.
e04jaf : (Integer,Integer,Integer,Integer,Integer,Integer,_) 
   Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Integer,_, 
   Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat, 
   Boolean,Integer,Integer,_) 
   Union(fn:FileName,fp:Asp24(FUNCT1)) -> Result 
++ e04jaf(n,ibound,liw,lw,bl,bl,bl,x,ifail,funct1) 
++ is an easy-to-use quasi-Newton algorithm for finding a 
++ minimum of a function F(x ,x ,...,x ), subject to fixed upper and 
++ 1 2 n 
++ lower bounds of the independent variables x ,x ,...,x , using 
++ 1 2 n 
++ function values only. 
++ See \downlink{Manual Page}{manpageXXe04jaf}.
e04mbf : (Integer,Integer,Integer,Integer,Integer,Integer,Integer,_) 
   Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,_, 
   Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Boolean,Integer,Integer,_, 
   Matrix DoubleFloat,Matrix DoubleFloat,_, 
   Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat, 
   Matrix DoubleFloat,Boolean,Boolean,Integer,Integer,_, 
   Matrix DoubleFloat,Matrix Integer,Integer,_, 
   Union(fn:FileName,fp:Asp20(QPHESS)) -> Result 
++ e04mbf(itmax,msglvl,n,nclin,nctotl,nrowa,a,bl, 
++ bu,cvec,linobj,liwork,lwork,x,ifail) 
++ is an easy-to-use routine for solving linear programming 
++ problems, or for finding a feasible point for such problems. It 
++ is not intended for large sparse problems. 
++ See \downlink{Manual Page}{manpageXXe04mbf}.
e04naf : (Integer,Integer,Integer,Integer,Integer,Integer,Integer,_) 
   Integer,DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,_, 
   Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat, 
   Matrix DoubleFloat,Boolean,Boolean,Integer,Integer,_, 
   Matrix DoubleFloat,Matrix Integer,Integer,_, 
   Union(fn:FileName,fp:Asp20(QPHESS)) -> Result 
++ e04naf(itmax,msglvl,n,nclin,nctotl,nrowa, 
++ nrowb,ncolh,bigbnd,a,bl, 
++ bu,cvec,featol,hess,cold,ipp,orthog,liwork,lwork,x,istate,ifail,qpess) 
++ is a comprehensive 
++ programming (QP) or linear programming (LP) problems. It is not 
++ intended for large sparse problems. 
++ See \downlink{Manual Page}{manpageXXe04naf}.
e04ucf : (Integer,Integer,Integer,Integer,Integer,Integer,_)
++ e04ucf(n,nclin,ncnln,nrowa,nrowj,nrowr,a,bl,bu,liwork,lwork,sta,
++ cra,der,fea,fun,hes,infb,infh,linf,lint,majm,majp,mini,
++ minf,mon,nonf,opt,ste,tao,tau,ve,istate,clamda,
++ r,x,ifail,confun,OBJFUN)) -> Result
++ is designed to minimize an arbitrary smooth function
++ subject to constraints on the
++ variables, linear constraints.
++ (E04UCF may be used for unconstrained, bound-constrained and
++ linearly constrained optimization.) The user must provide
++ subroutines that define the objective and constraint functions
++ and as many of their first partial derivatives as possible.
++ Unspecified derivatives are approximated by finite differences.
++ All matrices are treated as dense, and hence E04UCF is not
++ intended for large sparse problems.
++ See \downlink{Manual Page}{manpageXXe04ucf}.
e04ycf : (Integer,Integer,Integer,DoubleFloat,
++ e04ycf(job,m,n,fsumsq,s,lv,v,ifail)
++ returns estimates of elements of the variance
++ matrix of the estimated regression coefficients for a nonlinear
++ least squares problem. The estimates are derived from the
++ Jacobian of the function f(x) at the solution.
++ See \downlink{Manual Page}{manpageXXe04ycf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName,fp:Asp49(OBJFUN))
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Boolean)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Matrix Integer)
e04dgf(nArg:Integer,esArg:DoubleFloat,fuArg:DoubleFloat,_
itArg:Integer,linArg:DoubleFloat,listArg:Boolean,_
maArg:DoubleFloat,opArg:DoubleFloat,prArg:Integer,_
staArg:Integer,stoArg:Integer,veArg:Integer,_
xArg:Matrix DoubleFloat,ifailArg:Integer,_
objfunArg:Union(fn:FileName,fp:Asp49(OBJFUN))): Result ==
pushFortranOutputStack(objfunFilename := aspFilename "objfun")$FOP
if objfunArg case fn
then outputAsFortran(objfunArg.fn)
else outputAsFortran(objfunArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([objfunFilename]$Lisp,_
"e04dgf",_
"list":S,"ma":S,"op":S,"pr":S,"sta":S_,
"user":S]$Lisp,
"user":S,"objfun":S]$Lisp,
"op":S,"objf":S,["objgrd":S,"n":S]$Lisp,["x":S,"n":S]$Lisp,_
["work":S,["*":S,13$Lisp,"n":S]$Lisp]$Lisp,["user":S,"*":S]$Lisp_,
"objfun":S]$Lisp_,
"sto":S,"ve":S,"iter":S,"ifail":S,["iwwork":S_,
["*":S,"n":S,1$Lisp]$Lisp]$Lisp,["iuser":S,"*":S]$Lisp]$Lisp_,
["logical":S,"list":S]$Lisp]$Lisp_]
]$Lisp$_
[(nArg::Any,esArg::Any,fuArg::Any,itArg::Any,linArg::Any_,
listArg::Any,maArg::Any,opArg::Any,prArg::Any,staArg::Any_,
stoArg::Any,veArg::Any,ifailArg::Any,xArg::Any ])
@List Any]$Lisp]$Lisp$_
pretend List (Record(key:Symbol,entry:Any))$Result

e04fdf(mArg:Integer,nArg:Integer,liwArg:Integer,_
1wArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer,_
lsfun1Arg:Union(fn:FileName,fp:Asp50(LSFUN1))): Result ==
pushFortranOutputStack(lsfun1Filename := aspFilename "lsfun1")$FOP
if lsfun1Arg case fn
then outputAsFortran(lsfun1Arg.fn)
else outputAsFortran(lsfun1Arg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([lsfun1Filename]$Lisp,_
"e04fdf",_
["m":S,"n":S,"liw":S,"1w":S,"fsumsq":S_,
"ifail":S,"lsfun1":S,"w":S,"x":S,"iw":S]$Lisp_,
["fsumsq":S,"w":S,"iw":S,"lsfun1":S]$Lisp_,
["double":S,"fsumsq":S,["w":S,"1w":S]$Lisp_}
CHAPTER 15. \textit{CHAPTER N}

\begin{verbatim}
2650

, ["x":S,"n":S]Lisp,"lsfun1":S]Lisp_
]Lisp_,
["fsumq":S,"w":S,"x":S,"ifail":S]Lisp_,
[("mArg::Any,nArg::Any,liwArg::Any,lwArg::Any_ ,ifailArg::Any,xArg::Any")_ @List Any][Lisp]Lisp_)
pretend List (Record(key:Symbol,entry:Any))\Result

\texttt{e04gcf(mArg:Integer,nArg:Integer,liwArg:Integer_,}
liwArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer_,
lsfun2Arg:Union(fn:FileName,fp:Asp19(LSFUN2))): Result ==
pushFortranOutputStack(lsfun2Filename := aspFilename "lsfun2")\FOP
if lsfun2Arg case fn
 then outputAsFortran(lsfun2Arg.fn)
 else outputAsFortran(lsfun2Arg.fp)
popFortranOutputStack()\FOP
[(invokeNagman([lsfun2Filename]$Lisp,_
 "e04gcf",_
 ["fsumq":S,"w":S,"x":S,"ifail":S]Lisp_,
 ["double":S,"fsumq":S,["w":S,"lw":S]Lisp_,
 ,["x":S,"n":S]Lisp,"lsfun2":S]Lisp_,
["fsumq":S,"w":S,"x":S,"ifail":S]Lisp_,
[("mArg::Any,nArg::Any,liwArg::Any,lwArg::Any_,
 ifailArg::Any,xArg::Any])_ @List Any][Lisp]Lisp_)
pretend List (Record(key:Symbol,entry:Any))\Result

\texttt{e04jaf(nArg:Integer,iboundArg:Integer,liwArg:Integer_,}
liwArg:Integer,blArg:Matrix DoubleFloat,буArg:Matrix DoubleFloat_,
xArg:Matrix DoubleFloat,ifailArg:Integer_,
funct1Arg:Union(fn:FileName,fp:Asp24(FUNCT1))): Result ==
pushFortranOutputStack(funct1Filename := aspFilename "funct1")\FOP
if funct1Arg case fn
 then outputAsFortran(funct1Arg.fn)
 else outputAsFortran(funct1Arg.fp)
popFortranOutputStack()\FOP
[(invokeNagman([funct1Filename]$Lisp,_
 "e04jaf",_
 ["f":S,"iw":S,"w":S,"funct1":S]Lisp_,
 ,["бу":S,"n":S]Lisp,["x":S,"n":S]Lisp_,
\end{verbatim}
pretend List (Record(key:Symbol,entry:Any))

result
qphessArg: Union(fn: FileName, fp: Asp20(QPHESS)) $ Result ==
pushFortranOutputStack(qphessFilename := aspFilename "qphess") $ FOP
  if qphessArg case fn
    then outputAsFortran(qphessArg.fn)
    else outputAsFortran(qphessArg.fp)
popFortranOutputStack() $ FOP
[(invokeNagman([qphessFilename]$Lisp,_
  "e04naf",_  
  "nrowa":S,"nrowh":S,"ncolh":S,"bigbnd":S,"cold":S_,
  "cvec":S,"feasol":S_,
  "work":S]$Lisp_,
  "work":S]$Lisp_)
$pretend List (Record(key: Symbol, entry: Any)) $ Result

e04ucf(nArg: Integer, nclinArg: Integer, ncnlnArg: Integer, _
  nrowaArg: Integer, nrowjArg: Integer, _
  ncolhArg: Integer, _
  aArg: Matrix DoubleFloat, blArg: Matrix DoubleFloat, _
  buArg: Matrix DoubleFloat, _
  livorkArg: Integer, lworkArg: Integer, staArg: Boolean, _
  craArg: DoubleFloat, derArg: Integer, feaArg: DoubleFloat, _
  funArg: DoubleFloat, hesArg: Boolean, infrArg: DoubleFloat, _
  infsArg: DoubleFloat, linfrArg: DoubleFloat, lintArg: DoubleFloat, _
  listArg: Boolean, majiArg: Integer, majpArg: Integer, _
def pushFortranOutputStack(filename: FileName, entry: FOP) :=
  pushFortranOutputStack(entry)

def pushFortranOutputStack(entry: FOP) :=
  pushFortranOutputStack(entry)

def main() :=
  pushFortranOutputStack()

def popFortranOutputStack() :=
  popFortranOutputStack()

def invokeNagman(lisp: Lisp, confun: Lisp, objfun: Lisp) :=
  invokeNagman(lisp, confun, objfun)

def invokeNagman(lisp: Lisp, confun: Lisp, objfun: Lisp) :=
  invokeNagman(lisp, confun, objfun)
This package uses the NAG Library to calculate the numerical solution of ordinary differential equations. There are two main types of problem, those in which all boundary conditions are specified at one point (initial-value problems), and those in which the boundary conditions are distributed between two or more points (boundary-value problems and eigenvalue problems). Routines are available for initial-value problems, two-point boundary-value problems and
CHAPTER 15.  CHAPTER N

Sturm-Liouville eigenvalue problems.

D02(3NAG)  Foundation Library (12/10/92)  D02(3NAG)

D02 -- Ordinary Differential Equations  Introduction -- D02
Chapter D02
Ordinary Differential Equations

1. Scope of the Chapter

This chapter is concerned with the numerical solution of ordinary differential equations. There are two main types of problem, those in which all boundary conditions are specified at one point (initial-value problems), and those in which the boundary conditions are distributed between two or more points (boundary-value problems and eigenvalue problems). Routines are available for initial-value problems, two-point boundary-value problems and Sturm-Liouville eigenvalue problems.

2. Background to the Problems

For most of the routines in this chapter a system of ordinary differential equations must be written in the form

\[ y' = f(x, y_1, y_2, \ldots, y_n), \]
\[ y'_1 = f(x, y_1, y_2, \ldots, y_n), \]
\[ \ldots \]
\[ y'_n = f(x, y_1, y_2, \ldots, y_n), \]

that is the system must be given in first-order form. The \( n \) dependent variables (also, the solution) \( y_1, y_2, \ldots, y_n \) are functions of the independent variable \( x \), and the differential equations give expressions for the first derivatives \( y'_i = dy_i / dx \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \). For a system of \( n \) first-order equations, \( n \) associated boundary conditions are usually required to define the solution.

A more general system may contain derivatives of higher order, but such systems can almost always be reduced to the first-order form by introducing new variables. For example, suppose we have the third-order equation
\[ z''' + 2z'' + k(1-z') = 0. \]

We write \( y = z, \ y = z', \ y = z'' \), and the third order equation may then be written as the system of first-order equations

\[
\begin{align*}
y' &= y_1 \\
y' &= y_2 \\
y' &= -y_2 y_3 - k(1-y_3).
\end{align*}
\]

For this system \( n = 3 \) and we require 3 boundary conditions in order to define the solution. These conditions must specify values of the dependent variables at certain points. For example, we have an initial-value problem if the conditions are:

\[
\begin{align*}
y_1 &= 0 \quad \text{at } x=0 \\
y_2 &= 0 \quad \text{at } x=0 \\
y_3 &= 0.1 \quad \text{at } x=0.
\end{align*}
\]

These conditions would enable us to integrate the equations numerically from the point \( x=0 \) to some specified end-point. We have a boundary-value problem if the conditions are:

\[
\begin{align*}
y_1 &= 0 \quad \text{at } x=0 \\
y_2 &= 0 \quad \text{at } x=0 \\
y_3 &= 1 \quad \text{at } x=10.
\end{align*}
\]

These conditions would be sufficient to define a solution in the range \( 0 \leq x \leq 10 \), but the problem could not be solved by direct integration (see Section 2.2). More general boundary conditions are permitted in the boundary-value case.
2.1. Initial-value Problems

To solve first-order systems, initial values of the dependent variables $y_i$, for $i=1,2,...,n$ must be supplied at a given point, $a$. Also a point, $b$, at which the values of the dependent variables are required, must be specified. The numerical solution is then obtained by a step-by-step calculation which approximates values of the variables $y_i$, for $i=1,2,...,n$ at finite intervals over the required range $[a,b]$. The routines in this chapter adjust the step length automatically to meet specified accuracy tolerances. Although the accuracy tests used are reliable over each step individually, in general an accuracy requirement cannot be guaranteed over a long range. For many problems there may be no serious accumulation of error, but for unstable systems small perturbations of the solution will often lead to rapid divergence of the calculated values from the true values. A simple check for stability is to carry out trial calculations with different tolerances; if the results differ appreciably the system is probably unstable. Over a short range, the difficulty may possibly be overcome by taking sufficiently small tolerances, but over a long range it may be better to try to reformulate the problem.

A special class of initial-value problems are those for which the solutions contain rapidly decaying transient terms. Such problems are called stiff; an alternative way of describing them is to say that certain eigenvalues of the Jacobian matrix ($ddf/ddy$) have large negative real parts when compared to others. These problems require special methods for efficient numerical solution; the methods designed for non-stiff problems when applied to stiff problems tend to be very slow, because they need small step lengths to avoid numerical instability. A full discussion is given in Hall and Watt [6] and a discussion of the methods for stiff problems is given in Berzins, Brankin and Gladwell [1].

2.2. Boundary-value Problems

A full discussion of the design of the methods and codes for boundary-value problems is given in Gladwell [4]. In general, a system of nonlinear differential equations with boundary conditions given at two or more points cannot be guaranteed to have a solution. The solution has to be determined iteratively (if it exists). Finite-difference equations are set up on a mesh of points and estimated values for the solution at the grid points are chosen. Using these estimated values as starting values a Newton iteration is used to solve the finite-difference equations. The accuracy of the solution is then improved by
deferred corrections or the addition of points to the mesh or a combination of both. Good initial estimates of the solution may be required in some cases but results may be difficult to compute when the solution varies very rapidly over short ranges. A discussion is given in Chapters 9 and 11 of Gladwell and Sayers [5] and Chapter 4 of Childs et al [2].

2.3. Eigenvalue Problems

Sturm-Liouville problems of the form

\[(p(x)y')' + q(x,\lambda)y = 0\]

with appropriate boundary conditions given at two points, can be solved by a Scaled Pruefer method. In this method the differential equation is transformed to another which can be solved for a specified eigenvalue by a shooting method. A discussion is given in Chapter 11 of Gladwell and Sayers [5] and a complete description is given in Pryce [7].

2.6. References


3. Recommendations on Choice and Use of Routines

There are no routines which deal directly with COMPLEX equations.
These may however be transformed to larger systems of real equations of the required form. Split each equation into its real and imaginary parts and solve for the real and imaginary parts of each component of the solution. Whilst this process doubles the size of the system and may not always be appropriate it does make available for use the full range of routines provided presently.

3.1. Initial-value Problems

For simple first-order problems with low accuracy requirements, that is problems on a short range of integration, with derivative functions \( f \) which are inexpensive to calculate and where only a few correct figures are required, the best routines to use are likely to be the Runge-Kutta-Merson (RK) routines, D02BBF and D02BHF. For larger problems, over long ranges or with high accuracy requirements the variable-order, variable-step Adams routine D02CJF should usually be preferred. For stiff equations, that is those with rapidly decaying transient solutions, the Backward Differentiation Formula (BDF) variable-order, variable-step routine D02EJF should be used.

There are four routines for initial-value problems, two of which use the Runge-Kutta-Merson method:

- **D02BBF**: integrates a system of first order ordinary differential equations over a range with intermediate output and a choice of error control.
- **D02BHF**: integrates a system of first order ordinary differential equations with a choice of error control until a position is determined where a function of the solution is zero.

one uses an Adams method:

- **D02CJF**: combines the functionality of D02BBF and D02BHF.

and one uses a BDF method:

- **D02EJF**: combines the functionality of D02BBF and D02BHF.

3.2. Boundary-value Problems

D02GAF may be used for simple boundary-value problems with assigned boundary values. The user may find that convergence is difficult to achieve using D02GAF since only specifying the unknown boundary values and the position of the finite-difference mesh is permitted. In such cases the user may use D02RAF which permits specification of an initial estimate for the solution at
all mesh points and allows the calculation to be influenced in
other ways too. D02RAF is designed to solve a general nonlinear
two-point boundary value problem with nonlinear boundary
conditions.

A routine, D02GBF, is also supplied specifically for the general
linear two-point boundary-value problem written in a standard

The user is advised to use interpolation routines from the E01
Chapter to obtain solution values at points not on the final
mesh.

3.3. Eigenvalue Problems

There is one general purpose routine for eigenvalue problems,
D02KEF. It may be used to solve regular or singular second-order
Sturm-Liouville problems on a finite or infinite range.
Discontinuous coefficient functions can be treated and
eigenfunctions can be computed.

D02 -- Ordinary Differential Equations

Chapter D02

Ordinary Differential Equations

D02BBF ODEs, IVP, Runge-Kutta-Merson method, over a range,
intermediate output

D02BHF ODEs, IVP, Runge-Kutta-Merson method, until function of
solution is zero

D02CJF ODEs, IVP, Adams method, until function of solution is
zero, intermediate output

D02EJF ODEs, stiff IVP, BDF method, until function of solution is
zero, intermediate output

D02GAF ODEs, boundary value problem, finite difference technique
with deferred correction, simple nonlinear problem

D02GBF ODEs, boundary value problem, finite difference technique
with deferred correction, general linear problem

D02KEF 2nd order Sturm-Liouville problem, regular/singular
system, finite/infinite range, eigenvalue and
eigenfunction, user-specified break-points

D02RAF ODEs, general nonlinear boundary value problem, finite
difference technique with deferred correction,
continuation facility

D02BBF(3NAG) D02BBF D02BBF(3NAG)

D02 -- Ordinary Differential Equations
D02BBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.

1. Purpose

D02BBF integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a Runge-Kutta-Merson method, and returns the solution at points specified by the user.

2. Specification

```fortran
SUBROUTINE D02BBF (X, XEND, M, N, Y, TOL, IRELAB, RESULT, FCN, OUTPUT, W, IFAIL)
INTEGER M, N, IRELAB, IFAIL
DOUBLE PRECISION X, XEND, Y(N), TOL, W(N,7), RESULT(M,N)
EXTERNAL FCN, OUTPUT
```

3. Description

The routine integrates a system of ordinary differential equations

\[ y_i' = f(x, y_1, y_2, \ldots, y_n) \quad i=1,2,\ldots,n \]

from \( x = X \) to \( x = XEND \) using a Merson form of the Runge-Kutta method. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \), and the values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

The solution is returned via the user-supplied routine OUTPUT at a set of points specified by the user. This solution is obtained
by quintic Hermite interpolation on solution values produced by
the Runge-Kutta method.

The accuracy of the integration and, indirectly, the
interpolation is controlled by the parameters TOL and IRELAB.

For a description of Runge-Kutta methods and their practical
implementation see Hall and Watt [1].

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input/Output
   On entry: X must be set to the initial value of the
   independent variable x. On exit: XEND, unless an error has
   occurred, when it contains the value of x at the error.

2: XEND -- DOUBLE PRECISION Input
   On entry: the final value of the independent variable. If
   XEND < X on entry, integration will proceed in a negative
direction.

3: M -- INTEGER Input
   On entry: the first dimension of the array RESULT. This
   will usually be equal to the number of points at which the
   solution is required.
   Constraint: M > 0.

4: N -- INTEGER Input
   On entry: the number of differential equations.
   Constraint: N > 0.

5: Y(N) -- DOUBLE PRECISION array Input/Output
   On entry: the initial values of the solution y_1, y_2, ..., y_n.
   On exit: the computed values of the solution at the final
   value of X.

6: TOL -- DOUBLE PRECISION Input/Output
   On entry: TOL must be set to a positive tolerance for
   controlling the error in the integration.

D02BBF has been designed so that, for most problems, a
reduction in TOL leads to an approximately proportional
reduction in the error in the solution at XEND. The relation
between changes in TOL and the error at intermediate output
points is less clear, but for TOL small enough the error at intermediate output points should also be approximately proportional to TOL. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02BBF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02BBF with TOL=10.0 and TOL=10.0 if p correct decimal digits in the solution are required. Constraint: TOL > 0.0. On exit: normally unchanged. However if the range X to XEND is so short that a small change in TOL is unlikely to make any change in the computed solution then, on return, TOL has its sign changed. This should be treated as a warning that the computed solution is likely to be more accurate than would be produced by using the same value of TOL on a longer range of integration.

7: IRELAB -- INTEGER Input
On entry: IRELAB determines the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:

IRELAB = 0
EST = 10.0 <= TOL * max{1.0, |y_1|, |y_2|, ..., |y_n|};

IRELAB = 1
EST <= TOL;

IRELAB = 2
EST <= TOL * max{epsilon, |y_1|, |y_2|, ..., |y_n|}, where

(epsilon) is machine precision.

If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step.

If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then IRELAB should be given the value 1 on entry, whereas if the error requirement is in terms of the number of correct significant digits, then IRELAB should be given the value 2. Where there is no preference in the choice of error test IRELAB = 0 will result in a mixed error test. Constraint: 0 <= IRELAB <= 2.

8: RESULT(M,N) -- DOUBLE PRECISION array Output
On exit: the computed values of the solution at the points
given by OUTPUT.

9: FCN -- SUBROUTINE, supplied by the user.

External Procedure

FCN must evaluate the functions \( f \) (i.e., the derivatives \( y' \)) for given values of its arguments \( x, y, \ldots, y \).

Its specification is:

```fortran
SUBROUTINE FCN (X, Y, F)
DOUBLE PRECISION X, Y(n), F(n)
```

where \( n \) is the actual value of \( N \) in the call of D02BBF.

1: X -- DOUBLE PRECISION
Input
On entry: the value of the argument \( x \).

2: Y(*) -- DOUBLE PRECISION array
Input
On entry: the value of the argument \( y \), for \( i=1,2,\ldots,n \).

3: F(*) -- DOUBLE PRECISION array
Output
On exit: the value of \( f \), for \( i=1,2,\ldots,n \).

FCN must be declared as EXTERNAL in the (sub)program from which D02BBF is called. Parameters denoted as Input must not be changed by this procedure.

10: OUTPUT -- SUBROUTINE, supplied by the user.

External Procedure

OUTPUT allows the user to have access to intermediate values of the computed solution at successive points specified by the user. These solution values may be returned to the user via the array RESULT if desired (this is a non-standard feature added for use with the AXIOM system). OUTPUT is initially called by D02BBF with XSOL = X (the initial value of \( x \)). The user must reset XSOL to the next point where OUTPUT is to be called, and so on at each call to OUTPUT. If, after a call to OUTPUT, the reset point XSOL is beyond XEND, D02BBF will integrate to XEND with no further calls to OUTPUT; if a call to OUTPUT is required at the point XSOL = XEND, then XSOL must be given precisely the value XEND.

Its specification is:

```fortran
SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
```
INTEGER M,N,COUNT

1: XSOL -- DOUBLE PRECISION  
   Input/Output
   On entry: the current value of the independent
   variable x. On exit: the next value of x at which
   OUTPUT is to be called.

2: Y(N) -- DOUBLE PRECISION array  
   Input
   On entry: the computed solution at the point XSOL.

3: COUNT -- INTEGER  
   Input/Output
   On entry: Zero if OUTPUT has not been called before, or
   the previous value of COUNT.
   On exit: A new value of COUNT: this can be used to keep
   track of the number of times OUTPUT has been called.

4: M -- INTEGER  
   Input
   On entry: The first dimension of RESULT.

5: N -- INTEGER  
   Input
   On entry: The dimension of Y.

6: RESULT(M,N) -- DOUBLE PRECISION array  
   Input/Output
   On entry: the previous contents of RESULT.
   On exit: RESULT may be used to return the values of the
   intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program
from which D02BBF is called. Parameters denoted as
Input must not be changed by this procedure.

11: W(N,7) -- DOUBLE PRECISION array  
    Workspace

12: IFAIL -- INTEGER  
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.

    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry TOL <= 0,
   or        N <= 0,
or \( \text{IRELAB} \neq 0, 1 \) or 2.

\textbf{IFAIL} = 2

With the given value of \( \text{TOL} \), no further progress can be made across the integration range from the current point \( x = X \), or the dependence of the error on \( \text{TOL} \) would be lost if further progress across the integration range were attempted (see Section 8 for a discussion of this error exit). The components \( Y(1), Y(2), \ldots, Y(n) \) contain the computed values of the solution at the current point \( x = X \).

\textbf{IFAIL} = 3

\( \text{TOL} \) is too small for the routine to take an initial step (see Section 8). \( X \) and \( Y(1), Y(2), \ldots, Y(n) \) retain their initial values.

\textbf{IFAIL} = 4

\( X = X_{\text{END}} \) and \( X_{\text{SOL}} \neq X \) after the initial call to OUTPUT.

\textbf{IFAIL} = 5

A value of \( X_{\text{SOL}} \) returned by OUTPUT lies behind the previous value of \( X_{\text{SOL}} \) in the direction of integration.

\textbf{IFAIL} = 6

A serious error has occurred in an internal call to D02PAF(*). Check all subroutine calls and array dimensions. Seek expert help.

\textbf{IFAIL} = 7

A serious error has occurred in an internal call to D02XAF(*). Check all subroutine calls and array dimensions. Seek expert help.

\textbf{7. Accuracy}

The accuracy depends on \( \text{TOL} \), on the mathematical properties of the differential system, on the length of the range of integration and on the method. It can be controlled by varying \( \text{TOL} \) but the approximate proportionality of the error to \( \text{TOL} \) holds only for a restricted range of values of \( \text{TOL} \). For \( \text{TOL} \) too large, the underlying theory may break down and the result of varying \( \text{TOL} \) may be unpredictable. For \( \text{TOL} \) too small, rounding errors may affect the solution significantly and an error exit with \( \text{IFAIL} = 2 \) or \( \text{IFAIL} = 3 \) is possible.

At the intermediate output points the same remarks apply. For large values of \( \text{TOL} \) it is possible that the errors at some intermediate output points may be much larger than at \( X_{\text{END}} \). In any case, it must not be expected that the error will have the same size at all output points. At any point, it is a combination
of the errors arising from the integration of the differential
equation and the interpolation. The effect of combining these
errors will vary, though in most cases the integration error will
dominate.

The user who requires a more reliable estimate of the accuracy
achieved than can be obtained by varying TOL, is recommended to
call D02BDF(*) where both the solution and a global error
estimate are computed.

8. Further Comments

The time taken by the routine depends on the complexity and
mathematical properties of the system of differential equations
defined by FCN, on the range, the tolerance and the number of
calls to OUTPUT. There is also an overhead of the form a+b*n
where a and b are machine-dependent computing times.

If the routine fails with IFAIL = 3, then it can be called again
with a larger value of TOL (if this has not already been tried).
If the accuracy requested is really needed and cannot be obtained
with this routine, the system may be very stiff (see below) or so
badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is probable that it has
been called with a value of TOL which is so small that the
solution cannot be obtained on the range X to XEND. This can
happen for well-behaved systems and very small values of TOL. The
user should, however, consider whether there is a more
fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the
solution, the routine will usually stop with IFAIL = 2,
unless overflow occurs first. If overflow occurs using
D02BBF, D02PAF(*) can be used instead to trap the
increasing solution before overflow occurs. In any case,
numerical integration cannot be continued through a
singularity, and analytic treatment should be considered;

(b) for 'stiff' equations, where the solution contains rapidly
decaying components, the routine will use very small steps
in x (internally to D02BBF) to preserve stability. This
will usually exhibit itself by making the computing time
excessively long, or occasionally by an exit with IFAIL =
2. Merson's method is not efficient in such cases, and the
user should try using D02EBF(*) (Backward Differentiation
Formula). To determine whether a problem is stiff,
D02BDF(*) may be used.

For well-behaved systems with no difficulties such as stiffness
or singularities, the Merson method should work well for low accuracy calculations (three or four figures). For higher accuracy calculations or where FCN is costly to evaluate, Merson’s method may not be appropriate and a computationally less expensive method may be D02CBF(*) which uses an Adams method.

Users with problems for which D02BBF is not sufficiently general should consider using D02PAF(*) with D02XAF(*). D02PAF(*) is a more general Merson routine with many facilities including more general error control options and several criteria for interrupting the calculations. D02XAF(*) interpolates on values produced by D02PAF(*)..

9. Example

To integrate the following equations (for a projectile)

\[
y' = \tan(\phi)
\]

\[
-0.032\tan(\phi) \quad \frac{0.02v}{v \cos(\phi)}
\]

\[
\phi' = \frac{-0.032}{2v}
\]

over an interval \( X = 0.0 \) to \( X_{END} = 8.0 \), starting with values \( y=0.0, v=0.5 \) and \( (\phi) = (\pi)/5 \) and printing the solution at steps of 1.0. We write \( y=Y(1), v=Y(2) \) and \( (\phi)=Y(3) \), and we set \( TOL=1.0E-4 \) and \( TOL=1.0E-5 \) in turn so that we may compare the solutions. The value of \( (\pi) \) is obtained by using X01AAF(*).

Note the careful construction of routine OUT to ensure that the value of \( X_{END} \) is printed.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D02BHF integrates a system of first-order ordinary differential equations over an interval with suitable initial conditions, using a Runge-Kutta-Merson method, until a user-specified function of the solution is zero.

2. Specification

```fortran
SUBROUTINE D02BHF (X, XEND, N, Y, TOL, IRELAB, HMAX, FCN, G, W, IFAIL)
INTEGER N, IRELAB, IFAIL
DOUBLE PRECISION X, XEND, Y(N), TOL, HMAX, G, W(N,7)
EXTERNAL FCN, G
```

3. Description

The routine advances the solution of a system of ordinary differential equations

\[ y'_i = f(x, y_1, y_2, \ldots, y_n), \quad i = 1, 2, \ldots, n, \]

from \( x = X \) towards \( x = XEND \) using a Merson form of the Runge-Kutta method. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \) (see Section 5), and the values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

As the integration proceeds, a check is made on the function \( g(x, y) \) specified by the user, to determine an interval where it changes sign. The position of this sign change is then determined accurately by interpolating for the solution and its derivative. It is assumed that \( g(x, y) \) is a continuous function of the variables, so that a solution of \( g(x, y) = 0 \) can be determined by searching for a change in sign in \( g(x, y) \).

The accuracy of the integration and, indirectly, of the determination of the position where \( g(x, y) = 0 \), is controlled by the parameter TOL.

For a description of Runge-Kutta methods and their practical implementation see Hall and Watt [1].
4. References


5. Parameters

1: X -- DOUBLE PRECISION Input/Output
   On entry: X must be set to the initial value of the independent variable x. On exit: the point where g(x,y) = 0. 0 unless an error has occurred, when it contains the value of x at the error. In particular, if g(x,y)/=0.0 anywhere on the range X to XEND, it will contain XEND on exit.

2: XEND -- DOUBLE PRECISION Input
   On entry: the final value of the independent variable x.

   If XEND < X on entry, integration proceeds in a negative direction.

3: N -- INTEGER Input
   On entry: the number of differential equations, n.
   Constraint: N > 0.

4: Y(N) -- DOUBLE PRECISION array Input/Output
   On entry: the initial values of the solution y_1, y_2, ..., y_n.
   On exit: the computed values of the solution at the final point x = X.

5: TOL -- DOUBLE PRECISION Input/Output
   On entry: TOL must be set to a positive tolerance for controlling the error in the integration and in the determination of the position where g(x,y) = 0.0.

   D02BHF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution obtained in the integration. The relation between changes in TOL and the error in the determination of the position where g(x,y) = 0 is less clear, but for TOL small enough the error should be approximately proportional to TOL. However, the actual relation between TOL and the accuracy cannot be guaranteed. The user is strongly recommended to call D02BHF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge the user might compare results obtained by calling D02BHF with TOL=10.0 and TOL=10.0 if p correct decimal digits in the solution are required. Constraint: TOL > 0.0.
On exit: normally unchanged. However if the range from \( x = X \) to the position where \( g(x,y) = 0.0 \) (or to the final value of \( x \) if an error occurs) is so short that a small change in \( TOL \) is unlikely to make any change in the computed solution, then \( TOL \) is returned with its sign changed. To check results returned with \( TOL < 0.0 \), D02BHF should be called again with a positive value of \( TOL \) whose magnitude is considerably smaller than that of the previous call.

6: \( \text{IRELAB} \) -- INTEGER Input
On entry: IRELAB determines the type of error control. At each step in the numerical solution an estimate of the local error, \( EST \), is made. For the current step to be accepted the following condition must be satisfied:

\[
\begin{align*}
\text{IRELAB} = 0 & \quad \text{EST} \leq TOL \times \max\{1.0, |y_1|, |y_2|, \ldots, |y_n|\}; \\
\text{IRELAB} = 1 & \quad \text{EST} \leq TOL; \\
\text{IRELAB} = 2 & \quad \text{EST} \leq TOL \times \max\{\text{epsilon}, |y_1|, |y_2|, \ldots, |y_n|\},
\end{align*}
\]

where (epsilon) is machine precision.
If the appropriate condition is not satisfied, the step size is reduced and the solution recomputed on the current step.

If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then IRELAB should be given the value 1 on entry, whereas if the error requirement is in terms of the number of correct significant digits, then IRELAB should be given the value 2.
Where there is no preference in the choice of error test, IRELAB = 0 will result in a mixed error test. It should be borne in mind that the computed solution will be used in evaluating \( g(x,y) \). Constraint: 0 \( \leq \) IRELAB \( \leq \) 2.

7: \( \text{HMAX} \) -- DOUBLE PRECISION Input
On entry: if HMAX = 0.0, no special action is taken.

If \( HMAX \neq 0.0 \), a check is made for a change in sign of \( g(x,y) \) at steps not greater than \( |HMAX| \). This facility should be used if there is any chance of 'missing' the change in sign by checking too infrequently. For example, if two changes of sign of \( g(x,y) \) are expected within a distance \( h \), say, of each other, then a suitable value for \( HMAX \) might be \( HMAX = h/2 \). If only one change of sign in \( g(x,y) \) is
expected on the range X to XEND, then the choice HMAX = 0.0
is most appropriate.

8: FCN -- SUBROUTINE, supplied by the user.

FCN must evaluate the functions \( f \) (i.e., the derivatives
\( y' \)) for given values of its arguments \( x, y, \ldots, y \).
\( i \quad 1 \quad n \)

Its specification is:

```fortran
SUBROUTINE FCN (X, Y, F)
DOUBLE PRECISION X, Y(n), F(n)

1: X -- DOUBLE PRECISION
   On entry: the value of the argument x.

2: Y(*) -- DOUBLE PRECISION array
   On entry: the value of the argument \( y_i \), for
   \( i = 1, 2, \ldots, n \).

3: F(*) -- DOUBLE PRECISION array
   On exit: the value of \( f_i \), for \( i = 1, 2, \ldots, n \).

FCN must be declared as EXTERNAL in the (sub)program
from which D02BHF is called. Parameters denoted as
Input must not be changed by this procedure.

9: G -- DOUBLE PRECISION FUNCTION, supplied by the user.

G must evaluate the function \( g(x,y) \) at a specified point.

Its specification is:

```fortran
DOUBLE PRECISION FUNCTION G (X, Y)
DOUBLE PRECISION X, Y(n)

where n is the actual value of N in the call of D02BHF.

1: X -- DOUBLE PRECISION
   On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array
   On entry: the value of \( y_i \), for \( i = 1, 2, \ldots, n \).

G must be declared as EXTERNAL in the (sub)program from
which D02BHF is called. Parameters denoted as Input
must not be changed by this procedure.
CHAPTER 15. CHAPTER N

10: W(N,7) -- DOUBLE PRECISION array
     Workspace

11: IFAIL -- INTEGER
     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
     On entry TOL <= 0.0,
     or N <= 0,
     or IRELAB /= 0, 1 or 2.

IFAIL= 2
     With the given value of TOL, no further progress can be made
     across the integration range from the current point x = X,
     or dependence of the error on TOL would be lost if further
     progress across the integration range were attempted (see
     Section 8 for a discussion of this error exit). The
     components Y(1),Y(2),...,Y(n) contain the computed values of
     the solution at the current point x = X. No point at which
     g(x,y) changes sign has been located up to the point x = X.

IFAIL= 3
     TOL is too small for the routine to take an initial step
     (see Section 8). X and Y(1),Y(2),...,Y(n) retain their
     initial values.

IFAIL= 4
     At no point in the range X to XEND did the function g(x,y)
     change sign. It is assumed that g(x,y) = 0.0 has no
     solution.

IFAIL= 5
     A serious error has occurred in an internal call to
     C05AZF(*). Check all subroutine calls and array dimensions.
     Seek expert help.

IFAIL= 6
     A serious error has occurred in an internal call to
     D02PAF(*). Check all subroutine calls and array dimensions.
     Seek expert help.
IFAIL= 7
A serious error has occurred in an internal call to
D02XAF(*). Check all subroutine calls and array dimensions.
Seek expert help.

7. Accuracy

The accuracy depends on TOL, on the mathematical properties of
the differential system, on the position where g(x,y) = 0.0 and
on the method. It can be controlled by varying TOL but the
approximate proportionality of the error to TOL holds only for a
restricted range of values of TOL. For TOL too large, the
underlying theory may break down and the result of varying TOL
may be unpredictable. For TOL too small, rounding error may
affect the solution significantly and an error exit with IFAIL =
2 or IFAIL = 3 is possible.

The accuracy may also be restricted by the properties of g(x,y).
The user should try to code G without introducing any unnecessary
cancellation errors.

8. Further Comments

The time taken by the routine depends on the complexity and
mathematical properties of the system of differential equations
defined by FCN, the complexity of G, on the range, the position
of the solution and the tolerance. There is also an overhead of
the form a+b*n where a and b are machine-dependent computing
times.

For some problems it is possible that D02BHF will return IFAIL =
4 because of inaccuracy of the computed values Y, leading to
inaccuracy in the computed values of g(x,y) used in the search
for the solution of g(x,y) = 0.0. This difficulty can be overcome
by reducing TOL sufficiently, and if necessary, by choosing HMAX
sufficiently small. If possible, the user should choose XEND well
beyond the expected point where g(x,y) = 0.0; for example make
|XEND-X| about 50 larger than the expected range. As a simple
check, if, with XEND fixed, a change in TOL does not lead to a
significant change in Y at XEND, then inaccuracy is not a likely
source of error.

If the routine fails with IFAIL = 3, then it could be called
again with a larger value of TOL if this has not already been
tried. If the accuracy requested is really needed and cannot be
obtained with this routine, the system may be very stiff (see
below) or so badly scaled that it cannot be solved to the
required accuracy.
If the routine fails with IFAIL = 2, it is likely that it has been called with a value of TOL which is so small that a solution cannot be obtained on the range X to XEND. This can happen for well-behaved systems and very small values of TOL. The user should, however, consider whether there is a more fundamental difficulty. For example:

(a) in the region of a singularity (infinite value) of the solution, the routine will usually stop with IFAIL = 2, unless overflow occurs first. If overflow occurs using D02BHF, D02PAF(*) can be used instead to trap the increasing solution, before overflow occurs. In any case, numerical integration cannot be continued through a singularity, and analytical treatment should be considered;

(b) for 'stiff' equations, where the solution contains rapidly decaying components, the routine will compute in very small steps in x (internally to D02BHF) to preserve stability. This will usually exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Merson's method is not efficient in such cases, and the user should try D02EHF(*) which uses a Backward Differentiation Formula method. To determine whether a problem is stiff, D02BDF(*) may be used.

For well-behaved systems with no difficulties such as stiffness or singularities, the Merson method should work well for low accuracy calculations (three or four figures). For high accuracy calculations or where FCN is costly to evaluate, Merson's method may not be appropriate and a computationally less expensive method may be D02CHF(*) which uses an Adams method.

For problems for which D02BHF is not sufficiently general, the user should consider D02PAF(*). D02PAF(*) is a more general Merson routine with many facilities including more general error control options and several criteria for interrupting the calculations. D02PAF(*) can be combined with the rootfinder C05AZF(*) and the interpolation routine D02XAF(*) to solve equations involving $y_1, y_2, \ldots, y_n$ and their derivatives.

D02BHF can also be used to solve an equation involving x, $y_1, y_2, \ldots, y_n$ and the derivatives of these $y_1, y_2, \ldots, y_n$. For example in Section 9, D02BHF is used to find a value of $X > 0.0$ where $Y(1) = 0.0$. It could instead be used to find a turning-point of $y_1$ by replacing the function $g(x,y)$ in the program by:

```fortran
DOUBLE PRECISION FUNCTION G(X,Y)
```
DOUBLE PRECISION X,Y(3),F(3)
CALL FCN(X,Y,F)
   G = F(1)
RETURN
END

This routine is only intended to locate the first zero of g(x,y). If later zeros are required, users are strongly advised to construct their own more general root finding routines as discussed above.

9. Example

To find the value X > 0.0 at which y=0.0, where y, v, (phi) are defined by

\[ y' = \tan(\phi) \]
\[ v' = \frac{-0.032\tan(\phi) + 0.02v}{\cos(\phi)} \]
\[ (\phi)' = -\frac{0.032}{v} \]

and where at X = 0.0 we are given y=0.5, v=0.5 and (phi)=(\pi)/5. We write y=Y(1), v=Y(2) and (phi)=Y(3) and we set TOL=1.0E-4 and TOL=1.0E-5 in turn so that we can compare the solutions. We expect the solution X ~= 7.3 and so we set XEND = 10.0 to avoid determining the solution of y=0.0 too near the end of the range of integration. The value of (\pi) is obtained by using X01AAF(*).

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
D02CJF(3NAG)  D02CJF  D02CJF(3NAG)
D02 -- Ordinary Differential Equations  D02CJF
D02CJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.

1. Purpose

D02CJF integrates a system of first-order ordinary differential equations over a range with suitable initial conditions, using a variable-order, variable-step Adams method until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.

2. Specification

SUBROUTINE D02CJF (X, XEND, M, N, Y, FCN, TOL, RELABS, 1 RESULT, OUTPUT, G, W, IFAIL)

INTEGER M, N, IFAIL
DOUBLE PRECISION X, XEND, Y(N), TOL, G, W(28+21*N), RESULT(M,N)
CHARACTER*1 RELABS
EXTERNAL FCN, OUTPUT, G

3. Description

The routine advances the solution of a system of ordinary differential equations

\[ y'_i = f(x, y_1, y_2, \ldots, y_n), \quad i=1,2,\ldots,n, \]

from \( x = X \) to \( x = XEND \) using a variable-order, variable-step Adams method. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y_1, y_2, \ldots, y_n \).

The initial values of \( y_1, y_2, \ldots, y_n \) must be given at \( x = X \).

The solution is returned via the user-supplied routine OUTPUT at points specified by the user, if desired; this solution is obtained by C interpolation on solution values produced by the method. As the integration proceeds a check can be made on the user-specified function \( g(x,y) \) to determine an interval where it changes sign. The position of this sign change is then determined accurately by C interpolation to the solution. It is assumed that \( g(x,y) \) is a continuous function of the variables, so that a solution of \( g(x,y) = 0.0 \) can be determined by searching for a change in sign in \( g(x,y) \). The accuracy of the integration, the interpolation and, indirectly, of the determination of the
position where \( g(x,y) = 0.0 \), is controlled by the parameters TOL and RELABS.

For a description of Adams methods and their practical implementation see Hall and Watt [1].

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION
   Input/Output
   On entry: the initial value of the independent variable \( x \).
   Constraint: \( X \neq XEND \). On exit: if \( g \) is supplied by the user, it contains the point where \( g(x,y) = 0.0 \), unless \( g(x,y) 
eq 0.0 \) anywhere on the range \( X \) to \( XEND \), in which case, \( X \) will contain \( XEND \). If \( g \) is not supplied by the user it contains \( XEND \), unless an error has occurred, when it contains the value of \( x \) at the error.

2: \( XEND \) -- DOUBLE PRECISION
   Input
   On entry: the final value of the independent variable. If \( XEND < X \), integration proceeds in the negative direction.
   Constraint: \( XEND \neq X \).

3: \( M \) -- INTEGER
   Input
   On entry: the first dimension of the array RESULT. This will usually be equal to the number of points at which the solution is required.
   Constraint: \( M > 0 \).

4: \( N \) -- INTEGER
   Input
   On entry: the number of differential equations.
   Constraint: \( N \geq 1 \).

5: \( Y(N) \) -- DOUBLE PRECISION array
   Input/Output
   On entry: the initial values of the solution \( y_1, y_2, \ldots, y_n \) at \( x = X \). On exit: the computed values of the solution at the final point \( x = X \).

6: FCN -- SUBROUTINE, supplied by the user.
   External Procedure
   FCN must evaluate the functions \( f \) (i.e., the derivatives \( y' \)) for given values of their arguments \( x, y_1, y_2, \ldots, y_n \).
Its specification is:

```fortran
SUBROUTINE FCN (X, Y, F)
DOUBLE PRECISION X, Y(n), F(n)
where n is the actual value of N in the call of D02CJF.
```

1: X -- DOUBLE PRECISION  Input
On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array  Input
On entry: the value of the variable y , for
i
i=1,2,...,n.

3: F(*) -- DOUBLE PRECISION array  Output
On exit: the value of f , for i=1,2,...,n.

FCN must be declared as EXTERNAL in the (sub)program from which D02CJF is called. Parameters denoted as Input must not be changed by this procedure.

7: TOL -- DOUBLE PRECISION  Input
On entry: a positive tolerance for controlling the error in the integration. Hence TOL affects the determination of the position where g(x,y)=0.0, if g is supplied.

D02CJF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02CJF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02CJF with TOL=10.0 and TOL=10.0 where p correct decimal digits are required in the solution.

Constraint: TOL > 0.0.

8: RELABS -- CHARACTER*1  Input
On entry: the type of error control. At each step in the numerical solution an estimate of the local error, EST, is made. For the current step to be accepted the following condition must be satisfied:

```plaintext
/ n
/ -- 2
EST= / > (e /((tau) *|y |+(tau) )) <=1.0
/ -- i  r i a
/ \ i=1
```
where \((\tau)\) and \((\tau)\) are defined by
\[
\begin{align*}
\text{RELABS} & \quad \tau & \quad \tau \\
\text{r} & \quad \text{TOL} & \quad \text{TOL} \\
\text{a} & \quad \text{TOL} & \quad (\text{epsilon}) \\
\text{'M'} & \quad \text{TOL} & \quad \text{TOL} \\
\text{'A'} & \quad 0.0 & \quad \text{TOL} \\
\text{'R'} & \quad \text{TOL} & \quad (\text{epsilon}) \\
\text{'D'} & \quad \text{TOL} & \quad \text{TOL}
\end{align*}
\]
where \((\text{epsilon})\) is a small machine-dependent number and \(e_i\) is an estimate of the local error at \(y\), computed internally. If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step. If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then RELABS should be set to 'A' on entry, whereas if the error requirement is in terms of the number of correct significant digits, then RELABS should be set to 'R'. If the user prefers a mixed error test, then RELABS should be set to 'M', otherwise if the user has no preference, RELABS should be set to the default 'D'. Note that in this case 'D' is taken to be 'M'. Constraint: RELABS = 'M', 'A', 'R', 'D'.

9: RESULT(M,N) -- DOUBLE PRECISION array
Output
On exit: the computed values of the solution at the points given by OUTPUT.

10: OUTPUT -- SUBROUTINE, supplied by the user.
External Procedure
OUTPUT allows the user to have access to intermediate values of the computed solution at successive points specified by the user. These solution values may be returned to the user via the array RESULT if desired (this is a non-standard feature added for use with the AXIOM system). OUTPUT is initially called by D02CJF with XSOL = X (the initial value of x). The user must reset XSOL to the next point where OUTPUT is to be called, and so on at each call to OUTPUT. If, after a call to OUTPUT, the reset point XSOL is beyond XEND, D02CJF will integrate to XEND with no further calls to OUTPUT; if a call to OUTPUT is required at the point XSOL = XEND, then XSOL must be given precisely the value XEND.

Its specification is:
SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
INTEGER M,N,COUNT

1: XSOL -- DOUBLE PRECISION  
Input/Output
On entry: the current value of the independent
variable x. On exit: the next value of x at which
OUTPUT is to be called.

2: Y(N) -- DOUBLE PRECISION array  
Input
On entry: the computed solution at the point XSOL.

3: COUNT -- INTEGER  
Input/Output
On entry: Zero if OUTPUT has not been called before, or
the previous value of COUNT.
On exit: A new value of COUNT: this can be used to keep
track of the number of times OUTPUT has been called.

4: M -- INTEGER  
Input
On entry: The first dimension of RESULT.

5: N -- INTEGER  
Input
On entry: The dimension of Y.

6: RESULT(M,N) -- DOUBLE PRECISION array  
Input/Output
On entry: the previous contents of RESULT.
On exit: RESULT may be used to return the values of the
intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program
from which D02CJF is called. Parameters denoted as
Input must not be changed by this procedure.

11: G -- DOUBLE PRECISION FUNCTION, supplied by the user.
Internal Procedure
G must evaluate the function g(x,y) for specified values x,y
. It specifies the function g for which the first position x
where g(x,y)=0 is to be found.

If the user does not require the root finding option, the
actual argument G must be the dummy routine D02CJW. (D02CJW
is included in the NAG Foundation Library and so need not be
supplied by the user).

Its specification is:

DOUBLE PRECISION FUNCTION G (X, Y)
DOUBLE PRECISION X, Y(n)
where n is the actual value of N in the call of D02CJF.
1: X -- DOUBLE PRECISION  
   On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array  
   On entry: the value of the variable y, for
   i
   i=1,2,...,n.
   G must be declared as EXTERNAL in the (sub)program from
   which D02CJF is called. Parameters denoted as Input
   must not be changed by this procedure.

12: W(28+21*N) -- DOUBLE PRECISION array  
    Workspace

13: IFAIL -- INTEGER  
    Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    
    On exit: IFAIL = 0 unless the routine detects an error (see
    Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   On entry TOL <= 0.0,
   or N <= 0,
   or RELABS /= 'M', 'A', 'R' or 'D'.
   or X = XEND.

IFAIL = 2
   With the given value of TOL, no further progress can be made
   across the integration range from the current point x = X.
   (See Section 8 for a discussion of this error exit.) The
   components Y(1),Y(2),...,Y(N) contain the computed values of
   the solution at the current point x = X. If the user has
   supplied g, then no point at which g(x,y) changes sign has
   been located up to the point x = X.

IFAIL = 3
   TOL is too small for D02CJF to take an initial step. X and Y
   (1),Y(2),...,Y(N) retain their initial values.
IFAIL = 4
X SOL has not been reset or X SOL lies behind X in the
direction of integration, after the initial call to OUTPUT,
if the OUTPUT option was selected.

IFAIL = 5
A value of X SOL returned by OUTPUT has not been reset or
lies behind the last value of X SOL in the direction of
integration, if the OUTPUT option was selected.

IFAIL = 6
At no point in the range X to XEND did the function g(x,y)
change sign, if g was supplied. It is assumed that g(x,y)=0
has no solution.

IFAIL = 7
A serious error has occurred in an internal call. Check all
subroutine calls and array sizes. Seek expert help.

7. Accuracy
The accuracy of the computation of the solution vector Y may be
controlled by varying the local error tolerance TOL. In general,
a decrease in local error tolerance should lead to an increase in
accuracy. Users are advised to choose RELABS = 'M' unless they
have a good reason for a different choice.

If the problem is a root-finding one, then the accuracy of the
root determined will depend on the properties of g(x,y). The user
should try to code G without introducing any unnecessary
cancellation errors.

8. Further Comments
If more than one root is required then D02QFF(*) should be used.

If the routine fails with IFAIL = 3, then it can be called again
with a larger value of TOL if this has not already been tried. If
the accuracy requested is really needed and cannot be obtained
with this routine, the system may be very stiff (see below) or so
badly scaled that it cannot be solved to the required accuracy.

If the routine fails with IFAIL = 2, it is probable that it has
been called with a value of TOL which is so small that a solution
cannot be obtained on the range X to XEND. This can happen for
well-behaved systems and very small values of TOL. The user
should, however, consider whether there is a more fundamental
difficulty. For example:

(a) in the region of a singularity (infinite value) of the
solution, the routine will usually stop with IFAIL = 2, unless overflow occurs first. Numerical integration cannot be continued through a singularity, and analytic treatment should be considered;

(b) for 'stiff' equations where the solution contains rapidly decaying components, the routine will use very small steps in x (internally to D02CJF) to preserve stability. This will exhibit itself by making the computing time excessively long, or occasionally by an exit with IFAIL = 2. Adams methods are not efficient in such cases, and the user should try D02EJF.

9. Example

We illustrate the solution of four different problems. In each case the differential system (for a projectile) is

\[ y' = \tan(\phi) \]
\[ v' = \frac{-0.032\tan(\phi) + 0.02v}{v \cos(\phi)} \]
\[ (\phi)' = \frac{-0.032}{2v} \]

over an interval \( X = 0.0 \) to \( X_{\text{END}} = 10.0 \) starting with values \( y=0.5, v=0.5 \) and \( \phi=(\pi)/5 \). We solve each of the following problems with local error tolerances \( 1.0\times10^{-4} \) and \( 1.0\times10^{-5} \).

(i) To integrate to \( x=10.0 \) producing output at intervals of 2.0 until a point is encountered where \( y=0.0 \).

(ii) As (i) but with no intermediate output.

(iii) As (i) but with no termination on a root-finding condition.

(iv) As (i) but with no intermediate output and no root-finding termination condition.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D02 -- Ordinary Differential Equations
D02EJF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

Note for users via the AXIOM system: the interface to this routine has been enhanced for use with AXIOM and is slightly different to that offered in the standard version of the Foundation Library.

1. Purpose

D02EJF integrates a stiff system of first-order ordinary differential equations over an interval with suitable initial conditions, using a variable-order, variable-step method implementing the Backward Differentiation Formulae (BDF), until a user-specified function, if supplied, of the solution is zero, and returns the solution at points specified by the user, if desired.

2. Specification

SUBROUTINE D02EJF (X, XEND, M, N, Y, FCN, PEDERV, TOL, RELABS, OUTPUT, G, W, IW, RESULT, IFAIL)
INTEGER M, N, IW, IFAIL
DOUBLE PRECISION X, XEND, Y(N), TOL, G, W(IW), RESULT(M,N)
CHARACTER*1 RELABS
EXTERNAL FCN, PEDERV, OUTPUT, G

3. Description

The routine advances the solution of a system of ordinary differential equations

\[ y' = f(x, y_1, y_2, \ldots, y_n), \quad i = 1, 2, \ldots, n, \]

from \( x = X \) to \( x = XEND \) using a variable-order, variable-step method implementing the BDF. The system is defined by a subroutine FCN supplied by the user, which evaluates \( f \) in terms of \( x \) and \( y, y, \ldots, y \) (see Section 5). The initial values of \( y, y, \ldots, y \) must be given at \( x = X \).

The solution is returned via the user-supplied routine OUTPUT at
points specified by the user, if desired: this solution is
obtained by C interpolation on solution values produced by the
method. As the integration proceeds a check can be made on the
user-specified function $g(x,y)$ to determine an interval where it
changes sign. The position of this sign change is then determined
accurately by C interpolation to the solution. It is assumed
that $g(x,y)$ is a continuous function of the variables, so that a
solution of $g(x,y) = 0.0$ can be determined by searching for a
change in sign in $g(x,y)$. The accuracy of the integration, the
interpolation and, indirectly, of the determination of the
position where $g(x,y) = 0.0$, is controlled by the parameters TOL
and RELABS. The Jacobian of the system $y'=f(x,y)$ may be supplied
in routine PEDERV, if it is available.

For a description of BDF and their practical implementation see
Hall and Watt [1].

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION    Input/Output
   On entry: the initial value of the independent variable $x$.
   Constraint: $X \neq XEND$
   On exit: if $G$ is supplied by the user, $X$ contains the point where $g(x,y) = 0.0$, unless $g(x,y) \neq 0$.
   anywhere on the range $X$ to $XEND$, in which case, $X$ will
   contain $XEND$. If $G$ is not supplied $X$ contains $XEND$, unless
   an error has occurred, when it contains the value of $x$ at the
   error.

2: $XEND$ -- DOUBLE PRECISION    Input
   On entry: the final value of the independent variable. If
   $XEND < X$, integration proceeds in the negative direction.
   Constraint: $XEND \neq X$.

3: $M$ -- INTEGER       Input
   On entry: the first dimension of the array RESULT. This
   will usually be equal to the number of points at which the
   solution is required.
   Constraint: $M > 0$.

4: $N$ -- INTEGER    Input
   On entry: the number of differential equations, $n$.
   Constraint: $N \geq 1$. 
5: Y(N) -- DOUBLE PRECISION array
   Input/Output
   On entry: the initial values of the solution \( y_1, y_2, \ldots, y_n \)
   at \( x = X \). On exit: the computed values of the solution at
   the final point \( x = X \).

6: FCN -- SUBROUTINE, supplied by the user.
   External Procedure
   FCN must evaluate the functions \( f \) (i.e., the derivatives
   \( y' \)) for given values of their arguments \( x, y_1, y_2, \ldots, y_n \).
   Its specification is:

   ```
   SUBROUTINE FCN (X, Y, F)
   DOUBLE PRECISION X, Y(n), F(n)
   where n is the actual value of N in the call of D02EJF.
   ```

   1: X -- DOUBLE PRECISION
      Input
      On entry: the value of the independent variable \( x \).

   2: Y(*) -- DOUBLE PRECISION array
      Input
      On entry: the value of the variables \( y_i \), for
      \( i = 1, 2, \ldots, n \).

   3: F(*) -- DOUBLE PRECISION array
      Output
      On exit: the value of \( f_i \), for \( i = 1, 2, \ldots, n \).

   FCN must be declared as EXTERNAL in the (sub)program
   from which D02EJF is called. Parameters denoted as
   Input must not be changed by this procedure.

7: PEDERV -- SUBROUTINE, supplied by the user.
   External Procedure
   PEDERV must evaluate the Jacobian of the system (that is,
   \( \frac{df}{dy} \)) for given values of the
   variables \( x, y_1, y_2, \ldots, y_n \).
   Its specification is:

   ```
   SUBROUTINE PEDERV (X, Y, PW)
   DOUBLE PRECISION X, Y(n), PW(n,n)
   where n is the actual value of N in the call of D02EJF.
   ```
1: X -- DOUBLE PRECISION  
   On entry: the value of the independent variable x.

2: Y(*) -- DOUBLE PRECISION array  
   On entry: the value of the variable y, for i
           i=1,2,...,n.

3: PW(n,*) -- DOUBLE PRECISION array  
   ddf                                          
   On exit: the value of ----, for i,j=1,2,...,n.  
            ddy                                          

If the user does not wish to supply the Jacobian, the actual argument PEDERV must be the dummy routine D02EJY. (D02EJY is included in the NAG Foundation Library and so need not be supplied by the user. The name may be implementation dependent: see the User's Note for your implementation for details).

PEDERV must be declared as EXTERNAL in the (sub)program from which D02EJF is called. Parameters denoted as Input must not be changed by this procedure.

8: TOL -- DOUBLE PRECISION  
   On entry: TOL must be set to a positive tolerance for controlling the error in the integration. Hence TOL affects the determination of the position where g(x,y) = 0.0, if G is supplied.

D02EJF has been designed so that, for most problems, a reduction in TOL leads to an approximately proportional reduction in the error in the solution. However, the actual relation between TOL and the accuracy achieved cannot be guaranteed. The user is strongly recommended to call D02EJF with more than one value for TOL and to compare the results obtained to estimate their accuracy. In the absence of any prior knowledge, the user might compare the results obtained by calling D02EJF with TOL=10^-p and TOL=10^-p-1 if p correct decimal digits are required in the solution. Constraint: TOL > 0.0. On exit: normally unchanged. However if the range X to XEND is so short that a small change in TOL is unlikely to make any change in the computed solution, then, on return, TOL has its sign changed.

9: RELABS -- CHARACTER*1  
   On entry: the type of error control. At each step in the numerical solution an estimate of the local error, EST, is
made. For the current step to be accepted the following condition must be satisfied:

\[
\text{EST} = \frac{e}{\tau} + \frac{\tau}{|y| + \tau} \leq 1.0
\]

where \( \tau \) and \( \tau \) are defined by

\[
\text{RELABS} (\tau) = \begin{cases} 
'M' & \text{TOL} \\
'A' & 0.0 \\
'R' & \text{TOL} (\epsilon) \\
'D' & \text{TOL} (\epsilon)
\end{cases}
\]

where \( \epsilon \) is a small machine-dependent number and \( e \) is an estimate of the local error at \( y \), computed internally. If the appropriate condition is not satisfied, the step size is reduced and the solution is recomputed on the current step. If the user wishes to measure the error in the computed solution in terms of the number of correct decimal places, then RELABS should be set to 'A' on entry, whereas if the error requirement is in terms of the number of correct significant digits, then RELABS should be set to 'R'. If the user prefers a mixed error test, then RELABS should be set to 'M', otherwise if the user has no preference, RELABS should be set to the default 'D'. Note that in this case 'D' is taken to be 'R'. Constraint: RELABS = 'A', 'M', 'R' or 'D'.

10: OUTPUT -- SUBROUTINE, supplied by the user.

External Procedure

OUTPUT allows the user to have access to intermediate values of the computed solution at successive points specified by the user. These solution values may be returned to the user via the array RESULT if desired (this is a non-standard feature added for use with the AXIOM system). OUTPUT is initially called by D02EJF with XSOL = X (the initial value of x). The user must reset XSOL to the next point where OUTPUT is to be called, and so on at each call to OUTPUT. If, after a call to OUTPUT, the reset point XSOL is beyond XEND, D02EJF will integrate to XEND with no further calls to OUTPUT; if a call to OUTPUT is required at the point XSOL = XEND, then XSOL
must be given precisely the value XEND.

Its specification is:

```plaintext
SUBROUTINE OUTPUT(XSOL,Y,COUNT,M,N,RESULT)
DOUBLE PRECISION Y(N),RESULT(M,N),XSOL
INTEGER M,N,COUNT

1: XSOL -- DOUBLE PRECISION  Input/Output
On entry: the current value of the independent
variable x. On exit: the next value of x at which
OUTPUT is to be called.

2: Y(N) -- DOUBLE PRECISION array  Input
On entry: the computed solution at the point XSOL.

3: COUNT -- INTEGER  Input/Output
On entry: Zero if OUTPUT has not been called before, or
the previous value of COUNT.
On exit: A new value of COUNT: this can be used to keep
track of the number of times OUTPUT has been called.

4: M -- INTEGER  Input
On entry: The first dimension of RESULT.

5: N -- INTEGER  Input
On entry: The dimension of Y.

6: RESULT(M,N) -- DOUBLE PRECISION array  Input/Output
On entry: the previous contents of RESULT.
On exit: RESULT may be used to return the values of the
intermediate solutions to the user.

OUTPUT must be declared as EXTERNAL in the (sub)program
from which D02EJF is called. Parameters denoted as
Input must not be changed by this procedure.

11: G -- DOUBLE PRECISION FUNCTION, supplied by the user.

External Procedure
G must evaluate the function g(x,y) for specified values x,y
. It specifies the function g for which the first position x
where g(x,y) = 0 is to be found.

Its specification is:

```plaintext
DOUBLE PRECISION FUNCTION G(X, Y)
DOUBLE PRECISION X, Y(n)
where n is the actual value of N in the call of D02EJF.
```
On entry: the value of the independent variable $x$.

2: $Y(*)$ -- DOUBLE PRECISION array Input
On entry: the value of the variable $y_i$, for
$i = 1, 2, \ldots, n$.
If the user does not require the root finding option,
the actual argument $G$ must be the dummy routine D02EJW.
(D02EJW is included in the NAG Foundation Library and
so need not be supplied by the user).
$G$ must be declared as EXTERNAL in the (sub)program from
which D02EJF is called. Parameters denoted as Input
must not be changed by this procedure.

12: $W(IW)$ -- DOUBLE PRECISION array Workspace

13: $IW$ -- INTEGER Input
On entry: the dimension of the array $W$ as declared in the (sub)program
from which D02EJF is called.
Constraint: $IW \geq (12+N)\times N+50$.

14: RESULT(M,N) -- DOUBLE PRECISION array Output
On exit: the computed values of the solution at the points
given by OUTPUT.

15: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry TOL <= 0.0,
or $X = XEND$,
or $N <= 0$,
or RELABS /= 'M', 'A', 'R', 'D'.
or \( IW < (12+N) \times N + 50 \).

**IFAIL = 2**

With the given value of TOL, no further progress can be made across the integration range from the current point \( x = X \). (See Section 5 for a discussion of this error test.) The components \( Y(1), Y(2), \ldots, Y(n) \) contain the computed values of the solution at the current point \( x = X \). If the user has supplied \( G \), then no point at which \( g(x,y) \) changes sign has been located up to the point \( x = X \).

**IFAIL = 3**

TOL is too small for D02EJF to take an initial step. \( X \) and \( Y(1), Y(2), \ldots, Y(n) \) retain their initial values.

**IFAIL = 4**

XSO\(L \) lies behind \( X \) in the direction of integration, after the initial call to OUTPUT, if the OUTPUT option was selected.

**IFAIL = 5**

A value of XSO\(L \) returned by OUTPUT lies behind the last value of XSO\(L \) in the direction of integration, if the OUTPUT option was selected.

**IFAIL = 6**

At no point in the range \( X \) to XEND did the function \( g(x,y) \) change sign, if \( G \) was supplied. It is assumed that \( g(x,y) = 0 \) has no solution.

**IFAIL = 7**

A serious error has occurred in an internal call to C05AZF(*). Check all subroutine calls and array dimensions. Seek expert help.

**IFAIL = 8**

A serious error has occurred in an internal call to D02XKF(*). Check all subroutine calls and array dimensions. Seek expert help.

**IFAIL = 9**

A serious error has occurred in an internal call to D02NMF(*). Check all subroutine calls and array dimensions. Seek expert help.

7. Accuracy

The accuracy of the computation of the solution vector \( Y \) may be controlled by varying the local error tolerance TOL. In general, a decrease in local error tolerance should lead to an increase in
accuracy. Users are advised to choose RELABS = 'R' unless they have a good reason for a different choice. It is particularly appropriate if the solution decays.

If the problem is a root-finding one, then the accuracy of the root determined will depend strongly on \( \frac{\text{ddg}}{\text{ddx}} \) and \( \frac{\text{ddg}}{\text{ddy}} \), for \( i=1,2,\ldots,n \). Large values for these quantities may imply large errors in the root.

8. Further Comments

If more than one root is required, then to determine the second and later roots D02EJF may be called again starting a short distance past the previously determined roots. Alternatively the user may construct his own root finding code using D02QDF(*) (or the routines of the subchapter D02M-D02N), D02XKF(*) and C05AZF(*).

If it is easy to code, the user should supply the routine PEDERV. However, it is important to be aware that if PEDERV is coded incorrectly, a very inefficient integration may result and possibly even a failure to complete the integration (IFAIL = 2).

9. Example

We illustrate the solution of five different problems. In each case the differential system is the well-known stiff Robertson problem.

\[
\begin{align*}
a' &= -0.04a - 10 \times 10^{-4} b c \\
b' &= 0.04a - 10 \times 10^{-4} b c - 3 \times 10^{-2} b \\
c' &= 3 \times 10^{-2} b
\end{align*}
\]

with initial conditions \( a=1.0, b=c=0.0 \) at \( x=0.0 \). We solve each of the following problems with local error tolerances \( 1.0 \times 10^{-3} \) and \( 1.0 \times 10^{-4} \).

(i) To integrate to \( x=10.0 \) producing output at intervals of 2.0 until a point is encountered where \( a=0.9 \). The Jacobian is calculated numerically.

(ii) As (i) but with the Jacobian calculated analytically.

(iii) As (i) but with no intermediate output.
(iv) As (i) but with no termination on a root-finding condition.

(v) Integrating the equations as in (i) but with no intermediate output and no root-finding termination condition.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

1. Purpose

D02GAF solves the two-point boundary-value problem with assigned boundary values for a system of ordinary differential equations, using a deferred correction technique and a Newton iteration.

2. Specification

```fortran
SUBROUTINE D02GAF (U, V, N, A, B, TOL, FCN, MNP, X, Y, NP, W, LW, LW, LIW, IFAIL)
INTEGER N, MNP, NP, LW, IW(LIW), LIW, IFAIL
DOUBLE PRECISION U(N,2), V(N,2), A, B, TOL, X(MNP), Y(N,MNP), W(LW)
EXTERNAL FCN
```

3. Description

D02GAF solves a two-point boundary-value problem for a system of n differential equations in the interval [a,b]. The system is written in the form

\[ y'_i = f(x, y_{i1}, y_{i2}, \ldots, y_{in}), \quad i=1,2,\ldots,n \quad (1) \]

and the derivatives are evaluated by a subroutine FCN supplied by the user. Initially, n boundary values of the variables \( y_i \) must
be specified (assigned), some at a and some at b. The user also supplies estimates of the remaining n boundary values and all the boundary values are used in constructing an initial approximation to the solution. This approximate solution is corrected by a finite-difference technique with deferred correction allied with a Newton iteration to solve the finite-difference equations. The technique used is described fully in Pereyra [1]. The Newton \( \frac{df}{dx} \) iteration requires a Jacobian matrix \( \frac{ddf}{ddx} \) and this is calculated by numerical differentiation using an algorithm described in Curtis et al [2].

The user supplies an absolute error tolerance and may also supply an initial mesh for the construction of the finite-difference equations (alternatively a default mesh is used). The algorithm constructs a solution on a mesh defined by adding points to the initial mesh. This solution is chosen so that the error is everywhere less than the user’s tolerance and so that the error is approximately equidistributed on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If on the other hand the solution is required at several specific points then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

4. References


5. Parameters

1: \( U(N,2) \) -- DOUBLE PRECISION array
Input
On entry: \( U(i,1) \) must be set to the known (assigned) or estimated values of \( y \) at a and \( U(i,2) \) must be set to the \( i \)
known or estimated values of \( y \) at b, for \( i=1,2,\ldots,n \).
2: $V(N,2)$ -- DOUBLE PRECISION array  
   Input  
   On entry: $V(i,j)$ must be set to 0.0 if $U(i,j)$ is a known  
   (assigned) value and to 1.0 if $U(i,j)$ is an estimated value,  
   $i=1,2,...,n$; $j=1,2$. Constraint: precisely $N$ of the $V(i,j)$  
   must be set to 0.0, i.e., precisely $N$ of the $U(i,j)$ must be  
   known values, and these must not be all at $a$ or all at $b$.

3: N -- INTEGER  
   Input  
   On entry: the number of equations. Constraint: $N \geq 2$.

4: A -- DOUBLE PRECISION  
   Input  
   On entry: the left-hand boundary point, $a$.

5: B -- DOUBLE PRECISION  
   Input  
   On entry: the right-hand boundary point, $b$. Constraint: $B > A$.

6: TOL -- DOUBLE PRECISION  
   Input  
   On entry: a positive absolute error tolerance. If  
   $a=x_1<x_2<...<x_{NP}=b$  
   is the final mesh, $z(x_i)$ is the $j$th component of the  
   approximate solution at $x_i$, and $y(x_i)$ is the $j$th component  
   of the true solution of equation (1) (see Section 3) and the  
   boundary conditions, then, except in extreme cases, it is  
   expected that  
   $|z(x_i)-y(x_i)| \leq TOL$, $i=1,2,...,NP; j=1,2,...,n$  
   (2)  
   Constraint: $TOL > 0.0$.

7: FCN -- SUBROUTINE, supplied by the user.  
   External Procedure  
   FCN must evaluate the functions $f$ (i.e., the derivatives  
   $y'$) at the general point $x$.

   Its specification is:

   SUBROUTINE FCN (X, Y, F)
   DOUBLE PRECISION X, Y(n), F(n)
   where $n$ is the actual value of $N$ in the call of D02GAF.

1: X -- DOUBLE PRECISION  
   Input  
   On entry: the value of the argument $x$. 
2: Y(*) -- DOUBLE PRECISION array  
   Input
   On entry: the value of the argument y, for
   \( i = 1, 2, \ldots, n \).

3: F(*) -- DOUBLE PRECISION array  
   Output
   On exit: the values of f, for \( i = 1, 2, \ldots, n \).

FCN must be declared as EXTERNAL in the (sub)program
from which D02GAF is called. Parameters denoted as
Input must not be changed by this procedure.

8: MNP -- INTEGER  
   Input
   On entry: the maximum permitted number of mesh points.
   Constraint: MNP \geq 32.

9: X(MNP) -- DOUBLE PRECISION array  
   Input/Output
   On entry: if NP \geq 4 (see NP below), the first NP elements
   must define an initial mesh. Otherwise the elements of X
   need not be set. Constraint:
   \[ A = X(1) < X(2) < \ldots < X(NP) = B \] for NP=4
   (3)
   On exit: X(1), X(2), ..., X(NP) define the final mesh (with
   the returned value of NP) satisfying the relation (3).

10: Y(N,MNP) -- DOUBLE PRECISION array  
    Output
    On exit: the approximate solution \( z(x) \) satisfying (2), on
    \[ Y(j,i) = z(x), \quad i = 1, 2, \ldots, n; \quad j = 1, 2, \ldots, n, \]
    where NP is the number of points in the final mesh.

The remaining columns of Y are not used.

11: NP -- INTEGER  
    Input/Output
    On entry: determines whether a default or user-supplied
    mesh is used. If NP = 0, a default value of 4 for NP and a
    corresponding equispaced mesh \( X(1), X(2), \ldots, X(NP) \) are used.
    If NP \geq 4, then the user must define an initial mesh using
    the array X as described. Constraint: NP = 0 or 4 \leq NP \leq
    MNP. On exit: the number of points in the final (returned)
    mesh.

12: W(LW) -- DOUBLE PRECISION array  
    Workspace

13: LW -- INTEGER  
    Input
    On entry: the length of the array W as declared in the
    calling (sub)program. Constraint: LW \geq MNP*(3N + 6N+2) + 4N + 4N
14: IW(LIW) -- INTEGER array Workspace

15: LIW -- INTEGER Input
On entry: the length of the array IW as declared in the
2 calling (sub)program. Constraint: LIW>=MNP*(2N+1)+N +4N+2.

16: IFAIL -- INTEGER Input/Output
For this routine, the normal use of IFAIL is extended to
control the printing of error and warning messages as well
as specifying hard or soft failure (see the Essential
Introduction).

Before entry, IFAIL must be set to a value with the decimal
expansion cba, where each of the decimal digits c, b and a
must have a value of 0 or 1.

a=0 specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages
will be printed (see Section 6);
c=0 suppresses warning messages, otherwise warning
messages will be printed (see Section 6).
The recommended value for inexperienced users is 110 (i.e.,
hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL
contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
One or more of the parameters N, TOL, NP, MNP, LW or LIW has
been incorrectly set, or B <= A, or the condition (3) on X
is not satisfied, or the number of known boundary values
(specified by V) is not N.

IFAIL= 2
The Newton iteration has failed to converge. This could be
due to there being too few points in the initial mesh or to
the initial approximate solution being too inaccurate. If
this latter reason is suspected the user should use
subroutine D02RAF instead. If the warning 'Jacobian matrix
is singular' is printed this could be due to specifying zero
estimated boundary values and these should be varied. This
warning could also be printed in the unlikely event of the Jacobian matrix being calculated inaccurately. If the user cannot make changes to prevent the warning then subroutine D02RAF should be used.

IFAIL = 3
The Newton iteration has reached round-off level. It could be, however, that the answer returned is satisfactory. This error might occur if too much accuracy is requested.

IFAIL = 4
A finer mesh is required for the accuracy requested; that is MNP is not large enough.

IFAIL = 5
A serious error has occurred in a call to D02GAF. Check all array subscripts and subroutine parameter lists in calls to D02GAF. Seek expert help.

7. Accuracy
The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (2) except in extreme circumstances. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments
The time taken by the routine depends on the difficulty of the problem, the number of mesh points used (and the number of different meshes used), the number of Newton iterations and the number of deferred corrections.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation. The user may select the channel numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) - see Section 9 for an example. Otherwise the default channel numbers will be used, as specified in the implementation document.

A common cause of convergence problems in the Newton iteration is the user specifying too few points in the initial mesh. Although the routine adds points to the mesh to improve accuracy it is unable to do so until the solution on the initial mesh has been calculated in the Newton iteration.

If the user specifies zero known and estimated boundary values, the routine constructs a zero initial approximation and in many
cases the Jacobian is singular when evaluated for this approximation, leading to the breakdown of the Newton iteration.

The user may be unable to provide a sufficiently good choice of initial mesh and estimated boundary values, and hence the Newton iteration may never converge. In this case the continuation facility provided in D02RAF is recommended.

In the case where the user wishes to solve a sequence of similar problems, the final mesh from solving one case is strongly recommended as the initial mesh for the next.

9. Example

We solve the differential equation

\[ y''' = -yy'' - \beta (1 - y') \]

with boundary conditions

\[ y(0) = y'(0) = 0, \]
\[ y'(10) = 1 \]

for \( \beta = 0.0 \) and \( \beta = 0.2 \) to an accuracy specified by TOL=1.0E-3. We solve first the simpler problem with \( \beta = 0.0 \) using an equispaced mesh of 26 points and then we solve the problem with \( \beta = 0.2 \) using the final mesh from the first problem.

Note the call to X04ABF prior to the call to D02GAF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D02GBF solves a general linear two-point boundary value problem for a system of ordinary differential equations using a deferred correction technique.

2. Specification

```fortran
SUBROUTINE D02GBF (A, B, N, TOL, FCNF, FCNG, C, D, GAM,
  1 MNP, X, Y, NP, W, LW, IW, LIW, IFAIL)
    INTEGER N, MNP, NP, LW, IW(LIW), LIW, IFAIL
    DOUBLE PRECISION A, B, TOL, C(N,N), D(N,N), GAM(N), X(MNP),
  1 Y(N,MNP), W(LW)
    EXTERNAL FCNF, FCNG
```

3. Description

D02GBF solves the linear two-point boundary value problem for a system of n ordinary differential equations in the interval [a,b]. The system is written in the form

\[ y' = F(x)y + g(x) \]  \hspace{1cm} (1)

and the boundary conditions are written in the form

\[ Cy(a) + Dy(b) = \gamma \]  \hspace{1cm} (2)

Here \( F(x) \), C and D are n by n matrices, and \( g(x) \) and \( \gamma \) are n-component vectors. The approximate solution to (1) and (2) is found using a finite-difference method with deferred correction. The algorithm is a specialisation of that used in subroutine D02RAF which solves a nonlinear version of (1) and (2). The nonlinear version of the algorithm is described fully in Pereyra [1].

The user supplies an absolute error tolerance and may also supply an initial mesh for the construction of the finite-difference equations (alternatively a default mesh is used). The algorithm constructs a solution on a mesh defined by adding points to the initial mesh. This solution is chosen so that the error is everywhere less than the user's tolerance and so that the error is approximately equidistributed on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points, then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

4. References

5. Parameters

1: A -- DOUBLE PRECISION Input
On entry: the left-hand boundary point, a.

2: B -- DOUBLE PRECISION Input
On entry: the right-hand boundary point, b. Constraint: B > A.

3: N -- INTEGER Input
On entry: the number of equations; that is n is the order of system (1). Constraint: N >= 2.

4: TOL -- DOUBLE PRECISION Input
On entry: a positive absolute error tolerance. If
\[ a=x_1<x_2<...<x_{NP}=b \]
is the final mesh, z(x) is the approximate solution from D02GBF and y(x) is the true solution of equations (1) and (2) then, except in extreme cases, it is expected that
\[ ||z-y||\leq TOL \] (3)
where
\[ ||u||=\max_{1\leq i\leq N} \max_{1\leq j\leq NP} |u(x)|. \]
Constraint: TOL > 0.0.

5: FCNF -- SUBROUTINE, supplied by the user. External Procedure
FCNF must evaluate the matrix F(x) in (1) at a general point x.

Its specification is:

\begin{verbatim}
SUBROUTINE FCN (X, F)
  DOUBLE PRECISION X, F(n,n)
\end{verbatim}

where n is the actual value of N in the call of D02GBF.

1: X -- DOUBLE PRECISION Input
On entry: the value of the independent variable x.

2: F(n,n) -- DOUBLE PRECISION array Output
On exit: the \((i,j)\)th element of the matrix \(F(x)\), for
i,j=1,2,...,n. (See Section 9 for an example.)

FCN must be declared as EXTERNAL in the (sub)program from which D02GBF is called. Parameters denoted as Input must not be changed by this procedure.

6: FCNG -- SUBROUTINE, supplied by the user.

External Procedure

FCNG must evaluate the vector g(x) in (1) at a general point x.

Its specification is:

\[ \text{SUBROUTINE FCNG}(X, G) \]
\[ \text{DOUBLE PRECISION X, G(n)} \]
where n is the actual value of N in the call of D02GBF.

1: X -- DOUBLE PRECISION  
Input  
On entry: the value of the independent variable x.

2: G(*) -- DOUBLE PRECISION array  
Output  
On exit: the ith element of the vector g(x), for i=1,2,...,n. (See Section Section 9 for an example.)

FCNG must be declared as EXTERNAL in the (sub)program from which D02GBF is called. Parameters denoted as Input must not be changed by this procedure.

7: C(N,N) -- DOUBLE PRECISION array  
Input/Output  
On entry: the arrays C and D must be set to the matrices C and D in (2). GAM must be set to the vector (gamma) in (2).

On exit: the rows of C and D and the components of GAM are re-ordered so that the boundary conditions are in the order:

(i) conditions on y(a) only;

(ii) condition involving y(a) and y(b); and

(iii) conditions on y(b) only.

The routine will be slightly more efficient if the arrays C, D and GAM are ordered in this way before entry, and in this event they will be unchanged on exit.

Note that the problems (1) and (2) must be of boundary value type, that is neither C nor D may be identically zero. Note also that the rank of the matrix \([C,D]\) must be n for the problem to be properly posed. Any violation of these conditions will lead to an error exit.
10: MNP -- INTEGER
   Input
   On entry: the maximum permitted number of mesh points.
   Constraint: MNP >= 32.

11: X(MNP) -- DOUBLE PRECISION array
    Input/Output
    On entry: if NP >= 4 (see NP below), the first NP elements
    must define an initial mesh. Otherwise the elements of x
    need not be set. Constraint:
    \[ A = X(1) < X(2) < \ldots < X(NP) = B, \text{ for } NP \geq 4. \] (4)
    On exit: \( X(1), X(2), \ldots, X(NP) \) define the final mesh (with the
    returned value of NP) satisfying the relation (4).

12: Y(N,MNP) -- DOUBLE PRECISION array
    Output
    On exit: the approximate solution \( z(x) \) satisfying (3), on
    the final mesh, that is
    \[ Y(j,i) = z(x_j^i), \quad i = 1, 2, \ldots, NP; j = 1, 2, \ldots, n \] 
    where NP is the number of points in the final mesh.

    The remaining columns of Y are not used.

13: NP -- INTEGER
    Input/Output
    On entry: determines whether a default mesh or user-supplied
    mesh is used. If NP = 0, a default value of 4 for NP and a
    corresponding equispaced mesh \( X(1), X(2), \ldots, X(NP) \) are used.
    If NP >= 4, then the user must define an initial mesh X as
    in (4) above. On exit: the number of points in the final
    (returned) mesh.

14: W(LW) -- DOUBLE PRECISION array
    Workspace

15: LW -- INTEGER
    Input
    On entry: the length of the array W, Constraint:
    \[ 2 \leq LW \geq MNP \times (3N + 5N + 2) + 3N + 5N. \]

16: IW(LIW) -- INTEGER array
    Workspace

17: LIW -- INTEGER
    Input
    On entry: the length of the array IW, Constraint:
    \[ LIW \geq MNP \times (2N + 1) + N. \]

18: IFAIL -- INTEGER
    Input/Output
    For this routine, the normal use of IFAIL is extended to
    control the printing of error and warning messages as well
    as specifying hard or soft failure (see the Essential
    Introduction).

    Before entry, IFAIL must be set to a value with the decimal
expansion cba, where each of the decimal digits c, b and a
must have a value of 0 or 1.
a=0 specifies hard failure, otherwise soft failure;
b=0 suppresses error messages, otherwise error messages
will be printed (see Section 6);
c=0 suppresses warning messages, otherwise warning
messages will be printed (see Section 6).
The recommended value for inexperienced users is 110 (i.e.,
hard failure with all messages printed).

Unless the routine detects an error (see Section 6), IFAIL
contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the
current error message unit (as defined by X04AAF), unless
suppressed by the value of IFAIL on entry.

IFAIL= 1
One or more of the parameters N, TOL, NP, MNP, LW or LIW is
incorrectly set, B <= A or the condition (4) on X is not
satisfied.

IFAIL= 2
There are three possible reasons for this error exit to be
taken:
(i) one of the matrices C or D is identically zero (that
is the problem is of initial value and not boundary
value type). In this case, IW(1) = 0 on exit;

(ii) a row of C and the corresponding row of D are
identically zero (that is the boundary conditions are
rank deficient). In this case, on exit IW(1) contains
the index of the first such row encountered; and

(iii more than n of the columns of the n by 2n matrix \([C,D]
) are identically zero (that is the boundary
conditions are rank deficient). In this case, on exit
IW(1) contains minus the number of non-identically
zero columns.

IFAIL= 3
The routine has failed to find a solution to the specified
accuracy. There are a variety of possible reasons including:
(i) the boundary conditions are rank deficient, which may
be indicated by the message that the Jacobian is singular. However this is an unlikely explanation for the error exit as all rank deficient boundary conditions should lead instead to error exits with either IFAIL = 2 or IFAIL = 5; see also (iv) below;

(ii) not enough mesh points are permitted in order to attain the required accuracy. This is indicated by NP = MNP on return from a call to D02GBF. This difficulty may be aggravated by a poor initial choice of mesh points;

(iii) the accuracy requested cannot be attained on the computer being used; and

(iv) an unlikely combination of values of F(x) has led to a singular Jacobian. The error should not persist if more mesh points are allowed.

IFAIL= 4
A serious error has occurred in a call to D02GBF. Check all array subscripts and subroutine parameter lists in calls to D02GBF. Seek expert help.

IFAIL= 5
There are two possible reasons for this error exit which occurs when checking the rank of the boundary conditions by reduction to a row echelon form:

(i) at least one row of the n by 2n matrix \([C,D]\) is a linear combination of the other rows and hence the boundary conditions are rank deficient. The index of the first such row encountered is given by IW(1) on exit; and

(ii) as (i) but the rank deficiency implied by this error exit has only been determined up to a numerical tolerance. Minus the index of the first such row encountered is given by IW(1) on exit.

In case (ii) above there is some doubt as to the rank deficiency of the boundary conditions. However even if the boundary conditions are not rank deficient they are not posed in a suitable form for use with this routine.

For example, if

\[
\begin{pmatrix}
\gamma \\
1 & 0 \\
0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix},
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
\gamma = \begin{pmatrix}
1 & (\text{epsilon})
\end{pmatrix}
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 \\
0 & 1
\end{pmatrix}
\]

and (epsilon) is small enough, this error exit is likely to be taken. A better form for the boundary conditions in this
case would be

$$\begin{pmatrix}
\gamma \\
1 \\
-1
\end{pmatrix}
$$

$$C = \begin{pmatrix}
0 & 1 \\
0 & 0
\end{pmatrix},
D = \begin{pmatrix}
0 & 0 \\
-\gamma & -\gamma
\end{pmatrix},
(\gamma) = \begin{pmatrix}
\epsilon & (\gamma) - (\gamma) \\
2 & 1
\end{pmatrix}$$

7. Accuracy

The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (3) except in extreme circumstances. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments

The time taken by the routine depends on the difficulty of the problem, the number of mesh points (and meshes) used and the number of deferred corrections.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation. The user may select the channel numbers on which this output is to appear by calls of X04AAF (for error messages) or X04ABF (for monitoring information) - see Section 9 for an example. Otherwise the default channel numbers will be used, as specified in the implementation document.

In the case where the user wishes to solve a sequence of similar problems, the use of the final mesh from one case is strongly recommended as the initial mesh for the next.

9. Example

We solve the problem (written as a first order system)

$$(\epsilon)y'' + y' = 0$$

with boundary conditions

$$y(0) = 0, \quad y(1) = 1$$

for the cases $(\epsilon) = 10$ and $(\epsilon) = 10$ using the default initial mesh in the first case, and the final mesh of the first case as initial mesh for the second (more difficult) case. We give the solution and the error at each mesh point to illustrate the accuracy of the method given the accuracy request TOL=1.0E-3.
Note the call to X04ABF prior to the call to D02GBF.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

D02 -- Ordinary Differential Equations
D02KEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D02KEF finds a specified eigenvalue of a regular singular second-order Sturm-Liouville system on a finite or infinite range, using a Pruefer transformation and a shooting method. It also reports values of the eigenfunction and its derivatives. Provision is made for discontinuities in the coefficient functions or their derivatives.

2. Specification

```fortran
SUBROUTINE D02KEF (XPOINT, M, MATCH, COEFFN, BDYVAL, K, TOL, ELAM, DELAM, HMAX, MAXIT, MAXFUN, MONIT, REPORT, IFAIL)
INTEGER M, MATCH, K, MAXIT, MAXFUN, IFAIL
DOUBLE PRECISION XPOINT(M), TOL, ELAM, DELAM, HMAX(2,M)
EXTERNAL COEFFN, BDYVAL, MONIT, REPORT
```

3. Description

D02KEF has essentially the same purpose as D02KDF(*) with minor modifications to enable values of the eigenfunction to be obtained after convergence to the eigenvalue has been achieved.

It first finds a specified eigenvalue ($\lambda$) of a Sturm-Liouville system defined by a self-adjoint differential equation of the second-order

$$(p(x)y')' + q(x;\lambda)y = 0, \quad a < x < b$$

together with the appropriate boundary conditions at the two (finite or infinite) end-points $a$ and $b$. The functions $p$ and $q$, which are real-valued, must be defined by a subroutine COEFFN.
The boundary conditions must be defined by a subroutine BDYVAL, and, in the case of a singularity at a or b, take the form of an asymptotic formula for the solution near the relevant end-point.

When the final estimate \((\lambda)\) of the eigenvalue has been found, the routine integrates the differential equation once more with that value of \((\lambda)\), and with initial conditions chosen so that the integral

\[
S = \frac{1}{2} \int_{a}^{b} \frac{y(x)}{x; \lambda} dx
\]

is approximately one. When \(q(x; \lambda)\) is of the form \((\lambda)w(x)+q(x)\), which is the most common case, \(S\) represents the square of the norm of \(y\) induced by the inner product

\[
\langle f, g \rangle = \int_{a}^{b} f(x)g(x)w(x)dx,
\]

with respect to which the eigenfunctions are mutually orthogonal. This normalisation of \(y\) is only approximate, but experience shows that \(S\) generally differs from unity by only one or two per cent.

During this final integration the REPORT routine supplied by the user is called at each integration mesh point \(x\). Sufficient information is returned to permit the user to compute \(y(x)\) and \(y'(x)\) for printing or plotting. For reasons described in Section 8.2, D02KEF passes across to REPORT, not \(y\) and \(y'\), but the Pru efer variables \((\beta)\), \((\phi)\) and \((\rho)\) on which the numerical method is based. Their relationship to \(y\) and \(y'\) is given by the equations

\[
p(x)y' = \sqrt{(\beta)}e^{-\frac{1}{2} \phi}cos\left(\frac{1}{2} \rho\right);
\]

\[
y = \sqrt{(\beta)}e^{-\frac{1}{2} \phi}sin\left(\frac{1}{2} \rho\right).
\]

For the theoretical basis of the numerical method to be valid,
the following conditions should hold on the coefficient functions:

(a) \( p(x) \) must be non-zero and of one sign throughout the interval \((a,b)\); and,

\[
\frac{d}{dx} \frac{d}{d(\lambda)}
\]

(b) \( \frac{d}{d(\lambda)} \) must be of one sign throughout \((a,b)\) for all relevant values of \((\lambda)\), and must not be identically zero as \(x\) varies, for any \((\lambda)\).

Points of discontinuity in the functions \(p\) and \(q\) or their derivatives are allowed, and should be included as 'break-points' in the array XPOINT.

A good account of the theory of Sturm-Liouville systems, with some description of Pruefer transformations, is given in Birkhoff and Rota [4], Chapter X. An introduction for the user of Pruefer transformations for the numerical solution of eigenvalue problems arising from physics and chemistry is Bailey [2].

The scaled Pruefer method is fairly recent, and is described in a short note by Pryce [6] and in some detail in the technical report [5].

4. References


5. Parameters
CHAPTER 15.

1: XPOINT(M) -- DOUBLE PRECISION array
   On entry: the points where the boundary conditions computed
   by BDYVAL are to be imposed, and also any break-points,
   i.e., XPOINT(1) to XPOINT(m) must contain values x ,...,x
   1 m
   such that
   \[ x \leq x < x < ... < x \leq x \]
   1 2 3 m-1 m
   with the following meanings:
   (a) x and x are the left and right end-points, a and b,
       1 m
       of the domain of definition of the Sturm-Liouville
       system if these are finite. If either a or b is
       infinite, the corresponding value x or x may be a
       1 m
       more-or-less arbitrarily 'large' number of appropriate
       sign.
   (b) x and x are the Boundary Matching Points (BMP's),
       2 m-1
       that is the points at which the left and right
       boundary conditions computed in BDYVAL are imposed.
       If the left-hand end-point is a regular point then the
       user should set x = x (=a), while if it is a singular
       2 1
       point the user must set x > x. Similarly x = x (=b)
       2 1 m-1 m
       if the right-hand end-point is regular, and x < x if
       m-1 m
       it is singular.
   (c) The remaining m-4 points x ,...,x , if any, define
       3 m-2
       'break-points' which divide the interval [x ,x ]
       2 m-1
       into m-3 sub-intervals
       \[ i = [ x , x ], i = [ x , x ] \]
       1 2 3 m-3 m-2 m-1
       Numerical integration of the differential equation is
       stopped and restarted at each break-point. In simple
       cases no break-points are needed. However if p(x) or
       q(x;\lambda) are given by different formulae in
       different parts of the range, then integration is more
       efficient if the range is broken up by break-points in
       the appropriate way. Similarly points where any jumps
       occur in p(x) or q(x;\lambda), or in their
       derivatives up to the fifth order, should appear as
       break-points.
Constraint: $X(1) \leq X(2) < \ldots < X(M-1) \leq X(M)$.

2: \textbf{M} -- INTEGER \hspace{1cm} \text{Input}
On entry: the number of points in the array \text{XPOINT}.
Constraint: $M \geq 4$.

3: \textbf{MATCH} -- INTEGER \hspace{1cm} \text{Input/Output}
On entry: MATCH must be set to the index of the 'break-point'
to be used as the matching point (see Section 8.3). If
MATCH is set to a value outside the range $[2,m-1]$ then a
default value is taken, corresponding to the break-point
nearest the centre of the interval $[\text{XPOINT}(2),\text{XPOINT}(m-1)]$.
On exit: the index of the break-point actually used as the
matching point.

4: \textbf{COEFFN} -- SUBROUTINE, supplied by the user.
\hspace{1cm} \text{External Procedure}
\text{COEFFN} must compute the values of the coefficient functions
$p(x)$ and $q(x;(\lambda))$ for given values of $x$ and $(\lambda)$.
Section 3 states conditions which $p$ and $q$ must satisfy.

Its specification is:

\begin{verbatim}
SUBROUTINE COEFFN (P, Q, DQDL, X, ELAM, JINT)
DOUBLE PRECISION P, Q, DQDL, X, ELAM
INTEGER JINT

1: P -- DOUBLE PRECISION \hspace{1cm} \text{Output}
On exit: the value of $p(x)$ for the current value of $x$.

2: Q -- DOUBLE PRECISION \hspace{1cm} \text{Output}
On exit: the value of $q(x;(\lambda))$ for the current
value of $x$ and the current trial value of $(\lambda)$.

3: DQDL -- DOUBLE PRECISION \hspace{1cm} \text{Output}
On exit: the value of $\frac{d}{dx}q(x;(\lambda))$ for the
current value of $x$ and the current trial value of
$(\lambda)$. However DQDL is only used in error estimation
and an approximation (say to within 20\%) will suffice.

4: X -- DOUBLE PRECISION \hspace{1cm} \text{Input}
On entry: the current value of $x$.

5: ELAM -- DOUBLE PRECISION \hspace{1cm} \text{Input}
On entry: the current trial value of the eigenvalue
parameter $(\lambda)$.

6: JINT -- INTEGER \hspace{1cm} \text{Input}
\end{verbatim}
On entry: the index \( j \) of the sub-interval \( i \) (see specification of \( \text{XPOINT} \)) in which \( x \) lies.

See Section 8.4 and Section 9 for examples.

\( \text{COEFFN} \) must be declared as \texttt{EXTERNAL} in the (sub)program from which \texttt{DO2KEF} is called. Parameters denoted as Input must not be changed by this procedure.

\textbf{5: BDYVAL -- SUBROUTINE, supplied by the user.}

External Procedure

\texttt{BDYVAL} must define the boundary conditions. For each end-point, \texttt{BDYVAL} must return (in \texttt{YL} or \texttt{YR}) values of \( y(x) \) and \( p(x)y'(x) \) which are consistent with the boundary conditions at the end-points; only the ratio of the values matters. Here \( x \) is a given point (\( \text{XL} \) or \( \text{XR} \)) equal to, or close to, the end-point.

For a regular end-point (\( a \), say), \( x=a \); and a boundary condition of the form
\[
c_1 y(a)+c_2 y'(a)=0
\]
can be handled by returning constant values in \texttt{YL}, e.g.
\[
\text{YL}(1)=c_1 \quad \text{and} \quad \text{YL}(2)=-c_2 p(a).
\]

For a singular end-point however, \( \text{YL}(1) \) and \( \text{YL}(2) \) will in general be functions of \( \text{XL} \) and \( \text{ELAM} \), and \( \text{YR}(1) \) and \( \text{YR}(2) \) functions of \( \text{XR} \) and \( \text{ELAM} \), usually derived analytically from a power-series or asymptotic expansion. Examples are given in Section 8.5 Section 9.

Its specification is:

\[
\text{SUBROUTINE BDYVAL (XL, XR, ELAM, YL, YR))}
\]

\textbf{DOUBLE PRECISION XL, XR, ELAM, YL(3), YR(3)}

\textbf{1: XL -- DOUBLE PRECISION Input}
On entry: if \( a \) is a regular end-point of the system (so that \( a=x=x_1 \)), then \( \text{XL} \) contains \( a \). If \( a \) is a singular point (so that \( a<=x<x_1 \)), then \( \text{XL} \) contains a point \( x_1 <x<=x_2 \).

\textbf{2: XR -- DOUBLE PRECISION Input}
On entry: if \( b \) is a regular end-point of the system (so that \( x_2=x=b \)), then \( \text{XR} \) contains \( b \). If \( b \) is a singular point (so that \( x_1 <x=b \)).
point (so that $x < x \leq b$), then $X_R$ contains a point $x$
$m-1$ $m$
such that $x \leq x < x$.
$m-1$ $m$

3: ELAM -- DOUBLE PRECISION
   Input
   On entry: the current trial value of $(\lambda)$.

4: YL(3) -- DOUBLE PRECISION array
   Output
   On exit: YL(1) and YL(2) should contain values of $y(x)$
   and $p(x)y'(x)$ respectively (not both zero) which are
   consistent with the boundary condition at the left-hand
   end-point, given by $x = X_L$. YL(3) should not be set.

5: YR(3) -- DOUBLE PRECISION array
   Output
   On exit: YR(1) and YR(2) should contain values of $y(x)$
   and $p(x)y'(x)$ respectively (not both zero) which are
   consistent with the boundary condition at the right-
   hand end-point, given by $x = X_R$. YR(3) should not be
   set.

BDYVAL must be declared as EXTERNAL in the (sub)program
from which D02KEF is called. Parameters denoted as
Input must not be changed by this procedure.

6: K -- INTEGER
   Input
   On entry: the index $k$ of the required eigenvalue when the
   eigenvalues are ordered
   $(\lambda_0) < (\lambda_1) < (\lambda_2) < \ldots < (\lambda_k) < \ldots$
   Constraint: $K \geq 0$.

7: TOL -- DOUBLE PRECISION
   Input
   On entry: the tolerance parameter which determines the
   accuracy of the computed eigenvalue. The error estimate held
   in DELAM on exit satisfies the mixed absolute/relative error
   test
   $\text{DELAM} \leq TOL \times \max(1.0, |ELAM|)$
   (*)
   where ELAM is the final estimate of the eigenvalue. DELAM is
   usually somewhat smaller than the right-hand side of (*) but
   not several orders of magnitude smaller. Constraint: $TOL > 0.0$.

8: ELAM -- DOUBLE PRECISION
   Input/Output
   On entry: an initial estimate of the eigenvalue $(\lambda)$.
   On exit: the final computed estimate, whether or not an
   error occurred.

9: DELAM -- DOUBLE PRECISION
   Input/Output
On entry: an indication of the scale of the problem in the (\lambda)-direction. DELAM holds the initial 'search step' (positive or negative). Its value is not critical but the first two trial evaluations are made at ELAM and ELAM + DELAM, so the routine will work most efficiently if the eigenvalue lies between these values. A reasonable choice (if a closer bound is not known) is half the distance between adjacent eigenvalues in the neighbourhood of the one sought. In practice, there will often be a problem, similar to the one in hand but with known eigenvalues, which will help one to choose initial values for ELAM and DELAM.

If DELAM = 0.0 on entry, it is given the default value of 0.25*max(1.0,|ELAM|). On exit: with IFAIL = 0, DELAM holds an estimate of the absolute error in the computed eigenvalue, that is |(\lambda)-ELAM|\approx DELAM. (In Section 8.2 we discuss the assumptions under which this is true.) The true error is rarely more than twice, or less than a tenth, of the estimated error.

With IFAIL /= 0, DELAM may hold an estimate of the error, or its initial value, depending on the value of IFAIL. See Section 6 for further details.

10: HMAX(2,M) -- DOUBLE PRECISION array Input/Output
On entry: HMAX(1,j) a maximum step size to be used by the differential equation code in the jth sub-interval i (as described in the specification of parameter XPOINT), for j=1,2,...,m-3. If it is zero the routine generates a maximum step size internally.

It is recommended that HMAX(1,j) be set to zero unless the coefficient functions p and q have features (such as a narrow peak) within the jth sub-interval that could be 'missed' if a long step were taken. In such a case HMAX(1,j) should be set to about half the distance over which the feature should be observed. Too small a value will increase the computing time for the routine. See Section 8 for further suggestions.

The rest of the array is used as workspace. On exit: HMAX(1,m-1) and HMAX(1,m) contain the sensitivity coefficients (\sigma_l,\sigma_r), described in Section 8.6. Other entries contain diagnostic output in case of an error (see Section 6).

11: MAXIT -- INTEGER Input/Output
On entry: a bound on \( n \), the number of root-finding \( r \) iterations allowed, that is the number of trial values of \( \lambda \) that are used. If \( \text{MAXIT} \leq 0 \), no such bound is assumed. (See also under \( \text{MAXFUN} \).) Suggested value: \( \text{MAXIT} = 0 \). On exit: \( \text{MAXIT} \) will have been decreased by the number of iterations actually performed, whether or not it was positive on entry.

12: \( \text{MAXFUN} \) -- INTEGER  

\textbf{Input}  

On entry: a bound on \( n \), the number of calls to \( \text{COEFFN} \) made \( f \) in any one root-finding iteration. If \( \text{MAXFUN} \leq 0 \), no such bound is assumed. Suggested value: \( \text{MAXFUN} = 0 \).

\( \text{MAXFUN} \) and \( \text{MAXIT} \) may be used to limit the computational cost of a call to \( \text{D02KEF} \), which is roughly proportional to \( n \ast n \). \( r \ f \)

13: \( \text{MONIT} \) -- SUBROUTINE, supplied by the user.  

\textbf{External Procedure}  

\( \text{MONIT} \) is called by \( \text{D02KEF} \) at the end of each root-finding iteration and allows the user to monitor the course of the computation by printing out the parameters (see Section 8 for an example). If no monitoring is required, the dummy subroutine \( \text{D02KAY} \) may be used. (\( \text{D02KAY} \) is included in the NAG Foundation Library).

Its specification is:

\begin{verbatim}
SUBROUTINE MONIT (MAXIT, IFLAG, ELAM, FINFO)
   INTEGER MAXIT, IFLAG
   DOUBLE PRECISION ELAM, FINFO(15)
\end{verbatim}

1: \( \text{MAXIT} \) -- INTEGER  

\textbf{Input}  

On entry: the current value of the parameter \( \text{MAXIT} \) of \( \text{D02KEF} \); this is decreased by one at each iteration.

2: \( \text{IFLAG} \) -- INTEGER  

\textbf{Input}  

On entry: \( \text{IFLAG} \) describes what phase the computation is in, as follows:

\begin{itemize}
  \item \( \text{IFLAG} < 0 \) an error occurred in the computation of the 'miss-distance' at this iteration;
  \item \( \text{IFLAG} = 1 \) an error exit from \( \text{D02KEF} \) with \( \text{IFAIL} = -\text{IFLAG} \) will follow.
\end{itemize}
the routine is trying to bracket the eigenvalue \( \lambda \).

\[
\text{IFLAG} = 2
\]
the routine is converging to the eigenvalue \( \lambda \) (having already bracketed it).

3: \( \text{ELAM} \) -- DOUBLE PRECISION
Input
On entry: the current trial value of \( \lambda \).

4: \( \text{FINFO}(15) \) -- DOUBLE PRECISION array
Input
On entry: information about the behaviour of the shooting method, and diagnostic information in the case of errors. It should not normally be printed in full if no error has occurred (that is, if IFLAG > 0), though the first few components may be of interest to the user. In case of an error (IFLAG < 0) all the components of FINFO should be printed. The contents of FINFO are as follows:

\( \text{FINFO}(1) \): the current value of the 'miss-distance' or 'residual' function \( f(\lambda) \) on which the shooting method is based. \( \text{FINFO}(1) \) is set to zero if IFLAG < 0.

\( \text{FINFO}(2) \): an estimate of the quantity \( d\lambda \) defined as follows. Consider the perturbation in the miss-distance \( f(\lambda) \) that would result if the local error, in the solution of the differential equation, were always positive and equal to its maximum permitted value. Then \( d\lambda \) is the perturbation in \( \lambda \) that would have the same effect on \( f(\lambda) \). Thus, at the zero of \( f(\lambda) \), \( |d\lambda| \) is an approximate bound on the perturbation of the zero (that is the eigenvalue) caused by errors in numerical solution. If \( d\lambda \) is very large then it is possible that there has been a programming error in COEFFN such that \( q \) is independent of \( \lambda \). If this is the case, an error exit with IFAIL = 5 should follow. \( \text{FINFO}(2) \) is set to zero if IFLAG < 0.

\( \text{FINFO}(3) \): the number of internal iterations, using the same value of \( \lambda \) and tighter accuracy tolerances, needed to bring the accuracy (that is the value of \( d\lambda \)) to an acceptable value. Its value should normally be 1.0, and should almost never exceed 2.0.

\( \text{FINFO}(4) \): the number of calls to COEFFN at this iteration.
FINFO(5): the number of successful steps taken by the internal differential equation solver at this iteration. A step is successful if it is used to advance the integration (cf. COUT(8) in specification of D02PAF(*)).

FINFO(6): the number of unsuccessful steps used by the internal integrator at this iteration (cf. COUT(9) in specification of D02PAF(*)).

FINFO(7): the number of successful steps at the maximum step size taken by the internal integrator at this iteration (cf. COUT(3) in specification of D02PAF(*)).

FINFO(8): is not used.

FINFO(9) to FINFO(15): set to zero, unless IFLAG < 0 in which case they hold the following values describing the point of failure:

FINFO(9): contains the index of the sub-interval where failure occurred, in the range 1 to m-3. In case of an error in BDYVAL, it is set to 0 or m-2 depending on whether the left or right boundary condition caused the error.

FINFO(10): the value of the independent variable x, the point at which error occurred. In case of an error in BDYVAL, it is set to the value of XL or XR as appropriate (see the specification of BDYVAL).

FINFO(11), FINFO(12), FINFO(13): the current values of the Pruefer dependent variables (beta), (phi) and (rho) respectively. These are set to zero in case of an error in BDYVAL.

FINFO(14): the local-error tolerance being used by the internal integrator at the point of failure. This is set to zero in case of an error in BDYVAL.

FINFO(15): the last integration mesh point. This is set to zero in case of an error in BDYVAL.

MONIT must be declared as EXTERNAL in the (sub)program from which D02KEF is called. Parameters denoted as Input must not be changed by this procedure.

14: REPORT -- SUBROUTINE, supplied by the user.

External Procedure

This routine provides the means by which the user may
compute the eigenfunction \( y(x) \) and its derivative at each integration mesh point \( x \). (See Section 8 for an example).

Its specification is:

\[
\text{SUBROUTINE REPORT (X, V, JINT)}
\]
\[
\text{INTEGER } \quad \text{JINT}
\]
\[
\text{DOUBLE PRECISION } \quad \text{X, V(3)}
\]

1: \( X \) -- DOUBLE PRECISION Input
On entry: the current value of the independent variable \( x \). See Section 8.3 for the order in which values of \( x \) are supplied.

2: \( V(3) \) -- DOUBLE PRECISION array Input
On entry: \( V(1), V(2), V(3) \) hold the current values of the Pruefer variables \( (\beta), (\phi), (\rho) \) respectively.

3: \( \text{JINT} \) -- INTEGER Input
On entry: \( \text{JINT} \) indicates the sub-interval between break-points in which \( X \) lies exactly as for the routine COEFFN, except that at the extreme left end-point (when \( x = X\text{POINT}(2) \)) \( \text{JINT} \) is set to 0 and at the extreme right end-point (when \( x=x\text{POINT}(m-1) \)) \( \text{JINT} \) is set to \( m-2 \).

REPORT must be declared as EXTERNAL in the (sub)program from which D02KEF is called. Parameters denoted as Input must not be changed by this procedure.

15: \( \text{IFAIL} \) -- INTEGER Input/Output
On entry: \( \text{IFAIL} \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: \( \text{IFAIL} = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

\( \text{IFAIL} = 1 \)
A parameter error. All parameters (except \( \text{IFAIL} \)) are left unchanged. The reason for the error is shown by the value of HMAX(2,1) as follows:

\[
\text{HMAX}(2,1) = 1: \quad M < 4;
\]

\[
\text{HMAX}(2,1) = 2: \quad K < 0;
\]
HMAX(2,1) = 3: TOL <= 0.0;

HMAX(2,1) = 4: XPOINT(1) to XPOINT(m) are not in ascending order.

HMAX(2,2) gives the position i in XPOINT where this was detected.

IFAIL= 2
At some call to BDYVAL, invalid values were returned, that is, either YL(1) = YL(2) = 0.0, or YR(1) = YR(2) = 0.0 (a programming error in BDYVAL). See the last call of MONIT for details.

This error exit will also occur if p(x) is zero at the point where the boundary condition is imposed. Probably BDYVAL was called with XL equal to a singular end-point a or with XR equal to a singular end-point b.

IFAIL= 3
At some point between XL and XR the value of p(x) computed by COEFFN became zero or changed sign. See the last call of MONIT for details.

IFAIL= 4
MAXIT > 0 on entry, and after MAXIT iterations the eigenvalue had not been found to the required accuracy.

IFAIL= 5
The ‘bracketing’ phase (with parameter IFLAG of MONIT equal to 1) failed to bracket the eigenvalue within ten iterations. This is caused by an error in formulating the problem (for example, q is independent of (lambda)), or by very poor initial estimates of ELAM, DELAM.

On exit ELAM and ELAM + DELAM give the end-points of the interval within which no eigenvalue was located by the routine.

IFAIL= 6
MAXFUN > 0 on entry, and the last iteration was terminated because more than MAXFUN calls to COEFFN were used. See the last call of MONIT for details.

IFAIL= 7
To obtain the desired accuracy the local error tolerance was set so small at the start of some sub-interval that the differential equation solver could not choose an initial step size large enough to make significant progress. See the
last call of MONIT for diagnostics.

IFAIL= 8
At some point inside a sub-interval the step size in the
differential equation solver was reduced to a value too
small to make significant progress (for the same reasons as
with IFAIL = 7). This could be due to pathological behaviour
of p(x) and q(x;(lambda)) or to an unreasonable accuracy
requirement or to the current value of (lambda) making the
equations 'stiff'. See the last call of MONIT for details.

IFAIL= 9
TOL is too small for the problem being solved and the
machine precision is being used. The final value of ELAM
should be a very good approximation to the eigenvalue.

IFAIL= 10
c05azf(*), called by d02kef, has terminated with the error
exit corresponding to a pole of the residual function
f((lambda)). This error exit should not occur, but if it
does, try solving the problem again with a smaller value for
TOL.

IFAIL= 11
A serious error has occurred in an internal call to d02kdy.
Check all subroutine calls and array dimensions. Seek expert
help.

IFAIL= 12
A serious error has occurred in an internal call to
c05azf(*). Check all subroutine calls and array dimensions.
Seek expert help.

HMAX(2,1) holds the failure exit number from the routine
where the failure occurred. In the case of a failure in
c05azf(*), HMAX(2,2) holds the value of parameter IND of
c05azf(*).

Note: error exits with IFAIL = 2, 3, 6, 7, 8, 11 are caused by
being unable to set up or solve the differential equation at some
iteration, and will be immediately preceded by a call of MONIT
giving diagnostic information. For other errors, diagnostic
information is contained in HMAX(2,j), for j=1,2,...,m, where
appropriate.

7. Accuracy

See the discussion in Section 8.2.

8. Further Comments
8.1. Timing

The time taken by the routine depends on the complexity of the coefficient functions, whether they or their derivatives are rapidly changing, the tolerance demanded, and how many iterations are needed to obtain convergence. The amount of work per iteration is roughly doubled when TOL is divided by 16. To make the most economical use of the routine, one should try to obtain good initial values for ELAM and DELAM, and, where appropriate, good asymptotic formulae. The boundary matching points should not be set unnecessarily close to singular points. The extra time needed to compute the eigenfunction is principally the cost of one additional integration once the eigenvalue has been found.

8.2. General Description of the Algorithm

A shooting method, for differential equation problems containing unknown parameters, relies on the construction of a 'miss-distance function', which for given trial values of the parameters measures how far the conditions of the problem are from being met. The problem is then reduced to one of finding the values of the parameters for which the miss-distance function is zero, that is to a root-finding process. Shooting methods differ mainly in how the miss-distance is defined.

This routine defines a miss-distance \( f(\lambda) \) based on the rotation around the origin of the point \( P(x) = (p(x)y'(x), y(x)) \) in the Phase Plane as the solution proceeds from \( a \) to \( b \). The boundary-conditions define the ray (i.e., two-sided line through the origin) on which \( p(x) \) should start, and the ray on which it should finish. The eigenvalue index \( k \) defines the total number of half-turns it should make. Numerical solution is actually done by matching point \( x = c \). Then \( f(\lambda) \) is taken as the angle between the rays to the two resulting points \( P(a) \) and \( P(b) \). A relative scaling of the \( py' \) and \( y \) axes, based on the behaviour of the coefficient functions \( p \) and \( q \), is used to improve the numerical behaviour.

Please see figure in printed Reference Manual

The resulting function \( f(\lambda) \) is monotonic over -

\[
\frac{ddq}{ddq} \infty < (\lambda) < \infty, \text{ increasing if } \frac{-------}{dd(\lambda)} > 0 \text{ and decreasing}
\frac{ddq}{ddq}
\]

if \( ------- < 0 \), with a unique zero at the desired eigenvalue
dd(lambda)

The routine measures \( f(\lambda) \) in units of a half-turn. This means that as \( \lambda \) increases, \( f(\lambda) \) varies by about 1 as each eigenvalue is passed. (This feature implies that the values of \( f(\lambda) \) at successive iterations — especially in the early stages of the iterative process — can be used with suitable extrapolation or interpolation to help the choice of initial estimates for eigenvalues near to the one currently being found.)

The routine actually computes a value for \( f(\lambda) \) with errors, arising from the local errors of the differential equation code and from the asymptotic formulae provided by the user if singular points are involved. However, the error estimate output in DELAM is usually fairly realistic, in that the actual

\[ |(\lambda)-ELAM| \]

error is within an order of magnitude of DELAM.

We pass the values of \((\beta), (\phi), (\rho)\) across through REPORT rather than converting them to values of \(y, y'\) inside D02KEF, for the following reasons. First, there may be cases where auxiliary quantities can be more accurately computed from the Pruefer variables than from \(y\) and \(y'\). Second, in singular problems on an infinite interval \(y\) and \(y'\) may underflow towards the end of the range, whereas the Pruefer variables remain well-behaved. Third, with high-order eigenvalues (and therefore highly oscillatory eigenfunctions) the eigenfunction may have a complete oscillation (or more than one oscillation) between two mesh points, so that values of \(y\) and \(y'\) at mesh points give a very poor representation of the curve. The probable behaviour of the Pruefer variables in this case is that \((\beta)\) and \((\rho)\) vary slowly whilst \((\phi)\) increases quickly: for all three Pruefer variables linear interpolation between the values at adjacent mesh points is probably sufficiently accurate to yield acceptable intermediate values of \((\beta), (\phi), (\rho)\) (and hence of \(y,y')\) for graphical purposes.

Similar considerations apply to the exponentially decaying 'tails' Here \((\phi)\) has approximately constant value whilst \((\rho)\) increases rapidly in the direction of integration, though the step length is generally fairly small over such a range.

If the solution is output through REPORT at \(x\)-values which are too widely spaced, the step length can be controlled by choosing HMAX suitably, or, preferably, by reducing TOL. Both these choices will lead to more accurate eigenvalues and eigenfunctions but at some computational cost.

8.3. The Position of the Shooting Matching Point c
This point is always one of the values \( x \) in array XPOINT. It may be specified using the parameter MATCH. The default value is chosen to be the value of that \( x \), \( 2 \leq i \leq m-1 \), that lies closest to the middle of the interval \([x_2, x_{m-1}]\). If there is a tie, the rightmost candidate is chosen. In particular if there are no break-points then \( c=x_3 \) (=x_2) - that is the shooting is from left to right in this case. A break-point may be inserted purely to move \( c \) to an interior point of the interval, even though the form of the equations does not require it. This often speeds up convergence especially with singular problems.

Note that the shooting method used by the code integrates first from the left-hand end \( x \), then from the right-hand end \( x \), to meet at the matching point \( c \) in the middle. This will of course be reflected in printed or graphical output. The diagram shows a possible sequence of nine mesh points (tau) through (tau) in the order in which they appear, assuming there are just two sub-intervals (so \( m=5 \)).

Figure 1
Please see figure in printed Reference Manual

Since the shooting method usually fails to match up the two 'legs p(x)y' or both, at the matching point \( c \). The code in fact 'shares large jump does not imply an inaccurate eigenvalue, but implies either

(a) a badly chosen matching point: if \( q(x;(\lambda)) \) has a 'humped' shape, \( c \) should be chosen near the maximum value of \( q \), especially if \( q \) is negative at the ends of the interval.

(b) An inherently ill-conditioned problem, typically one where another eigenvalue is pathologically close to the one being sought. In this case it is extremely difficult to obtain an accurate eigenfunction.

In Section 9 below, we find the 11th eigenvalue and corresponding eigenfunction of the equation

\[
y'' + ((\lambda) - x - 2/x)y = 0 \quad \text{on} \quad 0 < x < \infty
\]
the boundary conditions being that \( y \) should remain bounded as \( x \) tends to 0 and \( x \) tends to \( \infty \). The coding of this problem is discussed in detail in Section 8.5.

The choice of matching point \( c \) is open. If we choose \( c=30.0 \) as in the D02KDF(*) example program we find that the exponentially increasing component of the solution dominates and we get extremely inaccurate values for the eigenfunction (though the eigenvalue is determined accurately). The values of the eigenfunction calculated with \( c=30.0 \) are given schematically in Figure 2.

**Figure 2**
Please see figure in printed Reference Manual

If we choose \( c \) as the maximum of the hump in \( q(x;(\lambda)) \) (see (a) above) we instead obtain the accurate results given in Figure 3.

**Figure 3**
Please see figure in printed Reference Manual

8.4. Examples of Coding the COEFFN Routine

Coding COEFFN is straightforward except when break-points are needed. The examples below show:

(a) a simple case,

(b) a case in which discontinuities in the coefficient functions or their derivatives necessitate break-points, and

(c) a case where break-points together with the HMAX parameter are an efficient way to deal with a coefficient function that is well-behaved except over one short interval.

Example A

The modified Bessel equation

\[
x(2x(xy'))'' + ((\lambda)x - (\nu)x) y = 0
\]

Assuming the interval of solution does not contain the origin, dividing through by \( x \), we have \( p(x)=x \),

\[
q(x;(\lambda))=(\lambda)x - (\nu)x /x.
\]
SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
P = X
Q = ELAM*X + NU*NU/X
DQDL = X
RETURN
END

where NU (standing for (nu)) is a real variable that might be defined in a DATA statement, or might be in user-declared COMMON so that its value could be set in the main program.

Example B

The Schroedinger equation

\[ y'' + \left( (\lambda + q(x)) \right) y = 0 \]

\[
\begin{cases}
2 & x - \text{10} & \text{(|x|\leq4),} \\
\text{10} & \text{|x|>4),}
\end{cases}
\]

over some interval 'approximating to (-infty, infty)', say [-20, 20]. Here we need break-points at \pm 4, forming three sub-intervals \(i = [-20, -4], i = [-4, 4], i = [4, 20]\). The code could be

SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
IF (JINT.EQ.2) THEN
   Q = ELAM + X*X - 10.0E0
ELSE
   Q = ELAM + 6.0E0/ABS(X)
ENDIF
P = 1.0E0
DQDL = 1.0E0
RETURN
END

The array XPOINT would contain the values \(x, -20.0, -4.0, +4.0, +20.0, x\) and \(m\) would be 6. The choice of appropriate values for \(x\) and \(x\) depends on the form of the asymptotic formula computed by BDYVAL and the technique is discussed in the next subsection.
Example C

\[ y'' + (\lambda)(1 - 2e^{-100x})y = 0, \text{ over } -10 \leq x \leq 10 \]

Here \( q(x; \lambda) \) is nearly constant over the range except for a sharp inverted spike over approximately \(-0.1 \leq x \leq 0.1\). There is a danger that the routine will build up to a large step size and 'step over' the spike without noticing it. By using break-points - say at \( \pm 0.5 \) - one can restrict the step size near the spike without impairing the efficiency elsewhere.

The code for COEFFN could be

```fortran
SUBROUTINE COEFFN(P,Q,DQDL,X,ELAM,JINT)
P = 1.0E0
DQDL = 1.0E0 - 2.0E0*EXP(-100.0E0*X*X)
Q = ELAM*DQDL
RETURN
END
```

\( \text{XPOINT} \) might contain \(-0.0, -10.0, -0.5, 0.5, 10.0, 10.0 \) (assuming \( \pm 10 \) are regular points) and \( m \) would be 6. \( HMAX(1,j) \), \( j=1,2,3 \) might contain \( 0.0, 0.1 \) and \( 0.0 \).

8.5. Examples of Boundary Conditions at Singular Points

Quoting from Bailey [2] page 243: 'Usually... the differential equation has two essentially different types of solution near a singular point, and the boundary condition there merely serves to distinguish one kind from the other. This is the case in all the standard examples of mathematical physics.'

In most cases the behaviour of the ratio \( p(x)y'/y \) near the point is quite different for the two types of solution. Essentially what the user provides through his \texttt{BDYVAL} routine is an approximation to this ratio, valid as \( x \) tends to the singular point (SP).

The user must decide (a) how accurate to make this approximation or asymptotic formula, for example how many terms of a series to use, and (b) where to place the boundary matching point (BMP) at which the numerical solution of the differential equation takes over from the asymptotic formula. Taking the BMP closer to the SP will generally improve the accuracy of the asymptotic formula, but will make the computation more expensive as the Pruefer
differential equations generally become progressively more ill-behaved as the SP is approached. The user is strongly recommended to experiment with placing the BMPs. In many singular problems quite crude asymptotic formulae will do. To help the user avoid needlessly accurate formulae, D02KEF outputs two 'sensitivity coefficients' \( (\sigma_1, \sigma_2) \) which estimate how much the \\
errors at the BMP’s affect the computed eigenvalue. They are described in detail below, see Section 8.6.

Example of coding BDYVAL:

The example below illustrates typical situations:

\[
\begin{align*}
\frac{d^2y}{dx^2} + (\lambda - x - \cdots)y &= 0, \quad \text{for } 0 < x < \infty \\
\end{align*}
\]

the boundary conditions being that \( y \) should remain bounded as \( x \) tends to 0 and \( x \) tends to \( \infty \).

At the end \( x=0 \) there is one solution that behaves like \( x \) and another that behaves like \( x^{-1} \). For the first of these solutions \( p(x)y'/y \) is asymptotically \( 2/x \) while for the second it is asymptotically \( -1/x \). Thus the desired ratio is specified by setting

\[
YL(1) = x \quad \text{and} \quad YL(2) = 2.0.
\]

At the end \( x=\infty \) the equation behaves like Airy’s equation shifted through \( \lambda \), i.e., like \( y'' - ty = 0 \) where \( t = x - (\lambda) \), so again there are two types of solution. The solution we require behaves as

\[
\begin{align*}
\exp(- \frac{2}{3} - t) / \sqrt{t} \\
\exp(\frac{2}{3} - t) / \sqrt{t}
\end{align*}
\]

and the other as

\[
\begin{align*}
\exp(- \frac{2}{3} - t) / \sqrt{t} \\
\exp(\frac{2}{3} - t) / \sqrt{t}
\end{align*}
\]

once, the desired solution has \( p(x)y'/y^- / \sqrt{t} \) so that we could set

********
YR(1) = 1.0 and YR(2) = -\sqrt{x-(\lambda)}. The complete subroutine might thus be

```fortran
SUBROUTINE BDYVAL(XL,XR,ELAM,YL,YR)
    real XL, XR, ELAM, YL(3), YR(3)
    YL(1) = XL
    YL(2) = 2.0E0
    YR(1) = 1.0E0
    YR(2) = -SQRT(XR - ELAM)
    RETURN
END
```

Clearly for this problem it is essential that any value given by D02KEF to XR is well to the right of the value of ELAM, so that the user must vary the right-hand BMP with the eigenvalue index k function Ai(x), so there is no problem in estimating ELAM.

More accurate asymptotic formulae are easily found - near x=0 by the standard Frobenius method, and near x=\infty by using standard asymptotics for Ai(x), Ai'(x) (see [1], p. 448). For example, by the Frobenius method the solution near x=0 has the expansion

\[
y(x) = x^2 \left( c_0 + c_1 x + c_2 x^2 + \ldots \right)
\]

with

\[
\begin{array}{cccccc}
\lambda & -\lambda c_0 & c_1 & c_2 & c_3 & c_4 \\
0 & 1 & 2 & 3 & 10 & 18
\end{array}
\]

This yields

\[
2
\]

\[
p(x)y' = \frac{p(x)y}{(\lambda)^2} \left( 1 - \frac{\lambda}{10} x + \ldots \right)
\]

8.6. The Sensitivity Parameters (\sigma_l) and (\sigma_r)

The sensitivity parameters (\sigma_l), (\sigma_r) (held in HMAX(1, m-1)
and HMAX(1,m) on output) estimate the effect of errors in the boundary conditions. For sufficiently small errors \( \Delta y \), \( \Delta y' \) in \( y \) and \( py' \) respectively, the relations

\[
\Delta \lambda = (y \Delta y' - py' \Delta y) \sigma_l
\]

\[
\Delta \lambda = (y \Delta y' - py' \Delta y) \sigma_r
\]

are satisfied where the subscripts \( l, r \) denote errors committed at left- and right-hand BMP’s respectively, and \( \Delta \lambda \) denotes the consequent error in the computed eigenvalue.

8.7. Missed Zeros

This is a pitfall to beware of at a singular point. If the BMP is chosen so far from the SP that a zero of the desired eigenfunction lies in between them, then the routine will fail to number of zeros of its eigenfunction, the result will be that:

(a) The wrong eigenvalue will be computed for the given index \( k \) - in fact some \( \lambda \) will be found where \( k' > 1 \).

(b) The same index \( k \) can cause convergence to any of several eigenvalues depending on the initial values of ELAM and DELAM.

It is up to the user to take suitable precautions - for instance by varying the position of the BMP’s in the light of his knowledge of the asymptotic behaviour of the eigenfunction at different eigenvalues.

9. Example

To find the 11th eigenvalue and eigenfunction of the example of Section 8.5, using the simple asymptotic formulae for the boundary conditions.

Comparison of the results from this example program with the corresponding results from DO2KDF(*) example program shows that similar output is produced from the routine MONIT, followed by the eigenfunction values from REPORT, and then a further line of information from MONIT (corresponding to the integration to find the eigenfunction). Final information is printed within the example program exactly as with DO2KDF(*).
Note the discrepancy at the matching point \( c(=\sqrt{\lambda}) \), the maximum of \( q(x;\lambda) \), in this case) between the solutions obtained by integrations from left and right end-points.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

D02 -- Ordinary Differential Equations
D02RAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D02RAF solves the two-point boundary-value problem with general boundary conditions for a system of ordinary differential equations, using a deferred correction technique and Newton iteration.

2. Specification

```fortran
SUBROUTINE D02RAF (N, MNP, NP, NUMBEG, NUMMIX, TOL, INIT, X, Y, IY, ABT, FCN, G, IJAC, JACBF, JACBG, DELEPS, JACEPS, JACGEP, WORK, LWORK, IWORK, LIWORK, IFAIL)

INTEGER N, MNP, NP, NUMBEG, NUMMIX, INIT, IY, IJAC, LWORK, LIWORK, IFAIL
DOUBLE PRECISION TOL, X(MNP), Y(IY,MNP), ABT(N), DELEPS, WORK(LWORK)
EXTERNAL FCN, G, JACBF, JACBG, JACEPS, JACGEP

3. Description

D02RAF solves a two-point boundary-value problem for a system of \( n \) ordinary differential equations in the interval \( (a,b) \) with \( b>a \). The system is written in the form

\[
y'(x) = f_i(x, y_1, y_2, \ldots, y_n), \quad i=1,2,\ldots,n
\]

and the derivatives \( f_i \) are evaluated by a subroutine FCN supplied by the user. With the differential equations (1) must be given a system of \( n \) (nonlinear) boundary conditions.
\[ g \left( y(a), y(b) \right) = 0, \quad i = 1, 2, \ldots, n \]

where

\[ y(x) = [y_1(x), y_2(x), \ldots, y_n(x)]^T. \quad (2) \]

The functions \( g \) are evaluated by a subroutine \( G \) supplied by the user. The solution is computed using a finite-difference technique with deferred correction allied to a Newton iteration to solve the finite-difference equations. The technique used is described fully in Pereyra [1].

The user must supply an absolute error tolerance and may also supply an initial mesh for the finite-difference equations and an initial approximate solution (alternatively a default mesh and approximation are used). The approximate solution is corrected using Newton iteration and deferred correction. Then, additional points are added to the mesh and the solution is recomputed with the aim of making the error everywhere less than the user’s tolerance and of approximately equidistributing the error on the final mesh. The solution is returned on this final mesh.

If the solution is required at a few specific points then these should be included in the initial mesh. If, on the other hand, the solution is required at several specific points then the user should use the interpolation routines provided in Chapter E01 if these points do not themselves form a convenient mesh.

The Newton iteration requires Jacobian matrices

\[
\begin{pmatrix}
\frac{\partial g_1}{\partial y_1} & \frac{\partial g_2}{\partial y_1} & \cdots & \frac{\partial g_n}{\partial y_1} \\
\frac{\partial g_1}{\partial y_2} & \frac{\partial g_2}{\partial y_2} & \cdots & \frac{\partial g_n}{\partial y_2} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial g_1}{\partial y_n} & \frac{\partial g_2}{\partial y_n} & \cdots & \frac{\partial g_n}{\partial y_n}
\end{pmatrix},
\begin{pmatrix}
\frac{\partial g_1}{\partial y_1} & \frac{\partial g_2}{\partial y_1} & \cdots & \frac{\partial g_n}{\partial y_1} \\
\frac{\partial g_1}{\partial y_2} & \frac{\partial g_2}{\partial y_2} & \cdots & \frac{\partial g_n}{\partial y_2} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial g_1}{\partial y_n} & \frac{\partial g_2}{\partial y_n} & \cdots & \frac{\partial g_n}{\partial y_n}
\end{pmatrix}
\]

These may be supplied by the user through subroutines JACOBF for \( \frac{\partial g_i}{\partial y_j} \) and JACOBT for the others. Alternatively the Jacobians may be calculated by numerical differentiation using the algorithm described in Curtis et al [2].
For problems of the type (1) and (2) for which it is difficult to determine an initial approximation from which the Newton iteration will converge, a continuation facility is provided. The user must set up a family of problems

\[ y' = f(x,y,(\epsilon)) , \quad g(y(a),y(b),(\epsilon)) = 0, \tag{3} \]

where \( f = [f_1, f_2, \ldots, f_n] \) etc, and where \( \epsilon \) is a continuation parameter. The choice \( \epsilon = 0 \) must give a problem (3) which is easy to solve and \( \epsilon = 1 \) must define the problem whose solution is actually required. The routine solves a sequence of problems with \( \epsilon \) values

\[ 0 = (\epsilon) < (\epsilon) < \ldots < (\epsilon) = 1. \tag{4} \]

The number \( p \) and the values \( \epsilon \) are chosen by the routine so that each problem can be solved using the solution of its predecessor as a starting approximation. Jacobians \( \frac{df}{d\epsilon} \) and \( \frac{dg}{d\epsilon} \) are required and they may be supplied by the user via routines JACEPS and JACGEP respectively or may be computed by numerical differentiation.

4. References


5. Parameters

1: N -- INTEGER Input
   On entry: the number of differential equations, n.
   Constraint: \( N > 0 \).

2: MNP -- INTEGER Input
On entry: MNP must be set to the maximum permitted number of points in the finite-difference mesh. If LWORK or LIWORK (see below) is too small then internally MNP will be replaced by the maximum permitted by these values. (A warning message will be output if on entry IFAIL is set to obtain monitoring information.) Constraint: MNP >= 32.

3: NP -- INTEGER Input/Output
On entry: NP must be set to the number of points to be used in the initial mesh. Constraint: 4 <= NP <= MNP. On exit: the number of points in the final mesh.

4: NUMBEG -- INTEGER Input
On entry: the number of left-hand boundary conditions (that is the number involving y(a) only). Constraint: 0 <= NUMBEG < N.

5: NUMMIX -- INTEGER Input
On entry: the number of coupled boundary conditions (that is the number involving both y(a) and y(b)). Constraint: 0 <= NUMMIX <= N - NUMBEG.

6: TOL -- DOUBLE PRECISION Input
On entry: a positive absolute error tolerance. If a = x_1 < x_2 < ... < x_NP = b
is the final mesh, z_j(x_i) is the jth component of the approximate solution at x_i, and y_j(x) is the jth component of the true solution of (1) and (2), then, except in extreme circumstances, it is expected that
||z_j(x_i) - y_j(x)|| <= TOL, i = 1, 2, ..., NP; j = 1, 2, ..., n.
Constraint: TOL > 0.0.

7: INIT -- INTEGER Input
On entry: indicates whether the user wishes to supply an initial mesh and approximate solution (INIT /= 0) or whether default values are to be used, (INIT = 0).

8: X(MNP) -- DOUBLE PRECISION array Input/Output
On entry: the user must set X(1) = a and X(NP) = b. If INIT = 0 on entry a default equispaced mesh will be used, otherwise the user must specify a mesh by setting X(i) = x_i for i = 2, 3, ..., NP-1. Constraints:
X(1) < X(NP), if INIT = 0,
X(1) < X(2) < ... < X(NP), if INIT /= 0.
On exit: \( X(1), X(2), \ldots, X(NP) \) define the final mesh (with the returned value of \( NP \)) and \( X(1) = a \) and \( X(NP) = b \).

9: \( Y(IY,MNP) \) -- DOUBLE PRECISION array

Input/Output

On entry: if \( INIT = 0 \), then \( Y \) need not be set.

If \( INIT \neq 0 \), then the array \( Y \) must contain an initial approximation to the solution such that \( Y(j,i) \) contains an approximation to \( y(x) \), \( i = 1, 2, \ldots, NP; j = 1, 2, \ldots, n \).

On exit: the approximate solution \( z(x) \) satisfying (5) on the final mesh, that is

\[
Y(j,i) = z(x), \quad i = 1, 2, \ldots, NP; j = 1, 2, \ldots, n,
\]

where \( NP \) is the number of points in the final mesh. If an error has occurred then \( Y \) contains the latest approximation to the solution. The remaining columns of \( Y \) are not used.

10: \( IY \) -- INTEGER

Input

On entry: the first dimension of the array \( Y \) as declared in the (sub)program from which D02RAF is called.

Constraint: \( IY \geq N \).

11: \( ABT(N) \) -- DOUBLE PRECISION array

Output

On exit: \( ABT(i) \), for \( i = 1, 2, \ldots, n \), holds the largest estimated error (in magnitude) of the \( i \)th component of the solution over all mesh points.

12: FCN -- SUBROUTINE, supplied by the user.

External Procedure

FCN must evaluate the functions \( f \) (i.e., the derivatives \( y' \)) at a general point \( x \) for a given value of (epsilon), the continuation parameter (see Section 3).

Its specification is:

```fortran
SUBROUTINE FCN (X, EPS, Y, F, N)
INTEGER N
DOUBLE PRECISION X, EPS, Y(N), F(N)
```

1: \( X \) -- DOUBLE PRECISION

Input

On entry: the value of the argument \( x \).

2: \( EPS \) -- DOUBLE PRECISION

Input

On entry: the value of the continuation parameter,
(epsilon). This is 1 if continuation is not being used.

3: Y(N) -- DOUBLE PRECISION array
   Input
   On entry: the value of the argument y, for
   \( i = 1, 2, \ldots, n \).

4: F(N) -- DOUBLE PRECISION array
   Output
   On exit: the values of \( f \), for \( i = 1, 2, \ldots, n \).

5: N -- INTEGER
   Input
   On entry: the number of equations.
   FCN must be declared as EXTERNAL in the (sub)program
   from which D02RAF is called. Parameters denoted as
   Input must not be changed by this procedure.

13: G -- SUBROUTINE, supplied by the user.
    External Procedure
   G must evaluate the boundary conditions in equation (3) and
   place them in the array BC.

   Its specification is:

   SUBROUTINE G (EPS, YA, YB, BC, N)
   INTEGER N
   DOUBLE PRECISION EPS, YA(N), YB(N), BC(N)

1: EPS -- DOUBLE PRECISION
   Input
   On entry: the value of the continuation parameter,
   (epsilon). This is 1 if continuation is not being used.

2: YA(N) -- DOUBLE PRECISION array
   Input
   On entry: the value \( y(a) \), for \( i = 1, 2, \ldots, n \).

3: YB(N) -- DOUBLE PRECISION array
   Input
   On entry: the value \( y(b) \), for \( i = 1, 2, \ldots, n \).

4: BC(N) -- DOUBLE PRECISION array
   Output
   On exit: the values \( g(y(a), y(b), (epsilon)) \), for
   \( i = 1, 2, \ldots, n \). These must be ordered as follows:
   (i) first, the conditions involving only \( y(a) \) (see
   NUMBEG description above);
   (ii) next, the NUMMIX coupled conditions involving
   both \( y(a) \) and \( y(b) \) (see NUMMIX description
   above); and,
(iii) finally, the conditions involving only \( y(b) \) (N- NUM Beg-N um Mix).

5: \( N \) -- INTEGER

On entry: the number of equations, \( n \).

\( G \) must be declared as EXTERNAL in the (sub)program from which D02RAF is called. Parameters denoted as Input must not be changed by this procedure.

14: IJAC -- INTEGER

On entry: indicates whether or not the user is supplying Jacobian evaluation routines. If IJAC \!\! = 0 then the user must supply routines JACOBF and JACOBG and also, when continuation is used, routines JACEPS and JACGEP. If IJAC \!\! = 0 numerical differentiation is used to calculate the Jacobian and the routines D02GAZ, D02GAY, D02GAZ and D02GAX

\[
\frac{\partial f}{\partial y_j}(x) \quad \text{for } i, j = 1, 2, \ldots, n,
\]

\[
\frac{\partial f}{\partial x_i}(y) \quad \text{for } j = 1, 2, \ldots, n.
\]

Its specification is:

```
SUBROUTINE JACOBF (X, EPS, Y, F, N)
INTEGER N
DOUBLE PRECISION X, EPS, Y(N), F(N,N)
```

1: \( X \) -- DOUBLE PRECISION

On entry: the value of the argument \( x \).

2: \( EPS \) -- DOUBLE PRECISION

On entry: the value of the continuation parameter (epsilon). This is 1 if continuation is not being used.

3: \( Y(N) \) -- DOUBLE PRECISION array

On entry: the value of the argument \( y \), for \( i = 1, 2, \ldots, n \).

4: \( F(N,N) \) -- DOUBLE PRECISION array

On exit: \( F(i,j) \) must be set to the value of \( \frac{\partial f}{\partial y_j}(x) \), for \( i, j = 1, 2, \ldots, n \).
5:  N -- INTEGER  
    Input
  On entry: the number of equations, n.
JACOBF must be declared as EXTERNAL in the (sub)program
from which D02RAF is called. Parameters denoted as
Input must not be changed by this procedure.

16:  JACOBG -- SUBROUTINE, supplied by the user.
    External Procedure
    ( ddg ) ( ddg )
    ( i ) ( i )
JACOBG must evaluate the Jacobians ( -------) and ( -------)
    ( ddy (a)) ( ddy (b))
    ( j ) ( j )
The ordering of the rows of AJ and BJ must correspond to
the ordering of the boundary conditions described in the
specification of subroutine G above.

Its specification is:

SUBROUTINE JACOBG (EPS, YA, YB, AJ, BJ, N)
    INTEGER N
    DOUBLE PRECISION EPS, YA(N), YB(N), AJ(N,N), BJ
    (N,N)
1:  EPS -- DOUBLE PRECISION  
    Input
  On entry: the value of the continuation parameter,
  (epsilon). This is 1 if continuation is not being used.

2:  YA(N) -- DOUBLE PRECISION array  
    Input
  On entry: the value y (a), for i=1,2,...,n.
    i

3:  YB(N) -- DOUBLE PRECISION array  
    Input
  On entry: the value y (b), for i=1,2,...,n.
    i

4:  AJ(N,N) -- DOUBLE PRECISION array  
    Output
    ddg
    On exit: AJ(i,j) must be set to the value ( -------),
    ddy (a)
    j
    for i,j=1,2,...,n.

5:  BJ(N,N) -- DOUBLE PRECISION array  
    Output
    ddg
    On exit: BJ(i,j) must be set to the value ( -------),
    ddy (b)
for $i,j=1,2\ldots,n$.

6: N -- INTEGER
   Input
   On entry: the number of equations, n.
   JACOBG must be declared as EXTERNAL in the (sub)program
   from which D02RAF is called. Parameters denoted as
   Input must not be changed by this procedure.

17: DELEPS -- DOUBLE PRECISION
    Input/Output
    On entry: DELEPS must be given a value which specifies
    whether continuation is required. If DELEPS <= 0.0 or DELEPS
    >= 1.0 then it is assumed that continuation is not required.
    If 0.0 < DELEPS < 1.0 then it is assumed that continuation
    is required unless DELEPS < square root of machine precision
    when an error exit is taken. DELEPS is used as the increment
    $(\varepsilon) - (\varepsilon)$ (see (4)) and the choice DELEPS = 0.1
    is recommended. On exit: an overestimate of the increment
    $(\varepsilon) - (\varepsilon)$ (in fact the value of the increment
    which would have been tried if the restriction $(\varepsilon) = 1$
    had not been imposed). If continuation was not requested
    then DELEPS = 0.0.

    If continuation is not requested then the parameters JACEPS
    and JACGEP may be replaced by dummy actual parameters in the
    call to D02RAF. (DO2GAZ and DO2GAX respectively may be used
    as the dummy parameters.)

18: JACEPS -- SUBROUTINE, supplied by the user.
    External Procedure
    ddf
    JACEPS must evaluate the derivative $---\frac{d}{d(\varepsilon)}$ given x and
    y if continuation is being used.

    Its specification is:

    SUBROUTINE JACEPS (X, EPS, Y, F, N)
    INTEGER N
    DOUBLE PRECISION X, EPS, Y(N), F(N)

    1: X -- DOUBLE PRECISION
       Input
       On entry: the value of the argument x.

    2: EPS -- DOUBLE PRECISION
       Input
       On entry: the value of the continuation parameter,
(epsilon).

3: Y(N) -- DOUBLE PRECISION array
   Input
   On entry: the solution values y at the point x, for
   i=1,2,...,n.

4: F(N) -- DOUBLE PRECISION array
   Output
   On exit: F(i) must contain the value \frac{df}{d(epsilon)}
   at the point (x,y), for i=1,2,...,n.

5: N -- INTEGER
   Input
   On entry: the number of equations, n.
   JACEPS must be declared as EXTERNAL in the (sub)program
   from which D02RAF is called. Parameters denoted as
   Input must not be changed by this procedure.

19: JACGEP -- SUBROUTINE, supplied by the user.
    External Procedure
    ddg
    JACGEP must evaluate the derivatives \frac{dg}{d(epsilon)}
    if continuation is being used.

Its specification is:

SUBROUTINE JACGEP (EPS, YA, YB, BCEP, N)
   INTEGER N
   DOUBLE PRECISION EPS, YA(N), YB(N), BCEP(N)

1: EPS -- DOUBLE PRECISION
    Input
    On entry: the value of the continuation parameter,
    (epsilon).

2: YA(N) -- DOUBLE PRECISION array
    Input
    On entry: the value of y (a), for i=1,2,...,n.

3: YB(N) -- DOUBLE PRECISION array
    Input
    On entry: the value of y (b), for i=1,2,...,n.

4: BCEP(N) -- DOUBLE PRECISION array
    Output
    On exit: BCEP(i) must contain the value of
    \frac{dg}{d(epsilon)}

    i
--------, for i=1,2,...,n.

dd(epsilon)

5: N -- INTEGER Input
   On entry: the number of equations, n.

JACGEP must be declared as EXTERNAL in the (sub)program
from which D02RAF is called. Parameters denoted as
Input must not be changed by this procedure.

20: WORK(LWORK) -- DOUBLE PRECISION array Workspace

21: LWORK -- INTEGER Input
   On entry:
   the dimension of the array WORK as declared in the
   (sub)program from which D02RAF is called.
   
   Constraints:
   LWORK>={MNP*(3N +6N+2)+4N +3N}.

22: IWORK(LIWORK) -- INTEGER array Workspace

23: LIWORK -- INTEGER Input
   On entry:
   the dimension of the array IWORK as declared in the
   (sub)program from which D02RAF is called.
   Constraints:
   LIWORK>={MNP*(2*N+1)+N, if IJAC /= 0,}
   
   LIWORK>={MNP*(2*N+1)+N +4*N+2, if IJAC = 0}.

24: IFAIL -- INTEGER Input/Output
   For this routine, the normal use of IFAIL is extended to
   control the printing of error and warning messages as well
   as specifying hard or soft failure (see the Essential
   Introduction).

   Before entry, IFAIL must be set to a value with the decimal
   expansion cba, where each of the decimal digits c, b and a
   must have a value of 0 or 1.
   a=0 specifies hard failure, otherwise soft failure;

   b=0 suppresses error messages, otherwise error messages
   will be printed (see Section 6);

   c=0 suppresses warning messages, otherwise warning
   messages will be printed (see Section 6).

   The recommended value for inexperienced users is 110 (i.e.,
   hard failure with all messages printed).

   Unless the routine detects an error (see Section 6), IFAIL
6. Error Indicators and Warnings

Errors detected by the routine:

For each error, an explanatory error message is output on the current error message unit (as defined by X04AAF), unless suppressed by the value of IFAIL on entry.

IFAIL= 1
One or more of the parameters N, MNP, NP, NUMBEG, NUMMIX, TOL, DELEPS, LWORK or LIWORK has been incorrectly set, or X (1) >= X(NP) or the mesh points X(i) are not in strictly ascending order.

IFAIL= 2
A finer mesh is required for the accuracy requested; that is MNP is not large enough. This error exit normally occurs when the problem being solved is difficult (for example, there is a boundary layer) and high accuracy is requested. A poor initial choice of mesh points will make this error exit more likely.

IFAIL= 3
The Newton iteration has failed to converge. There are several possible causes for this error:
(i) faulty coding in one of the Jacobian calculation routines;

(ii) if IJAC = 0 then inaccurate Jacobians may have been calculated numerically (this is a very unlikely cause); or,

(iii) a poor initial mesh or initial approximate solution has been selected either by the user or by default or there are not enough points in the initial mesh. Possibly, the user should try the continuation facility.

IFAIL= 4
The Newton iteration has reached round-off error level. It could be however that the answer returned is satisfactory. The error is likely to occur if too high an accuracy is requested.

IFAIL= 5
The Jacobian calculated by JACOBG (or the equivalent matrix calculated by numerical differentiation) is singular. This may occur due to faulty coding of JACOBG or, in some
circumstances, to a zero initial choice of approximate solution (such as is chosen when INIT = 0).

IFAIL = 6
There is no dependence on (epsilon) when continuation is being used. This can be due to faulty coding of JACEPS or JACGEP or, in some circumstances, to a zero initial choice of approximate solution (such as is chosen when INIT = 0).

IFAIL = 7
DELEPS is required to be less than machine precision for continuation to proceed. It is likely that either the problem (3) has no solution for some value near the current value of (epsilon) (see the advisory print out from D02RAF) or that the problem is so difficult that even with continuation it is unlikely to be solved using this routine. If the latter cause is suspected then using more mesh points initially may help.

IFAIL = 8
Indicates that a serious error has occurred in a call to D02RAF. Check all array subscripts and subroutine parameter lists in calls to D02RAF. Seek expert help.

IFAIL = 9
Indicates that a serious error has occurred in a call to D02RAR. Check all array subscripts and subroutine parameter lists in calls to D02RAR. Seek expert help.

7. Accuracy

The solution returned by the routine will be accurate to the user's tolerance as defined by the relation (5) except in extreme circumstances. The final error estimate over the whole mesh for each component is given in the array ABT. If too many points are specified in the initial mesh, the solution may be more accurate than requested and the error may not be approximately equidistributed.

8. Further Comments

There are too many factors present to quantify the timing. The time taken by the routine is negligible only on very simple problems.

The user is strongly recommended to set IFAIL to obtain self-explanatory error messages, and also monitoring information about the course of the computation.

In the case where the user wishes to solve a sequence of similar
problems, the use of the final mesh and solution from one case as
the initial mesh is strongly recommended for the next.

9. Example

We solve the differential equation

\[ y'''' = -0.5 y''' - 2(\epsilon)(1-y') \]

with \((\epsilon)=1\) and boundary conditions

\[ y(0)=y'(0)=0, \quad y'(10)=1 \]

to an accuracy specified by \(\text{TOL}=1.0\times 10^{-4}\). The continuation facility
is used with the continuation parameter \((\epsilon)\) introduced as
in the differential equation above and with \(\text{DELEPS} = 0.1\)
initially. (The continuation facility is not needed for this
problem and is used here for illustration.)

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

———

NagOrdinaryDifferentialEquationsPackage (NAGD02)

Exports:

\[ \text{d02bbf} \quad \text{d02bhf} \quad \text{d02cjf} \quad \text{d02ejf} \quad \text{d02gaf} \]
\[ \text{d02gbf} \quad \text{d02kef} \quad \text{d02kef} \quad \text{d02raf} \]

— package NAGD02 NagOrdinaryDifferentialEquationsPackage ——
CHAPTER 15. CHAPTER N

)abbrev package NAGD02 NagOrdinaryDifferentialEquationsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Mon Jun 20 17:56:33 1994
++ Description:
++ This package uses the NAG Library to calculate the numerical solution of
++ ordinary differential equations. There are two main types of problem,
++ those in which all boundary conditions are specified at one point
++ (initial-value problems), and those in which the boundary
++ conditions are distributed between two or more points (boundary-
++ value problems and eigenvalue problems). Routines are available
++ for initial-value problems, two-point boundary-value problems and
++ Sturm-Liouville eigenvalue problems.

NagOrdinaryDifferentialEquationsPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports ==> with
d02bbf : (DoubleFloat,Integer,Integer,Integer,_
    DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp7(FCN)),_
    Union(fn:FileName,fp:Asp8(OUTPUT))) -> Result
++ d02bbf(xend,m,n,irelab,x,y,tol,ifail,fcn,output)
++ integrates a system of first-order ordinary differential
++ equations over an interval with suitable initial conditions,
++ using a Runge-Kutta-Merson method, and returns the solution at
++ points specified by the user.
++ See \downlink{Manual Page}{manpageXXd02bbf}.
d02bhf : (DoubleFloat,Integer,Integer,DoubleFloat,_
    DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
    Union(fn:FileName,fp:Asp9(G)),_
    Union(fn:FileName,fp:Asp7(FCN))) -> Result
++ d02bhf(xend,n,irelab,hmax,x,y,tol,ifail,g,fcn)
++ integrates a system of first-order ordinary differential
++ equations over an interval with suitable initial conditions,
++ using a Runge-Kutta-Merson method, until a user-specified
++ function of the solution is zero.
++ See \downlink{Manual Page}{manpageXXd02bhf}.
d02cjf : (DoubleFloat,Integer,Integer,DoubleFloat,_
    String,DoubleFloat,Matrix DoubleFloat,DoubleFloat,_,
    Union(fn:FileName,fp:Asp9(G)),Union(fn:FileName,fp:Asp7(FCN)),_,
    Union(fn:FileName,fp:Asp8(OUTPUT))) -> Result
++ d02cjf(xend,m,n,tol,relabs,x,y,ifail,g,fcn,output)
++ integrates a system of first-order ordinary differential
++ equations over a range with suitable initial conditions, using a
++ variable-order, variable-step Adams method until a user-specified
++ function, if supplied, of the solution is zero, and returns the
++ solution at points specified by the user, if desired.
++ See \downlink{Manual Page}{manpageXXd02cjf}.
d02ejf : (DoubleFloat, Integer, Integer, String, __
    Integer, DoubleFloat, Matrix DoubleFloat, DoubleFloat, Integer, _
    Union(fn: FileName, fp: Asp9(G)), Union(fn: FileName, fp: Asp7(FCN)), _
    Union(fn: FileName, fp: Asp31(PEDERV)), _
    Union(fn: FileName, fp: Asp8(OUTPUT))) -> Result
++ d02ejf(xend, m, n, relabs, iw, x, y, tol, ifail, g, fcn, pederv, output)
++ integrates a stiff system of first-order ordinary
++ differential equations over an interval with suitable initial
++ conditions, using a variable-order, variable-step method
++ implementing the Backward Differentiation Formulae (BDF), until a
++ user-specified function, if supplied, of the solution is zero,
++ and returns the solution at points specified by the user, if
++ desired.
++ See \downlink{Manual Page}{manpageXXd02ejf}.

d02gaf : (Matrix DoubleFloat, Matrix DoubleFloat, Integer, DoubleFloat, __
    DoubleFloat, DoubleFloat, Integer, Integer, Matrix DoubleFloat, _
    Integer, Integer, Union(fn: FileName, fp: Asp7(FCN))) -> Result
++ d02gaf(u, v, n, a, b, tol, mnp, lw, liw, x, np, ifail, fcn)
++ solves the two-point boundary-value problem with assigned
++ boundary values for a system of ordinary differential equations,
++ using a deferred correction technique and a Newton iteration.
++ See \downlink{Manual Page}{manpageXXd02gaf}.

d02gbf : (DoubleFloat, DoubleFloat, Integer, DoubleFloat, __
    Integer, Integer, Integer, Integer, Matrix DoubleFloat, Matrix DoubleFloat, _
    Matrix DoubleFloat, Matrix DoubleFloat, Integer, Integer, _
    Matrix DoubleFloat, Matrix DoubleFloat, Integer, Integer, _
    Union(fn: FileName, fp: Asp77(FCNF)), _
    Union(fn: FileName, fp: Asp78(FCNG))) -> Result
++ d02gbf(a, b, n, tol, mnp, lw, liw, c, d, gam, x, np, ifail, fcn, fcnf, fcng)
++ solves a general linear two-point boundary value problem
++ for a system of ordinary differential equations using a deferred
++ correction technique.
++ See \downlink{Manual Page}{manpageXXd02gbf}.

d02kef : (Matrix DoubleFloat, Integer, Integer, DoubleFloat, __
    Integer, Integer, Integer, Integer, Integer, Matrix DoubleFloat, Integer, _
    Integer, Integer, Union(fn: FileName, fp: Asp10(COEFFN)), _
    Union(fn: FileName, fp: Asp80(BDYVAL))) -> Result
++ d02kef(xpoint, m, k, tol, maxfun, match, elam, delam, _
    hmax, maxit, ifail, coeffn, bdyval)
++ finds a specified eigenvalue of a regular singular second-
++ order Sturm-Liouville system on a finite or infinite range, using
++ a Pruefer transformation and a shooting method. It also reports
++ values of the eigenfunction and its derivatives. Provision is
++ made for discontinuities in the coefficient functions or their
++ derivatives.
++ See \downlink{Manual Page}{manpageXXd02kef}.
++ ASP domains Asp12 and Asp33 are used to supply default
++ subroutines for the MONIT and REPORT arguments via their
++ \axiomOp{outputAsFortran} operation.
CHAPTER 15. CHAPTER N

Integer, Union(fn: FileName, fp: Asp10(COEFFN)),
Union(fn: FileName, fp: Asp80(BDYVAL)), FileName, FileName) -> Result
++ d02kef(xpoint,m,k,tol,maxfun,match,elam,delam,
++ hmax,maxit,ifail,coeffn,bdyval,monit,report)
++ finds a specified eigenvalue of a regular singular second-
++ order Sturm-Liouville system on a finite or infinite range, using
++ a Pruefer transformation and a shooting method. It also reports
++ values of the eigenfunction and its derivatives. Provision is
++ made for discontinuities in the coefficient functions or their
++ derivatives.
++ See \downlink{Manual Page}{manpageXXd02kef}.
++ Files \spad{monit} and \spad{report} will be used to define
++ the subroutines for the
++ MONIT and REPORT arguments.
++ See \downlink{Manual Page}{manpageXXd02gbf}.

d02raf : (Integer,Integer,Integer,Integer,_
DoubleFloat,Integer,Integer,Integer,Integer,Integer,_
Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,Integer,_
Union(fn: FileName, fp: Asp41(FCN,JACOBF,JACEPS)),_
Union(fn: FileName, fp: Asp42(G,JACOBG,JACGEP))) -> Result
++ d02raf(n,mnp,numbeg,nummix,tol,init,iy,ijac,lwork,
++ liwork,np,x,y,deleps,ifail,fcn,g)
++ solves the two-point boundary-value problem with general
++ boundary conditions for a system of ordinary differential
++ equations, using a deferred correction technique and Newton
++ iteration.
++ See \downlink{Manual Page}{manpageXXd02gbf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn: FileName, fp: Asp7(FCN))
import Union(fn: FileName, fp: Asp8(OUTPUT))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(Matrix DoubleFloat)

d02bbf(xendArg:DoubleFloat,mArg:Integer,nArg:Integer,_
irelabArg:Integer,xArg:DoubleFloat,yArg:Matrix DoubleFloat,_
tolArg:DoubleFloat,ifailArg:Integer,_
fcnArg:Union(fn: FileName, fp: Asp7(FCN)),_)
outputArg: Union(fn: FileName, fp: Asp8(OUTPUT)): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
  if fcnArg case fn
    then outputAsFortran(fcnArg.fn)
  else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
  if outputArg case fn
    then outputAsFortran(outputArg.fn)
  else outputAsFortran(outputArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fcnFilename, outputFilename]$Lisp,_
  "d02bbf",_
  ["result":S,"w":S,"fcn":S,"output":S]$Lisp,_
  [["double":S,"xend":S,["result":S,"m":S,"n":S]$Lisp_,
    "x":S,["y":S,"n":S]$Lisp,"tol":S,["w":S,"n":S,7$Lisp]$Lisp]$Lisp,_
    "fcn":S,"output":S]$Lisp_,
  [[xendArg::Any,mArg::Any,nArg::Any,irelabArg::Any,xArg::Any,_
    tolArg::Any,ifailArg::Any,yArg::Any ]])_]
pretend List (Record(key:Symbol,entry:Any))$Result

"d02bhf(xendArg::DoubleFloat,nArg::Integer,irelabArg::Integer,_
  hmaxArg::DoubleFloat,xArg::DoubleFloat,yArg::Matrix DoubleFloat,_
  tolArg::DoubleFloat,ifailArg::Integer,_
  gArg:Union(fn:FileName,fp:Asp9(G))): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
  if gArg case fn
    then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
  if fcnArg case fn
    then outputAsFortran(fcnArg.fn)
  else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([gFilename,fcnFilename]$Lisp,_
  "d02bhf",_"xend":S,"n":S,"irelab":S,"hmax":S,"x":S_,
  ["w":S,"g":S,"fcn":S]$Lisp,_
    "tol":S,["w":S,"n":S,7$Lisp]$Lisp,"g":S,"fcn":S]$Lisp_,
  ["x":S,"y":S,"tol":S,"ifail":S]$Lisp,]_
CHAPTER 15. CHAPTER N

```
pretend List (Record(key:Symbol,entry:Any))

\[
\text{d02cjf}(x\text{endArg}: \text{DoubleFloat}, m\text{Arg}: \text{Integer}, n\text{Arg}: \text{Integer},
\text{tolArg}: \text{DoubleFloat}, \text{relabsArg}: \text{String}, x\text{Arg}: \text{DoubleFloat},
\text{yArg}: \text{Matrix} \text{DoubleFloat}, \text{ifailArg}: \text{Integer},
\text{gArg}: \text{Union}(\text{fn:FileName,fp:Asp9(G)}),
\text{fcnArg}: \text{Union}(\text{fn:FileName,fp:Asp7(FCN)}),
\text{pedervArg}: \text{Union}(\text{fn:FileName,fp:Asp31(PEDERV)}):
\]
\[
\text{Result} == \text{invokeNagman}([\text{gFilename,fcnFilename,outputFilename}]\text{Lisp},
\text{"d02cjf"},
[\"xend\":S,\"m\":S,\"n\":S,\"tol\":S,\"relabs\":S_,
\"x\":S,\"ifail\":S,\"g\":S,\"fcn\":S,\"output\":S_,
\"result\":S,\"y\":S,\"w\":S]\text{Lisp}_-,
[\"result\":S,\"w\":S,\"g\":S,\"fcn\":S,\"output\":S]\text{Lisp}_-,
[\"double\":S,\"xend\":S,\"tol\":S,[\"result\":S,\"m\":S,\"n\":S]\text{Lisp}_-,
\"x\":S,[\"y\":S,\"n\":S]\text{Lisp},[\"w\":S,[\"*:S_,
[\"*:S,21\text{Lisp},\"n\":S]\text{Lisp},28\text{Lisp}]\text{Lisp}]\text{Lisp},\"g\":S_,
\"fcn\":S,\"output\":S]\text{Lisp}_-,
[\"integer\":S,\"m\":S,\"n\":S,\"ifail\":S]\text{Lisp}_-
,[\"character\":S,\"relabs\":S]\text{Lisp}\text{Lisp}_-,
[\"result\":S,\"x\":S,\"y\":S,\"ifail\":S]\text{Lisp}_-,
[\[\text{xArg:Symbol,entry:Any}]\text{Lisp}_-
\text{pretend List (Record(key:Symbol,entry:Any))}\text{Result}
```

```
\[
\text{d02ejf}(x\text{endArg}: \text{DoubleFloat}, m\text{Arg}: \text{Integer}, n\text{Arg}: \text{Integer},
\text{tolArg}: \text{DoubleFloat}, \text{relabsArg}: \text{String}, x\text{Arg}: \text{DoubleFloat},
\text{yArg}: \text{Matrix} \text{DoubleFloat}, \text{ifailArg}: \text{Integer},
\text{gArg}: \text{Union}(\text{fn:FileName,fp:Asp9(G)}),
\text{fcnArg}: \text{Union}(\text{fn:FileName,fp:Asp7(FCN)}),
\text{pedervArg}: \text{Union}(\text{fn:FileName,fp:Asp31(PEDERV)}),
```
outputArg:Union(fn:FileName,fp:Asp8(OUTPUT)): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
  then outputAsFortran("fcnArg.fn")
  else outputAsFortran("fcnArg.fp")
popFortranOutputStack()$FOP
pushFortranOutputStack(pedervFilename := aspFilename "pederv")$FOP
if pedervArg case fn
  then outputAsFortran(pedervArg.fn)
  else outputAsFortran(pedervArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
if outputArg case fn
  then outputAsFortran(outputArg.fn)
  else outputAsFortran(outputArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([gFilename,fcnFilename,pedervFilename,_
outputFilename]$Lisp,_
"d02ejf",_
["xend":S,"m":S,"m":S,"relabs":S,"iw":S_
["double":S,"xend":S,"result":S,"m":S,"n":S]$Lisp_
"g":S,"fcn":S,"pederv":S,"output":S]$Lisp_
,["integer":S,"m":S,"n":S,"iw":S,"ifail":S_
]$Lisp_
,"["character":S,"relabs":S]$Lisp_
]$Lisp_
,pretend List (Record(key:Symbol,entry:Any))$Result

outputArg:Union(fn:FileName,fp:Asp8(OUTPUT)): Result ==
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
  then outputAsFortran(gArg.fn)
  else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
  then outputAsFortran("fcnArg.fn")
  else outputAsFortran("fcnArg.fp")
popFortranOutputStack()$FOP
pushFortranOutputStack(pedervFilename := aspFilename "pederv")$FOP
if pedervArg case fn
  then outputAsFortran(pedervArg.fn)
  else outputAsFortran(pedervArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(outputFilename := aspFilename "output")$FOP
if outputArg case fn
  then outputAsFortran(outputArg.fn)
  else outputAsFortran(outputArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([gFilename,fcnFilename,pedervFilename,_
outputFilename]$Lisp,_
"d02ejf",_
["xend":S,"m":S,"m":S,"relabs":S,"iw":S_
["double":S,"xend":S,"result":S,"m":S,"n":S]$Lisp_
"g":S,"fcn":S,"pederv":S,"output":S]$Lisp_
,["integer":S,"m":S,"n":S,"iw":S,"ifail":S_
]$Lisp_
,"["character":S,"relabs":S]$Lisp_
]$Lisp_
,pretend List (Record(key:Symbol,entry:Any))$Result
popFortranOutputStack()$FOP
[(invokeNagman([fcnfFilename]$Lisp,_,
"d02gaf",_,
,"w":S,"v":S,"y":S,"x":S,"w":S_
,"iw":S]$Lisp,_
["y":S,"w":S,"iw":S,"fcn":S]$Lisp,_
[["double":S,["u":S,"n":S,$Lisp],["v":S,"n":S,$Lisp]$Lisp_
["x":S,"mnp":S]$Lisp,["iw":S,"iw":S]$Lisp_
,"fcn":S]$Lisp_
]$Lisp_,
[[(nArg::Any,aArg::Any,bArg::Any,tolArg::Any,mnpArg::Any,_
lwArg::Any,liwArg::Any,mpArg::Any,ifailArg::Any,uArg::Any,_
vArg::Any,xArg::Any ])]_0List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

d02gbf(aArg:DoubleFloat,bArg:DoubleFloat,nArg:Integer,_
tolArg:DoubleFloat,mnpArg:Integer,lwArg:Integer,_
lwArg:Integer,cArg:Matrix DoubleFloat,dArg:Matrix DoubleFloat,_
gamArg:Matrix DoubleFloat,xArg:Matrix DoubleFloat,mpArg::Any,fpArg::Any,ifailArg::Integer,_
fcnfArg:Union(fn:FileName,fp:Asp77(FCNF)),_,
fcngArg:Union(fn:FileName,fp:Asp78(FCNNG))): Result ==
pushFortranOutputStack(fcnfFilename := aspFilename "fcnf")$FOP
if fcnfArg case fn
then outputAsFortran(fcnfArg.fn)
else outputAsFortran(fcnfArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(fcnfFilename := aspFilename "fcnf")$FOP
if fcnfArg case fn
then outputAsFortran(fcnfArg.fn)
else outputAsFortran(fcnfArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fcnfFilename,fcngFilename]$Lisp,_,
"d02gbf",_,
,"w":S,"v":S,"y":S,"x":S,"w":S_
,"iw":S]$Lisp,_
["y":S,"w":S,"iw":S,"fcn":S]$Lisp,_
["gam":S,"n":S,$Lisp,["x":S,"mnp":S]$Lisp_
,["w":S,"lw":S]$Lisp,"fcng":S]$Lisp_]
d02kef(xpointArg:Matrix DoubleFloat,mArg:Integer,kArg:Integer,_
tolArg:DoubleFloat,elamArg:DoubleFloat,delamArg:DoubleFloat,hmaxArg:Matrix DoubleFloat,_
maxitArg:Integer,cArg::Any,dArg::Any,elamArg::Any,delamArg::Any,matchArg::Any,_
coeffnArg::Any,ifailArg::Any)
}

@List Any$Lisp$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

invokeNagman([coeffnFilename,bdyvalFilename,monitFilename,reportFilename]$Lisp,_,
"d02kef",_,
["m":S,"k":S,"tol":S,"maxfun":S,"match":S_,
"elam":S,"delam":S,"ifail":S,"coeffn":S_,
[["double":S,"xpoint":S,"m":S]$Lisp,"tol":S_,
["integer":S,"m":S,"k":S,"maxfun":S,"match":S_,
,"maxit":S,"ifail":S]$Lisp_]
]$Lisp_
pretend List (Record(key:Symbol,entry:Any))$Result
d02kef(xpointArg:Matrix DoubleFloat,mArg:Integer,kArg:Integer,_
  tolArg:DoubleFloat,maxfunArg:Integer,matchArg:Integer,_
  elamArg:DoubleFloat,delamArg:DoubleFloat,hmaxArg:Matrix DoubleFloat,_
  maxitArg:Integer,ifailArg:Integer,_
  coeffnArg:Union(fn:FileName,fp:Asp10(COEFFN)),_
  bdyvalArg:Union(fn:FileName,fp:Asp80(BDYVAL)),_
  monitArg:FileName,reportArg:FileName): Result ==
pushFortranOutputStack(coeffnFilename := aspFilename "coeffn")$FOP
  if coeffnArg case fn
    then outputAsFortran(coeffnArg.fn)
    else outputAsFortran(coeffnArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(bdyvalFilename := aspFilename "bdyval")$FOP
  if bdyvalArg case fn
    then outputAsFortran(bdyvalArg.fn)
    else outputAsFortran(bdyvalArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(monitFilename := aspFilename "monit")$FOP
  outputAsFortran(monitArg)
popFortranOutputStack()$FOP
pushFortranOutputStack(reportFilename := aspFilename "report")$FOP
  outputAsFortran(reportArg)
popFortranOutputStack()$FOP
[(invokeNagman([coeffnFilename,bdyvalFilename,monitFilename,_
                   reportFilename]$Lisp,_
                   "d02kef",_
                   ["m"::S,"k"::S,"tol"::S,"maxfun"::S,"match"::S_
                   ["double"::S,"xpoint"::S,"m"::S]$Lisp,"tol"::S_
                   ["integer"::S,"m"::S,"k"::S,"maxfun"::S,"match"::S_
                   ,"maxit"::S,"ifail"::S]$Lisp_
               ]$Lisp,_
               [[[mArg::Any,kArg::Any,tolArg::Any,maxfunArg::Any,_
                 matchArg::Any,elamArg::Any,delamArg::Any,maxitArg::Any,_
                 ifailArg::Any,xpointArg::Any,hmaxArg::Any ]]_n
               @List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result
d02raf(nArg:Integer,mnpArg:Integer,numbegArg:Integer,_
  nummixArg:Integer,ioArg:Integer,tolArg:DoubleFloat,initArg:Integer,_
  iyArg:Integer,ijacArg:Integer,lworkArg:Integer,_
  liworkArg:Integer,npArg:Integer,xArg:Matrix DoubleFloat,_
  ntauXArg:Matrix DoubleFloat,ntauYArg:Matrix DoubleFloat,
  ntauXYArg:Matrix DoubleFloat,ntauXZArg:Matrix DoubleFloat,ntauYZArg:Matrix DoubleFloat)
yArg: Matrix DoubleFloat, delepsArg: DoubleFloat, ifailArg: Integer,
fcnArg: Union(fn: FileName, fp: Asp41(FCN, JACOBF, JACEPS)),
gArg: Union(fn: FileName, fp: Asp42(G, JACOBG, JACGEP))): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
then outputAsFortran(fcnArg.fn)
else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(gFilename := aspFilename "g")$FOP
if gArg case fn
then outputAsFortran(gArg.fn)
else outputAsFortran(gArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([fcnFilename, gFilename]$Lisp,
"d02raf",
["n": S, "mnp": S, "numbeg": S, "nummix": S, "tol": S,
", "init": S, "iy": S, "ijac": S, "lwork": S, "liwork": S,
", "np": S, "deleps": S, "ifail": S, "fcn": S, "g": S,
", "abt": S, "x": S, "y": S, "work": S, "iwork": S,
]$Lisp,
["abt": S, "work": S, "iwork": S, "fcn": S, "g": S]$Lisp,
["double": S, "tol": S, ["abt": S, "n": S]$Lisp,
,["x": S, "mnp": S]$Lisp, ["y": S, "mnp": S]$Lisp,
"deleps": S, ["work": S, "lwork": S]$Lisp, ["fcn": S, "g": S]$Lisp,
["integer": S, "n": S, "mnp": S, "numbeg": S,
", "nummix": S, "init": S, "iy": S, "ijac": S, "lwork": S, "liwork": S,
"np": S, "ifail": S, ["iwork": S, "liwork": S]$Lisp]$Lisp]
]$Lisp,
["abt": S, "np": S, "x": S, "y": S, "deleps": S, "ifail": S]$Lisp,
[(nArg:: Any, npArg:: Any, numbegArg:: Any, nummixArg:: Any, tolArg:: Any,
initArg:: Any, iyArg:: Any, ijacArg:: Any, lworkArg:: Any, liworkArg:: Any,
npArg:: Any, delepsArg:: Any, ifailArg:: Any, xArg:: Any, yArg:: Any )]
@List Any]$Lisp]$Lisp)
pretend List (Record(key: Symbol, entry: Any))$Result

— NAGD02.dotabb —

"NAGD02" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGD02"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGD02" -> "ALIST"
package NAGD03 NagPartialDifferentialEquationsPackage

— NagPartialDifferentialEquationsPackage.input —

)set break resume
)sys rm -f NagPartialDifferentialEquationsPackage.output
)spool NagPartialDifferentialEquationsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NagPartialDifferentialEquationsPackage
--R
--R NagPartialDifferentialEquationsPackage is a package constructor
--R Abbreviation for NagPartialDifferentialEquationsPackage is NAGD03
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGD03
--R
--R------------------------------------------------ Operations --------------------------------
--R d03edf : (Integer,Integer,Integer,Integer,DoubleFloat,Integer,Matrix(DoubleFloat),Matrix(DoubleFloat),Matrix(DoubleFloat),Integer) -> Result
--R d03eef : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,String,Integer,Union(fn: FileName,fp: Asp73(PDEF)),Union(fn: FileName,fp: Asp74(BNDY))) -> Result
--R d03faf : ...
--R
--E 1

)spool
)lisp (bye)

——

— NagPartialDifferentialEquationsPackage.help —

This package uses the NAG Library to solve partial differential equations.

D03 -- Partial Differential Equations Introduction -- D03
Chapter D03
Partial Differential Equations

1. Scope of the Chapter

This chapter is concerned with the solution of partial differential equations.

2. Background to the Problems
The definition of a partial differential equation problem includes not only the equation itself but also the domain of interest and appropriate subsidiary conditions. Indeed, partial differential equations are usually classified as elliptic, hyperbolic or parabolic according to the form of the equation and the form of the subsidiary conditions which must be assigned to produce a well-posed problem. Ultimately it is hoped that this chapter will contain routines for the solution of equations of each of these types together with automatic mesh generation routines and other utility routines particular to the solution of partial differential equations. The routines in this chapter will often call upon routines from the Linear Algebra Chapter F04 -- Simultaneous Linear Equations.

The classification of partial differential equations is easily described in the case of linear equations of the second order in two independent variables, i.e., equations of the form

\[ au + 2bu + cu + du + eu + fu + g = 0, \]

where \( a, b, c, d, e, f \) and \( g \) are functions of \( x \) and \( y \) only.

Equation (1) is called elliptic, hyperbolic or parabolic according as \( ac - b^2 \) is positive, negative or zero. Useful definitions of the concepts of elliptic, hyperbolic and parabolic character can also be given for differential equations in more than two independent variables, for systems and for nonlinear differential equations.

For elliptic equations, of which Laplace's equation

\[ u + u = 0 \]

\( xx \quad yy \)

is the simplest example of second order, the subsidiary conditions take the form of boundary conditions, i.e., conditions which provide information about the solution at all points of a closed boundary. For example, if equation (2) holds in a plane domain \( D \) bounded by a contour \( C \), a solution \( u \) may be sought subject to the condition

\[ u = f \quad \text{on} \quad C, \]

where \( f \) is a given function. The condition (3) is known as a Dirichlet boundary condition. Equally common is the Neumann boundary condition

\[ u' = g \quad \text{on} \quad C, \]
which is one form of a more general condition
\[ u' + fu = g \quad \text{on } C, \] (5)
where \( u' \) denotes the derivative of \( u \) normal to the contour \( C \) and \( f \) and \( g \) are given functions. Provided that \( f \) and \( g \) satisfy certain restrictions, condition (5) yields a well-posed boundary value problem for Laplace’s equation. In the case of the Neumann problem, one further piece of information, e.g. the value of \( u \) at a particular point, is necessary for uniqueness of the solution. Boundary conditions similar to the above are applicable to more general second order elliptic equations, whilst two such conditions are required for equations of fourth order.

For hyperbolic equations, the wave equation
\[ u_{tt} - u_{xx} = 0 \] (6)
is the simplest example of second order. It is equivalent to a first order system
\[ u_t - v = 0, \quad v_t - u = 0. \] (7)
The subsidiary conditions may take the form of initial conditions, i.e., conditions which provide information about the solution at points on a suitable open boundary. For example, if equation (6) is satisfied for \( t > 0 \), a solution \( u \) may be sought such that
\[ u(x,0) = f(x), \quad u_t(x,0) = g(x), \] (8)
where \( f \) and \( g \) are given functions. This is an example of an initial value problem, sometimes known as Cauchy’s problem.

For parabolic equations, of which the heat conduction equation
\[ u_t - u_{xx} = 0 \] (9)
is the simplest example, the subsidiary conditions always include some of initial type and may also include some of boundary type. For example, if equation (9) is satisfied for \( t > 0 \) and \( 0 < x < 1 \), a solution \( u \) may be sought such that
\[ u(x,0) = f(x), \quad 0 < x < 1, \] (10)
and
\begin{equation}
\begin{aligned}
\quad u(0,t) &= 0, \quad u(1,t) = 1, \quad t > 0.
\end{aligned}
\end{equation}

This is an example of a mixed initial/boundary value problem.

For all types of partial differential equations, finite difference methods (Mitchell and Griffiths [5]) and finite element methods (Wait and Mitchell [9]) are the most common means of solution and such methods obviously feature prominently either in this chapter or in the companion NAG Finite Element Library. Many of the utility routines in this chapter are concerned with the solution of the large sparse systems of equations which arise from the finite difference and finite element methods.

Alternative methods of solution are often suitable for special classes of problems. For example, the method of characteristics is the most common for hyperbolic equations involving time and one space dimension (Smith [7]). The method of lines (see Mikhlin and Smolitsky [4]) may be used to reduce a parabolic equation to a (stiff) system of ordinary differential equations, which may be solved by means of routines from Chapter D02 -- Ordinary Differential Equations. Similarly, integral equation or boundary element methods (Jaswon and Symm [3]) are frequently used for elliptic equations. Typically, in the latter case, the solution of a boundary value problem is represented in terms of certain boundary functions by an integral expression which satisfies the differential equation throughout the relevant domain. The boundary functions are obtained by applying the given boundary conditions to this representation. Implementation of this method necessitates discretisation of only the boundary of the domain, the dimensionality of the problem thus being effectively reduced by one. The boundary conditions yield a full system of simultaneous equations, as opposed to the sparse systems yielded by the finite difference and finite element methods, but the full system is usually of much lower order. Solution of this system yields the boundary functions, from which the solution of the problem may be obtained, by quadrature, as and where required.

2.1. References


3. Recommendations on Choice and Use of Routines

The choice of routine will depend first of all upon the type of partial differential equation to be solved. At present no special allowances are made for problems with boundary singularities such as may arise at corners of domains or at points where boundary conditions change. For such problems results should be treated with caution.

Users may wish to construct their own partial differential equation solution software for problems not solvable by the routines described in Sections 3.1 to 3.4 below. In such cases users can employ appropriate routines from the Linear Algebra Chapters to solve the resulting linear systems; see Section 3.5 for further details.

3.1. Elliptic Equations

The routine D03EDF solves a system of seven-point difference equations in a rectangular grid (in two dimensions), using the multigrid iterative method. The equations are supplied by the user, and the seven-point form allows cross-derivative terms to be represented (see Mitchell and Griffiths [5]). The method is particularly efficient for large systems of equations with diagonal dominance.

The routine D03EEF discretises a second-order equation on a two-dimensional rectangular region using finite differences and a seven-point molecule. The routine allows for cross-derivative
terms, Dirichlet, Neumann or mixed boundary conditions, and
either central or upwind differences. The resulting seven-
diagonal difference equations are in a form suitable for passing
directly to the multigrid routine D03EDF, although other solution
methods could easily be used.

The routine D03FAF, based on the routine HW3CRT from FISHPACK
(Swarztrauber and Sweet [8]), solves the Helmholtz equation in a
three-dimensional cuboidal region, with any combination of
Dirichlet, Neumann or periodic boundary conditions. The method
used is based on the fast Fourier transform algorithm, and is
likely to be particularly efficient on vector-processing
machines.

3.2. Hyperbolic Equations

There are no routines available yet for the solution of these
equations.

3.3. Parabolic Equations

There are no routines available yet for the solution of these
equations.

But problems in two space dimensions plus time may be treated as
a succession of elliptic equations [1], [6] using appropriate
D03E- routines or one may use codes from the NAG Finite Element
Library.

3.4. Utility Routines

There are no utility routines available yet, but routines are
available in the Linear Algebra Chapters for the direct and
iterative solution of linear equations. Here we point to some of
the routines that may be of use in solving the linear systems
that arise from finite difference or finite element
approximations to partial differential equation solutions.
Chapters F01 and F04 should be consulted for further information
and for the routine documents. Decision trees for the solution of
linear systems are given in Section 3.5 of the F04 Chapter
Introduction.

The following routines allow the direct solution of symmetric
positive-definite systems:

<table>
<thead>
<tr>
<th>Band</th>
<th>F04ACF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable band (skyline)</td>
<td>F01MCF and F04MCF</td>
</tr>
</tbody>
</table>
and the following routines allow the iterative solution of symmetric positive-definite systems:

Sparse (incomplete F01MAF and F04MBF Cholesky)
Sparse (conjugate F04MBF gradient)

The latter routine above allows the user to supply a pre-conditioner and also allows the solution of indefinite symmetric systems.

The following routines allow the direct solution of unsymmetric systems:

Band F01LBF and F04LDF
Almost block-diagonal F01LHF and F04LHF
Tridiagonal F01LEF and F04LEF or F04EAF
Sparse F01BRF (and F01BSF) and F04AXF

and the following routine allows the iterative solution of unsymmetric systems:

Sparse F04QAF

The above routine allows the user to supply a pre-conditioner and also allows the solution of least-squares systems.

3.5. Index

Elliptic equations
- equations on rectangular grid (seven-point 2-D molecule) D03EDF
- discretisation on rectangular grid (seven-point 2-D molecule) D03EEF
- Helmholtz’s equation in three dimensions D03FAF
D03 -- Partial Differential Equations

Chapter D03

Partial Differential Equations

D03EDF  Elliptic PDE, solution of finite difference equations by a multigrid technique

D03EEF  Discretize a 2nd order elliptic PDE on a rectangle

D03FAF  Elliptic PDE, Helmholtz equation, 3-D Cartesian coordinates

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

D03 -- Partial Differential Equations

D03EDF -- NAG Foundation Library Routine Document

1. Purpose

D03EDF solves seven-diagonal systems of linear equations which arise from the discretization of an elliptic partial differential equation on a rectangular region. This routine uses a multigrid technique.

2. Specification

SUBROUTINE D03EDF (NGX, NGY, LDA, A, RHS, UB, MAXIT, ACC, US, U, IOUT, NUMIT, IFAIL)

INTEGER NGX, NGY, LDA, MAXIT, IOUT, NUMIT, IFAIL

DOUBLE PRECISION A(LDA,7), RHS(LDA), UB(NGX*NGY), ACC, US(NGX*NGY), U(LDA)

3. Description

D03EDF solves, by multigrid iteration, the seven-point scheme

\[
\begin{align*}
A & \quad \text{u}_{i,j} + A & \text{u}_{i-1,j+1} & \text{u}_{i,j} + A & \text{u}_{i+1,j} \\
3 & \quad 4 & 5 \\
A & \quad A & \quad A & \quad A & \quad A \\
\text{u}_{i,j-1} & \quad \text{u}_{i,j} & \quad \text{u}_{i,j} & \quad \text{u}_{i,j} & \quad \text{u}_{i,j+1}
\end{align*}
\]
\[ A u = f, \]
\[
\begin{array}{cccc}
 1 & 2 \\
 1, j, i, j-1 & i, j, i+1, j-1 & i, j \\
\end{array}
\]
\[ i=1,2,\ldots,n ; j=1,2,\ldots,n , \]
\[ x \ y \]

which arises from the discretization of an elliptic partial differential equation of the form

\[(\alpha)(x, y)u + (\beta)(x, y)u + (\gamma)(x, y)u + (\delta)(x, y)u \]
\[ + (\epsilon)(x, y)u + (\phi)(x, y)u = (\psi)(x, y) \]
\[ y \ y \]

and its boundary conditions, defined on a rectangular region. This we write in matrix form as

\[ Au = f \]

The algorithm is described in separate reports by Wesseling [2], [3] and McCarthy [1].

Systems of linear equations, matching the seven-point stencil defined above, are solved by a multigrid iteration. An initial estimate of the solution must be provided by the user. A zero guess may be supplied if no better approximation is available.

A 'smoother' based on incomplete Crout decomposition is used to eliminate the high frequency components of the error. A restriction operator is then used to map the system on to a sequence of coarser grids. The errors are then smoothed and prolongated (mapped onto successively finer grids). When the finest cycle is reached, the approximation to the solution is corrected. The cycle is repeated for MAXIT iterations or until the required accuracy, ACC, is reached.

D03EDF will automatically determine the number \( l \) of possible coarse grids, 'levels' of the multigrid scheme, for a particular problem. In other words, D03EDF determines the maximum integer \( l \) so that \( n \) and \( n \) can be expressed in the form

\[
\begin{array}{cccc}
 1 & 2 \\
 1, j, i, j-1 & i, j, i+1, j-1 & i, j \\
\end{array}
\]
\[ x \ y \]

\[ n = m^2 + 1, \quad n = n^2 + 1, \quad \text{with } m \geq 2 \text{ and } n \geq 2 . \]
It should be noted that the rate of convergence improves significantly with the number of levels used (see McCarthy [1]), so that \( n \) and \( n \) should be carefully chosen so that \( n - 1 \) and \( n - 1 \) have factors of the form \( 2^l \), with \( l \) as large as possible. For good convergence the integer \( l \) should be at least 2.

D03EDF has been found to be robust in application, but being an iterative method the problem of divergence can arise. For a strictly diagonally dominant matrix \( A \)

\[
\begin{array}{cc}
4 & k \\
ij & \text{--} & \text{ij} \\
|A| & > & |A| \\
\text{--} & k/=4 \\
\end{array}
\]

no such problem is foreseen. The diagonal dominance of \( A \) is not a necessary condition, but should this condition be strongly violated then divergence may occur. The quickest test is to try the routine.

4. References


5. Parameters

1: NGX -- INTEGER Input
   On entry: the number of interior grid points in the \( x \)-direction, \( n \). NGX-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGX \( \geq 3 \).

2: NGY -- INTEGER Input
   On entry: the number of interior grid points in the \( y \)-direction, \( n \). NGY-1 should preferably be divisible by as high a power of 2 as possible. Constraint: NGY \( \geq 3 \).
3: LDA -- INTEGER  
   Input
   On entry: the first dimension of the array A as declared in
   the (sub)program from which D03EDF is called, which must
   also be a lower bound for the dimensions of the arrays RHS,
   US and U. It is always sufficient to set
   LDA>=\((4*(NGX+1)*(NGY+1))/3\), but slightly smaller values may
   be permitted, depending on the values of NGX and NGY. If on
   entry, LDA is too small, an error message gives the minimum
   permitted value. (LDA must be large enough to allow space
   for the coarse-grid approximations).

4: A(LDA,7) -- DOUBLE PRECISION array  
   Input/Output
   k
   On entry: A(i+(j-1)*NGX,k) must be set to A , for i =
   ij
   1,2,...,NGX; j = 1,2,...,NGY and k = 1,2,...,7. On exit: A
   is overwritten.

5: RHS(LDA) -- DOUBLE PRECISION array  
   Input/Output
   On entry: RHS(i+(j-1)*NGX) must be set to f , for i =
   ij
   1,2,...,NGX; j = 1,2,...,NGY. On exit: the first NGX*NGY
   elements are unchanged and the rest of the array is used as
   workspace.

6: UB(NGX*NGY) -- DOUBLE PRECISION array  
   Input/Output
   On entry: UB(i+(j-1)*NGX) must be set to the initial
   estimate for the solution u . On exit: the corresponding
   ij
   component of the residual r=f-Au.

7: MAXIT -- INTEGER  
   Input
   On entry: the maximum permitted number of multigrid
   iterations. If MAXIT = 0, no multigrid iterations are
   performed, but the coarse-grid approximations and incomplete
   Crout decompositions are computed, and may be output if IOUT
   is set accordingly. Constraint: MAXIT >= 0.

8: ACC -- DOUBLE PRECISION  
   Input
   On entry: the required tolerance for convergence of the
   residual 2-norm:

   \[
   \| r \| = \sqrt{\frac{\sum_{k=1}^{NGX*NGY} (r_k)^2}{NGX*NGY}}
   \]

   where r=f-Au and u is the computed solution. Note that the
   norm is not scaled by the number of equations. The routine
will stop after fewer than MAXIT iterations if the residual 2-norm is less than the specified tolerance. (If MAXIT > 0, at least one iteration is always performed.)

If on entry ACC = 0.0, then the machine precision is used as a default value for the tolerance; if ACC > 0.0, but ACC is less than the machine precision, then the routine will stop when the residual 2-norm is less than the machine precision and IFAIL will be set to 4. Constraint: ACC >= 0.0.

9: US(LDA) -- DOUBLE PRECISION array
On exit: the residual 2-norm, stored in element US(1).

10: U(LDA) -- DOUBLE PRECISION array
On exit: the computed solution u is returned in U(i+(j-1)*NGX), for i = 1,2,...,NGX; j = 1,2,...,NGY.

11: IOUT -- INTEGER
On entry: controls the output of printed information to the advisory message unit as returned by X04ABF:
IOUT = 0
  No output.
IOUT = 1
  The solution u, for i = 1,2,...,NGX; j = 1,2,...,NGY.
IOUT = 2
  The residual 2-norm after each iteration, with the reduction factor over the previous iteration.
IOUT = 3
  As for IOUT = 1 and IOUT = 2.
IOUT = 4
  As for IOUT = 3, plus the final residual (as returned in UB).
IOUT = 5
  As for IOUT = 4, plus the initial elements of A and RHS.
IOUT = 6
  As for IOUT = 5, plus the Galerkin coarse grid approximations.
IOUT = 7
  As for IOUT = 6, plus the incomplete Crout
decompositions.

IOUT = 8
As for IOUT = 7, plus the residual after each iteration.
The elements $A(p,k)$, the Galerkin coarse grid approximations
and the incomplete Crout decompositions are output in the format:

\[
\begin{align*}
  Y\text{-index} &= j \\
  X\text{-index} &= i \quad A(p,1) \quad A(p,2) \quad A(p,3) \quad A(p,4) \quad A(p,5) \quad A(p,6) \\
  &\quad A(p,7)
\end{align*}
\]

where $p=i+(j-1)*NGX$, $i = 1,2,\ldots,NGX$ and $j = 1,2,\ldots,NGY$.
The vectors $U(p)$, $UB(p)$, $RHS(p)$ are output in matrix form
with $NGY$ rows and $NGX$ columns. Where $NGX > 10$, the $NGX$
values for a given $j$-value are produced in rows of 10.
Values of $IOUT > 4$ may yield considerable amounts of output.
Constraint: $0 \leq IOUT \leq 8$.

12: NUMIT -- INTEGER
On exit: the number of iterations performed.

13: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry $NGX < 3$,
or $NGY < 3$,
or $LDA$ is too small,
or $ACC < 0.0$,
or $MAXIT < 0$,
or \( \text{IOUT} < 0 \),

or \( \text{IOUT} > 8 \).

\( \text{IFAIL} = 2 \)

MAXIT iterations have been performed with the residual 2-norm decreasing at each iteration but the residual 2-norm has not been reduced to less than the specified tolerance (see ACC). Examine the progress of the iteration by setting \( \text{IOUT} \geq 2 \).

\( \text{IFAIL} = 3 \)

As for \( \text{IFAIL} = 2 \), except that at one or more iterations the residual 2-norm did not decrease. It is likely that the method fails to converge for the given matrix \( A \).

\( \text{IFAIL} = 4 \)

On entry ACC is less than the machine precision. The routine terminated because the residual norm is less than the machine precision.

7. Accuracy

See ACC (Section 5).

8. Further Comments

The rate of convergence of this routine is strongly dependent upon the number of levels, \( l \), in the multigrid scheme, and thus the choice of NGX and NGY is very important. The user is advised to experiment with different values of NGX and NGY to see the effect they have on the rate of convergence; for example, using a value such as NGX = 65 (=2 +1) followed by NGX = 64 (for which \( l = 1 \)).

9. Example

The program solves the elliptic partial differential equation

\[
\begin{align*}
U - (\alpha) U + U &= -4, \\
\alpha &= 1.7
\end{align*}
\]

on the unit square \( 0 \leq x,y \leq 1 \), with boundary conditions

\[
\begin{align*}
\{ x=0, & \ (0 \leq y \leq 1) \} \\
U=0 \ & \text{on } \ y=0, \ (0 \leq x \leq 1) \\
U=1 \ & \text{on } \ x=1, \ 0 \leq y \leq 1. \\
\{ y=1, & \ (0 \leq x \leq 1) \}
\end{align*}
\]

For the equation to be elliptic, \( \alpha \) must be less than 2.
The equation is discretized on a square grid with mesh spacing \( h \) in both directions using the following approximations:

Please see figure in printed Reference Manual

\[
U_{xx} = \frac{1}{h^2} \left( U - 2U + U \right)
\]

\[
U_{yy} = \frac{1}{h^2} \left( U - 2U + U \right)
\]

\[
U_{xy} = \frac{1}{2h} \left( U - U + U - 2U + U - U + U \right)
\]

Thus the following equations are solved:

\[
-(\alpha)u + \left(1 - (\alpha)\right)u
\]

\[
+ \left(1 - (\alpha)\right)u + (-4 + (\alpha))u + \left(1 - (\alpha)\right)u
\]

\[
+ \left(1 - (\alpha)\right)u + (\alpha)u
\]

\[-4h
\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D03EEF discretizes a second order elliptic partial differential equation (PDE) on a rectangular region.

2. Specification

```
SUBROUTINE D03EEF (XMIN, XMAX, YMIN, YMAX, PDEF, BNDY,
  1 NGX, NGY, LDA, A, RHS, SCHEME, IFAIL)
INTEGER NGX, NGY, LDA, IFAIL
DOUBLE PRECISION XMIN, XMAX, YMIN, YMAX, A(LDA,7), RHS(LDA)
CHARACTER*1 SCHEME
EXTERNAL PDEF, BNDY
```

3. Description

D03EEF discretizes a second order linear elliptic partial differential equation of the form

\[
\begin{align*}
(\alpha)(x,y) \frac{dd}{dxdy} U + (\beta)(x,y) \frac{dd}{dx} U + (\gamma)(x,y) \frac{dd}{dy} U + (\delta)(x,y) \frac{dd}{dx} U + (\epsilon)(x,y) \frac{dd}{dy} U + (\phi)(x,y) U &= (\psi)(x,y)
\end{align*}
\]

on a rectangular region

\[
x \leq x \leq x
\]
\[
A \quad B
\]
\[
y \leq y \leq y
\]
\[
A \quad B
\]

subject to boundary conditions of the form

\[
\frac{dd}{dndn} U = a(x,y)U + b(x,y) \Rightarrow c(x,y)
\]
where \( \frac{\partial^2 u}{\partial n^2} \) denotes the outward pointing normal derivative on the boundary. Equation (1) is said to be elliptic if

\[
2 \left\{ \alpha(x,y) \right\} \geq \left\{ \beta(x,y) \right\}
\]

for all points in the rectangular region. The linear equations produced are in a form suitable for passing directly to the multigrid routine D03EDF.

The equation is discretized on a rectangular grid, with \( n \) grid points in the \( x \)-direction and \( n \) grid points in the \( y \)-direction.

The grid spacing used is therefore

\[
h = \frac{(x_B - x_A)}{(n - 1)}
\]

\[
h = \frac{(y_B - y_A)}{(n - 1)}
\]

and the co-ordinates of the grid points \((x_i, y_j)\) are

\[
x = x_A + (i-1)h , \quad i=1,2,\ldots,n
\]

\[
y = y_A + (j-1)h , \quad j=1,2,\ldots,n
\]

At each grid point \((x_i, y_j)\) six neighbouring grid points are used to approximate the partial differential equation, so that the equation is discretized on the following seven-point stencil:

Please see figure in printed Reference Manual

For convenience the approximation \( u_{ij} \) to the exact solution

\( U(x_i, y_j) \) is denoted by \( u_{ij} \), and the neighbouring approximations \( u_{ij}^0 \) are labelled according to points of the compass as shown. Where numerical labels for the seven points are required, these are also shown above.
The following approximations are used for the second derivatives:

\[
\begin{align*}
\frac{\partial^2 U}{\partial x^2} &= \frac{1}{h^2} \left( U_{n+1} - 2U_n + U_{n-1} \right) \\
\frac{\partial^2 U}{\partial y^2} &= \frac{1}{h^2} \left( U_{m+1} - 2U_m + U_{m-1} \right) \\
\frac{\partial^2 U}{\partial x \partial y} &= \frac{1}{h^2} \left( U_{n+1,m} - U_{n-1,m} + U_{n,m+1} - U_{n,m-1} \right)
\end{align*}
\]

Two possible schemes may be used to approximate the first derivatives:

**Central Differences**

\[
\begin{align*}
\frac{\partial U}{\partial x} &= \frac{1}{2h} \left( U_{n+1} - U_n \right) \\
\frac{\partial U}{\partial y} &= \frac{1}{2h} \left( U_{m+1} - U_m \right)
\end{align*}
\]

**Upwind Differences**

\[
\begin{align*}
\frac{\partial U}{\partial x} &= \frac{1}{h} \left( U_{n+1} - U_n \right) \text{ if } \delta(x,y) > 0 \\
\frac{\partial U}{\partial x} &= \frac{1}{h} \left( U_{n} - U_{n-1} \right) \text{ if } \delta(x,y) < 0
\end{align*}
\]
Central differences are more accurate than upwind differences, but upwind differences may lead to a more diagonally dominant matrix for those problems where the coefficients of the first derivatives are significantly larger than the coefficients of the second derivatives.

The approximations used for the first derivatives may be written in a more compact form as follows:

\[
\begin{align*}
\frac{ddU}{dx} &= \frac{1}{2h} ((k - 1)u - 2k u + (k + 1)u) \\
&= \frac{1}{2h} \left( (k - 1)u - 2k u + (k + 1)u \right) \\
\frac{ddU}{dy} &= \frac{1}{2h} ((k - 1)u - 2k u + (k + 1)u) \\
&= \frac{1}{2h} \left( (k - 1)u - 2k u + (k + 1)u \right)
\end{align*}
\]

where \( k = \text{sign} (\Delta) \) and \( k = \text{sign} (\epsilon) \) for upwind differences, and \( k = k = 0 \) for central differences.

At all points in the rectangular domain, including the boundary, the coefficients in the partial differential equation are evaluated by calling the user-supplied subroutine PDEF, and applying the approximations. This leads to a seven-diagonal system of linear equations of the form:

\[
\begin{align*}
A_{ij} u_{i,j-1} + A_{ij} u_{i,j} + A_{ij} u_{i,j+1} &+ A_{i+1,j} u_{i+1,j-1} + A_{i+1,j} u_{i+1,j} + A_{i+1,j} u_{i+1,j+1} = f, &i=1,2,\ldots,n; j=1,2,\ldots,n,
\end{align*}
\]
where the coefficients are given by

1
\begin{align*}
A &= (\beta)(x, y) \quad +(\gamma)(x, y) \quad +(\epsilon)(x, y) \quad -(k - 1) \\
i &j \quad 2h \quad 2h \quad y \quad h
\end{align*}

2
\begin{align*}
A &= -(\beta)(x, y) \\
i &j \quad 2h
\end{align*}

3
\begin{align*}
A &= (\alpha)(x, y) \quad +(\beta)(x, y) \quad +(\delta)(x, y) \quad -(k - 1) \\
i &j \quad 2h \quad 2h \quad x \quad x
\end{align*}

4
\begin{align*}
A &= -(\alpha)(x, y) \quad -(\beta)(x, y) \quad -(\gamma)(x, y) \quad -(\delta)(x, y) \quad -(\epsilon)(x, y) \quad -(\phi)(x, y) \\
i &j \quad h \quad h \quad h \quad h
\end{align*}

5
\begin{align*}
A &= (\alpha)(x, y) \quad +(\beta)(x, y) \quad +(\delta)(x, y) \quad +(k + 1) \\
i &j \quad 2h \quad 2h \quad x \quad x
\end{align*}

6
\begin{align*}
A &= -(\beta)(x, y) \\
i &j \quad 2h
\end{align*}

7
\begin{align*}
A &= (\beta)(x, y) \quad +(\gamma)(x, y) \quad +(\epsilon)(x, y) \quad -(k + 1) \\
i &j \quad 2h \quad 2h \quad y \quad y
\end{align*}

f = (\psi)(x, y)

\begin{align*}
i &j
\end{align*}
These equations then have to be modified to take account of the boundary conditions. These may be Dirichlet (where the solution is given), Neumann (where the derivative of the solution is given), or mixed (where a linear combination of solution and derivative is given).

If the boundary conditions are Dirichlet, there are an infinity of possible equations which may be applied:

\[(\mu)_{ij}u = (\mu)_{ij}f, \ (\mu)_{ij}/=0. \tag{2}\]

If DG3EDF is used to solve the discretized equations, it turns out that the choice of \((\mu)_{ij}\) can have a dramatic effect on the rate of convergence, and the obvious choice \((\mu)=1\) is not the best. Some choices may even cause the multigrid method to fail altogether. In practice it has been found that a value of the same order as the other diagonal elements of the matrix is best, and the following value has been found to work well in practice:

\[(\mu)_{ij} = \min \left( \left\{ \frac{h_i}{h_j} \right\}, A \right). \]

If the boundary conditions are either mixed or Neumann (i.e., \(B \neq 0\) on return from the user-supplied subroutine BNDY), then one of the points in the seven-point stencil lies outside the domain. In this case the normal derivative in the boundary conditions is used to eliminate the 'fictitious' point, \(u\):

\[
\frac{ddU}{dn} = -\left(\frac{u_{outside} - u_{inside}}{2h}\right). \tag{3}
\]

It should be noted that if the boundary conditions are Neumann and \((\phi)(x,y)\neq0\), then there is no unique solution. The routine returns with IFAIL = 5 in this case, and the seven-diagonal matrix is singular.

The four corners are treated separately. The user-supplied subroutine BNDY is called twice, once along each of the edges meeting at the corner. If both boundary conditions at this point are Dirichlet and the prescribed solution values agree, then this value is used in an equation of the form (2). If the prescribed solution is discontinuous at the corner, then the average of the two values is used. If one boundary condition is Dirichlet and the other is mixed, then the value prescribed by the Dirichlet
condition is used in an equation of the form given above. Finally, if both conditions are mixed or Neumann, then two 'fictitious' points are eliminated using two equations of the form (3).

It is possible that equations for which the solution is known at all points on the boundary, have coefficients which are not defined on the boundary. Since this routine calls the user-supplied subroutine PDEF at all points in the domain, including boundary points, arithmetic errors may occur in the user’s routine PDEF which this routine cannot trap. If the user has an equation with Dirichlet boundary conditions (i.e., $B = 0$ at all points on the boundary), but with PDE coefficients which are singular on the boundary, then D03EDF could be called directly only using interior grid points with the user’s own discretization.

After the equations have been set up as described above, they are checked for diagonal dominance. That is to say,

$$
\begin{align*}
4 \quad & - k \\
|A_{ij}| > & |A_{ij}|, \quad i=1,2,\ldots,n; \quad j=1,2,\ldots,n. \\
ij \quad & \quad \quad \quad \quad x \quad y \\
k/=4
\end{align*}
$$

If this condition is not satisfied then the routine returns with IFAIL = 6. The multigrid routine D03EDF may still converge in this case, but if the coefficients of the first derivatives in the partial differential equation are large compared with the coefficients of the second derivative, the user should consider using upwind differences (SCHEME = 'U').

Since this routine is designed primarily for use with D03EDF, this document should be read in conjunction with the document for that routine.

4. References


5. Parameters

1: XMIN -- DOUBLE PRECISION  
   Input  

2: XMAX -- DOUBLE PRECISION  
   Input  

   On entry: the lower and upper $x$ co-ordinates of the rectangular region respectively, $x_A$ and $x_B$. Constraint: $XMIN \leq XMAX$. 


2778

CHAPTER 15. CHAPTER N

< XMAX.

3: YMIN -- DOUBLE PRECISION Input

4: YMAX -- DOUBLE PRECISION Input

On entry: the lower and upper y co-ordinates of the
rectangular region respectively, y and y. Constraint: YMIN
A < YMAX.

5: PDEF -- SUBROUTINE, supplied by the user.

External Procedure

PDEF must evaluate the functions (alpha)(x,y), (beta)(x,y),
(gamma)(x,y), (delta)(x,y), (epsilon)(x,y), (phi)(x,y) and
(psi)(x,y) which define the equation at a general point
(x,y).

Its specification is:

SUBROUTINE PDEF (X, Y, ALPHA, BETA, GAMMA,
1 DELTA, EPSLON, PHI, PSI)

DOUBLE PRECISION X, Y, ALPHA, BETA, GAMMA, DELTA,
1 EPSLON, PHI, PSI

1: X -- DOUBLE PRECISION Input

2: Y -- DOUBLE PRECISION Input

On entry: the x and y co-ordinates of the point at
which the coefficients of the partial differential
equation are to be evaluated. 8

3: ALPHA -- DOUBLE PRECISION Output

4: BETA -- DOUBLE PRECISION Output

5: GAMMA -- DOUBLE PRECISION Output

6: DELTA -- DOUBLE PRECISION Output

7: EPSLON -- DOUBLE PRECISION Output

8: PHI -- DOUBLE PRECISION Output

9: PSI -- DOUBLE PRECISION Output

On exit: ALPHA, BETA, GAMMA, DELTA, EPSLON, PHI and PSI
must be set to the values of (alpha)(x,y), (beta)(x,y),
(gamma)(x,y), (delta)(x,y), (epsilon)(x,y), (phi)(x,y)
and (psi)(x,y) respectively at the point specified by X
and Y.

PDEF must be declared as EXTERNAL in the (sub)program
from which D03EEF is called. Parameters denoted as
Input must not be changed by this procedure.

6: BNDY -- SUBROUTINE, supplied by the user.

External Procedure
BNDY must evaluate the functions $a(x,y)$, $b(x,y)$, and $c(x,y)$
involved in the boundary conditions.

Its specification is:

```fortran
SUBROUTINE BNDY (X, Y, A, B, C, IBND)
INTEGER IBND
DOUBLE PRECISION X, Y, A, B, C
```

1: X -- DOUBLE PRECISION
   Input

2: Y -- DOUBLE PRECISION
   Input
   On entry: the x and y co-ordinates of the point at
   which the boundary conditions are to be evaluated.

3: A -- DOUBLE PRECISION
   Output

4: B -- DOUBLE PRECISION
   Output

5: C -- DOUBLE PRECISION
   Output
   On exit: A, B and C must be set to the values of the
   functions appearing in the boundary conditions.

6: IBND -- INTEGER
   Input
   On entry: specifies on which boundary the point (X,Y)
   lies. IBND = 0, 1, 2 or 3 according as the point lies
   on the bottom, right, top or left boundary.

BNDY must be declared as EXTERNAL in the (sub)program
from which D03EEF is called. Parameters denoted as
Input must not be changed by this procedure.

7: NGX -- INTEGER
   Input

8: NGY -- INTEGER
   Input
   On entry: the number of interior grid points in the x- and y
   -directions respectively, $n_x$ and $n_y$. If the seven-diagonal
   $x$ $y$
equations are to be solved by D03EDF, then NGX-1 and NGY-1
   should preferably be divisible by as high a power of 2 as
   possible. Constraint: NGX $\geq 3$, NGY $\geq 3$.

9: LDA -- INTEGER
   Input
   On entry: the first dimension of the array A as declared in the
   (sub)program from which D03EEF is called.
Constraint: if only the seven-diagonal equations are required, then \( LDA \geq NGX \times NGY \). If a call to this routine is to be followed by a call to D03EDF to solve the seven-diagonal linear equations, \( LDA \geq (4 \times (NGX + 1) \times (NGY + 1))/3 \).

Note: this routine only checks the former condition. D03EDF, if called, will check the latter condition.

10: \( A(LDA, 7) \) -- DOUBLE PRECISION array
On exit: \( A(i, j) \), for \( i = 1, 2, \ldots, NGX \times NGY \); \( j = 1, 2, \ldots, 7 \), contains the seven-diagonal linear equations produced by the discretization described above. If \( LDA > NGX \times NGY \), the remaining elements are not referenced by the routine, but if \( LDA \geq (4 \times (NGX + 1) \times (NGY + 1))/3 \) then the array \( A \) can be passed directly to D03EDF, where these elements are used as workspace.

11: \( RHS(LDA) \) -- DOUBLE PRECISION array
On exit: the first \( NGX \times NGY \) elements contain the right-hand sides of the seven-diagonal linear equations produced by the discretization described above. If \( LDA > NGX \times NGY \), the remaining elements are not referenced by the routine, but if \( LDA \geq (4 \times (NGY + 1) \times (NGY + 1))/3 \) then the array \( RHS \) can be passed directly to D03EDF, where these elements are used as workspace.

12: \( SCHEME \) -- CHARACTER*1
On entry: the type of approximation to be used for the first derivatives which occur in the partial differential equation.

If \( SCHEME = 'C' \), then central differences are used.

If \( SCHEME = 'U' \), then upwind differences are used.

Constraint: \( SCHEME = 'C' \) or \( 'U' \).

Note: generally speaking, if at least one of the coefficients multiplying the first derivatives (DELTA or EPSLON as returned by PDEF) are large compared with the coefficients multiplying the second derivatives, then upwind differences may be more appropriate. Upwind differences are less accurate than central differences, but may result in more rapid convergence for strongly convective equations. The easiest test is to try both schemes.

13: \( IFAIL \) -- INTEGER
On entry: \( IFAIL \) must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).

For this routine, because the values of output parameters may be useful even if IFAIL /= 0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry XMIN >= XMAX,
or YMIN >= YMAX,
or NGX < 3,
or NGY < 3,
or LDA < NGX*NGY,
or SCHEME is not one of 'C' or 'U'.

IFAIL= 2
At some point on the boundary there is a derivative in the boundary conditions (B /= 0 on return from a BNDY) and there is a non-zero coefficient of the mixed derivative (BETA /= 0 on return from PDEF).

IFAIL= 3
A null boundary has been specified, i.e., at some point both A and B are zero on return from a call to BNDY.

IFAIL= 4
The equation is not elliptic, i.e., 4*ALPHA*GAMMA<BETA after a call to PDEF. The discretization has been completed, but the convergence of D03EDF cannot be guaranteed.

IFAIL= 5
The boundary conditions are purely Neumann (only the derivative is specified) and there is, in general, no unique
solution.

IFAIL= 6

The equations were not diagonally dominant. (See Section 3).

7. Accuracy

Not applicable.

8. Further Comments

If this routine is used as a pre-processor to the multigrid routine D03EDF it should be noted that the rate of convergence of that routine is strongly dependent upon the number of levels in the multigrid scheme, and thus the choice of NGX and NGY is very important.

9. Example

The program solves the elliptic partial differential equation

\[
\begin{align*}
\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} + 50 \left( \frac{\partial U}{\partial x} + \frac{\partial U}{\partial y} \right) &= f(x, y) \\
\frac{\partial U}{\partial x} \bigg|_{x=0} \text{ and } \frac{\partial U}{\partial y} \bigg|_{y=0} &= \text{given on } x=0 \text{ and } y=0, \\
\frac{\partial U}{\partial n} \bigg|_{x=1} \text{ and } \frac{\partial U}{\partial n} \bigg|_{y=1} &= \text{given on } x=1 \text{ and } y=1.
\end{align*}
\]

on the unit square 0<x, y<1, with boundary conditions

\[
\begin{align*}
\frac{\partial U}{\partial x} \bigg|_{x=0} \text{ and } \frac{\partial U}{\partial y} \bigg|_{y=0} &= \text{given on } x=0 \text{ and } y=0, \\
\frac{\partial U}{\partial n} \bigg|_{x=1} \text{ and } \frac{\partial U}{\partial n} \bigg|_{y=1} &= \text{given on } x=1 \text{ and } y=1.
\end{align*}
\]

The function f(x,y) and the exact form of the boundary conditions are derived from the exact solution U(x,y)=sinxsiny.

The equation is first solved using central differences. Since the coefficients of the first derivatives are large, the linear equations are not diagonally dominated, and convergence is slow. The equation is solved a second time with upwind differences, showing that convergence is more rapid, but the solution is less accurate.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
D03 -- Partial Differential Equations

D03FAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

D03FAF solves the Helmholtz equation in Cartesian co-ordinates in three dimensions using the standard seven-point finite difference approximation. This routine is designed to be particularly efficient on vector processors.

2. Specification

```fortran
SUBROUTINE D03FAF (XS, XF, L, LBDCND, BDXS, BDXF, YS, YF,
1 M, MBDCND, BDYS, BDYF, ZS, ZF, N,
2 NBDCND, BDZS, BDZF, LAMBDA, LDIMF,
3 MDIMF, F, PERTRB, W, LWRK, IFAIL)
```

3. Description

D03FAF solves the three-dimensional Helmholtz equation in cartesian co-ordinates:

\[
\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + (\lambda)u = f(x, y, z)
\]

This subroutine forms the system of linear equations resulting from the standard seven-point finite difference equations, and then solves the system using a method based on the fast Fourier transform (FFT) described by Swarztrauber [1]. This subroutine is based on the routine HW3CRT from FISHPACK (see Swarztrauber and Sweet [2]).

More precisely, the routine replaces all the second derivatives by second-order central difference approximations, resulting in a
block tridiagonal system of linear equations. The equations are modified to allow for the prescribed boundary conditions. Either the solution or the derivative of the solution may be specified on any of the boundaries, or the solution may be specified to be periodic in any of the three dimensions. By taking the discrete Fourier transform in the \(x\)- and \(y\)-directions, the equations are reduced to sets of tridiagonal systems of equations. The Fourier transforms required are computed using the multiple FFT routines found in Chapter C06 of the NAG Fortran Library.

4. References


5. Parameters

1: \(XS\) -- DOUBLE PRECISION
   On entry: the lower bound of the range of \(x\), i.e., \(XS \leq x \leq XF\). Constraint: \(XS < XF\).

2: \(XF\) -- DOUBLE PRECISION
   On entry: the upper bound of the range of \(x\), i.e., \(XS \leq x \leq XF\). Constraint: \(XS < XF\).

3: \(L\) -- INTEGER
   On entry: the number of panels into which the interval \((XS,XF)\) is subdivided. Hence, there will be \(L+1\) grid points in the \(x\)-direction given by \(x = XS + (i-1) \cdot (\text{delta}x), \) for \(i = 1,2,\ldots,L+1\), where \((\text{delta}x) = (XF-XS) / L\) is the panel width. Constraint: \(L \geq 5\).

4: \(LBDCND\) -- INTEGER
   On entry: indicates the type of boundary conditions at \(x = XS\) and \(x = XF\).
   \(LBDCND = 0\)
   if the solution is periodic in \(x\), i.e., \(u(XS,y,z) = u(XF,y,z)\).
   \(LBDCND = 1\)
   if the solution is specified at \(x = XS\) and \(x = XF\).
   \(LBDCND = 2\)
   if the solution is specified at \(x = XS\) and the
derivative of the solution with respect to \( x \) is specified at \( x = X_F \).

**LBDCND = 3**
if the derivative of the solution with respect to \( x \) is specified at \( x = X_S \) and \( x = X_F \).

**LBDCND = 4**
if the derivative of the solution with respect to \( x \) is specified at \( x = X_S \) and the solution is specified at \( x = X_F \).
Constraint: \( 0 \leq LBDCND \leq 4 \).

5: \( BDXS(MDIMF,N+1) \) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to \( x \) at \( x = X_S \). When \( LBDCND = 3 \) or 4, \( BDXS(j,k) = u(X_S,y,z), \) for \( j=1,2,...,M+1; k=1,2,...,N+1. \)
When \( LBDCND \) has any other value, \( BDXS \) is not referenced.

6: \( BDXF(MDIMF,N+1) \) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to \( x \) at \( x = X_F \). When \( LBDCND = 2 \) or 3, \( BDXF(j,k) = u(X_F,y,z), \) for \( j=1,2,...,M+1; k=1,2,...,N+1. \)
When \( LBDCND \) has any other value, \( BDXF \) is not referenced.

7: \( YS \) -- DOUBLE PRECISION Input
On entry: the lower bound of the range of \( y \), i.e., \( YS \leq y \leq YF \). Constraint: \( YS < YF \).

8: \( YF \) -- DOUBLE PRECISION Input
On entry: the upper bound of the range of \( y \), i.e., \( YS \leq y \leq YF \). Constraint: \( YS < YF \).

9: \( M \) -- INTEGER Input
On entry: the number of panels into which the interval \( (YS,YF) \) is subdivided. Hence, there will be \( M+1 \) grid points in the \( y \)-direction given by \( y = YS + (j-1)*(delta)y \) for \( j=1,2,...,M+1, \) where \( (delta)y = (YF-YS)/M \) is the panel width. Constraint: \( M \geq 5 \).

10: \( MBDCND \) -- INTEGER Input
On entry: indicates the type of boundary conditions at \( y = YS \) and \( y = YF \).
\( MBDCND = 0 \)
if the solution is periodic in \( y \), i.e.,
\[ u(x, Y_F, z) = u(x, Y_S, z). \]

**MBDCND = 1**
- if the solution is specified at \( y = Y_S \) and \( y = Y_F \).

**MBDCND = 2**
- if the solution is specified at \( y = Y_S \) and the derivative of the solution with respect to \( y \) is specified at \( y = Y_F \).

**MBDCND = 3**
- if the derivative of the solution with respect to \( y \) is specified at \( y = Y_S \) and \( y = Y_F \).

**MBDCND = 4**
- if the derivative of the solution with respect to \( y \) is specified at \( y = Y_S \) and the solution is specified at \( y = Y_F \).

**Constraint:** \( 0 \leq MBDCND \leq 4 \).

**11: BDYS(LDIMF,N+1) -- DOUBLE PRECISION array**
- Input
  - On entry: the values of the derivative of the solution with respect to \( y \) at \( y = Y_S \). When \( MBDCND = 3 \) or \( 4 \), \( BDYS(i,k) = u(x, y, z) \), for \( i = 1, 2, \ldots, L + 1; k = 1, 2, \ldots, N + 1 \).
  - When \( MBDCND \) has any other value, \( BDYS \) is not referenced.

**12: BDYF(LDIMF,N+1) -- DOUBLE PRECISION array**
- Input
  - On entry: the values of the derivative of the solution with respect to \( y \) at \( y = Y_F \). When \( MBDCND = 2 \) or \( 3 \), \( BDYF(i,k) = u(x, Y_F, z) \), for \( i = 1, 2, \ldots, L + 1; k = 1, 2, \ldots, N + 1 \).
  - When \( MBDCND \) has any other value, \( BDYF \) is not referenced.

**13: ZS -- DOUBLE PRECISION**
- Input
  - On entry: the lower bound of the range of \( z \), i.e., \( ZS \leq z \leq ZF \). Constraint: \( ZS < ZF \).

**14: ZF -- DOUBLE PRECISION**
- Input
  - On entry: the upper bound of the range of \( z \), i.e., \( ZS \leq z \leq ZF \). Constraint: \( ZS < ZF \).

**15: N -- INTEGER**
- Input
  - On entry: the number of panels into which the interval \((ZS, ZF)\) is subdivided. Hence, there will be \( N + 1 \) grid points in the \( z \)-direction given by \( z = ZS + (k-1) \times (\text{delta}z) \), for \( k = 1, 2, \ldots, N + 1 \), where \( (\text{delta}z) = (ZF - ZS)/N \) is the panel width.
Constraint: \( N \geq 5 \).

16: \( \text{NBDCND} \) -- INTEGER Input
On entry: specifies the type of boundary conditions at \( z = ZS \) and \( z = ZF \).
\( \text{NBDCND} = 0 \)
  if the solution is periodic in \( z \), i.e., \( u(x,y,ZF) = u(x,y,ZS) \).
\( \text{NBDCND} = 1 \)
  if the solution is specified at \( z = ZS \) and \( z = ZF \).
\( \text{NBDCND} = 2 \)
  if the solution is specified at \( z = ZS \) and the derivative of the solution with respect to \( z \) is specified at \( z = ZF \).
\( \text{NBDCND} = 3 \)
  if the derivative of the solution with respect to \( z \) is specified at \( z = ZS \) and \( z = ZF \).
\( \text{NBDCND} = 4 \)
  if the derivative of the solution with respect to \( z \) is specified at \( z = ZS \) and the solution is specified at \( z = ZF \).
Constraint: \( 0 \leq \text{NBDCND} \leq 4 \).

17: \( \text{BDZS}(\text{LDIMF},M+1) \) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to \( z \) at \( z = ZS \). When \( \text{NBDCND} = 3 \) or \( 4 \), \( \text{BDZS} \)
\( (i,j)=u(x,y,ZS)=u(x,y,z) \), for \( i=1,2,\ldots,L+1; \)
\( z \) \( i \ j \)
\( j=1,2,\ldots,M+1. \)

When \( \text{NBDCND} \) has any other value, \( \text{BDZS} \) is not referenced.

18: \( \text{BDZF}(\text{LDIMF},M+1) \) -- DOUBLE PRECISION array Input
On entry: the values of the derivative of the solution with respect to \( z \) at \( z = ZF \). When \( \text{NBDCND} = 2 \) or \( 3 \), \( \text{BDZF} \)
\( (i,j)=u(x,y,ZF)=u(x,y,z) \), for \( i=1,2,\ldots,L+1; \)
\( z \) \( i \ j \)
\( j=1,2,\ldots,M+1. \)

When \( \text{NBDCND} \) has any other value, \( \text{BDZF} \) is not referenced.

19: \( \text{LAMBDAD} \) -- DOUBLE PRECISION Input
On entry: the constant \( \text{(lambda)} \) in the Helmholtz equation.
For certain positive values of \( \text{(lambda)} \) a solution to the differential equation may not exist, and close to these values the solution of the discretized problem will be
extremely ill-conditioned. If (\(\lambda\)) > 0, then D03FAF will set IFAIL to 3, but will still attempt to find a solution. However, since in general the values of (\(\lambda\)) for which no solution exists cannot be predicted a priori, the user is advised to treat any results computed with (\(\lambda\)) > 0 with great caution.

20: LDIMF -- INTEGER
On entry:
the first dimension of the arrays F, BDYS, BDYF, BDZS and BDZF as declared in the (sub)program from which D03FAF is called.
Constraint: LDIMF \(\geq\) L + 1.

21: MDIMF -- INTEGER
On entry:
the second dimension of the array F and
the first dimension of the arrays BDXS and BDXF as declared in the (sub)program from which D03FAF is called.
Constraint: MDIMF \(\geq\) M + 1.

22: F(LDIMF,MDIMF,N+1) -- DOUBLE PRECISION array
On entry:
the values of the right-side of the Helmholtz equation and boundary values (if any).
\[
F(i,j,k) = f(x_{i,j,k}) \quad (i=2,3,...,L, \quad j=2,3,...,M \quad \text{and} \quad k=2,3,...,N).
\]

On the boundaries F is defined by
LBDCND F(1,j,k) = F(L+1,j,k)  \(\forall\) \(i\) = 1, 2, ..., \(M\) \(\forall\) \(j\) = 1, 2, ..., \(N\) \(\forall\) \(k\) = 1, 2, ..., \(N\) + 1

MBDCND F(i,1,k) = F(i,M+1,k)
\begin{align*}
0 & \quad f(x,y,z) f(x,y,z) \\
& \quad i \ k \ k \ k
1 & \quad u(x,y,z) u(x,y,z) \\
& \quad i \ k \ k \ k
2 & \quad u(x,y,z) f(x,y,z) \quad i=1,2,\ldots,L+1 \\
& \quad i \ k \ k \ k
3 & \quad f(x,y,z) f(x,y,z) \quad k=1,2,\ldots,N+1 \\
& \quad i \ k \ k \ k
4 & \quad f(x,y,z) u(x,y,z) \\
& \quad i \ k \ k \ k
\end{align*}

NBDCND \quad F(i,j,1) \quad F(i,j,N+1)

\begin{align*}
0 & \quad f(x,y,z) f(x,y,z) \\
& \quad i \ j \ j \ j
1 & \quad u(x,y,z) u(x,y,z) \\
& \quad i \ j \ j \ j
2 & \quad u(x,y,z) f(x,y,z) \quad i=1,2,\ldots,L+1 \\
& \quad i \ j \ j \ j
3 & \quad f(x,y,z) f(x,y,z) \quad j=1,2,\ldots,M+1 \\
& \quad i \ j \ j \ j
4 & \quad f(x,y,z) u(x,y,z) \\
& \quad i \ j \ j \ j
\end{align*}

Note: if the table calls for both the solution \( u \) and the right-hand side \( f \) on a boundary, then the solution must be specified. On exit: \( F \) contains the solution \( u(i,j,k) \) of the finite difference approximation for the grid point \( (x,y,z) \) for \( i=1,2,\ldots,L+1, \ j=1,2,\ldots,M+1 \) and \( k=1,2,\ldots,N+1 \).

23: \textbf{PERTRB} -- DOUBLE PRECISION \quad \textbf{Output}

On exit: \textbf{PERTRB} = 0, unless a solution to Poisson's equation \((\lambda) = 0\) is required with a combination of periodic or derivative boundary conditions \((\text{LBDCND}, \text{MBDCND} \text{ and } \text{NBDCND} = 0 \text{ or } 3)\). In this case a solution may not exist. \textbf{PERTRB} is a constant, calculated and subtracted from the array \( F \), which ensures that a solution exists. \texttt{D03FAF} then computes this solution, which is a least-squares solution to
the original approximation. This solution is not unique and is unnormalised. The value of PERTRB should be small compared to the right-hand side F, otherwise a solution has been obtained to an essentially different problem. This comparison should always be made to insure that a meaningful solution has been obtained.

24: W(LWRK) -- DOUBLE PRECISION array Workspace

25: LWRK -- INTEGER Input
On entry: the dimension of the array W as declared in the (sub)program from which D03FAF is called.
LRWK>=2*(N+1)*max(L,M)+3*L+3*M+4*N+6 is an upper bound on the required size of W. If LWRK is too small, the routine exits with IFAIL = 2, and if on entry IFAIL = 0 or IFAIL = -1, a message is output giving the exact value of LWRK required to solve the current problem.

26: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. Users who are unfamiliar with this parameter should refer to the Essential Introduction for details.
On exit: IFAIL = 0 unless the routine detects an error or gives a warning (see Section 6).
For this routine, because the values of output parameters may be useful even if IFAIL /=0 on exit, users are recommended to set IFAIL to -1 before entry. It is then essential to test the value of IFAIL on exit.

6. Error Indicators and Warnings

Errors or warnings specified by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry XS >= XF,
or L < 5,
or LBDCND < 0,
or LBDCND > 4,
or YS >= YF,
or $M < 5$,
or $MBDCND < 0$,
or $MBDCND > 4$,
or $ZS \geq ZF$,
or $N < 5$,
or $NBDCND < 0$,
or $NBDCND > 4$,
or $LDIMF < L + 1 > 0$,
or $MDIMF < M + 1$.

$IFAIL= 2$
On entry $LWRK$ is too small.

$IFAIL= 3$
On entry $(\lambda) > 0$.

7. Accuracy

None.

8. Further Comments

The execution time is roughly proportional to $L \times M \times N \times (\log L \times \log M + 5)$, but also depends on input parameters $LBDCND$ and $MBDCND$.

9. Example

The example solves the Helmholtz equation

\[ \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + (\lambda)u = f(x,y,z) \]

\[ (\pi) \]

for $(x,y,z)$ is in $[0,1] \times [0,2\pi] \times [0, \pi]$ where $(\lambda) = -2$, and

\[ f(x,y,z) \] is derived from the exact solution
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
++ differential equations.

NagPartialDifferentialEquationsPackage():Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==> with
d03edf : (Integer,Integer,Integer,Integer,Integer,DoubleFloat,Integer,Matrix DoubleFloat,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
++ d03edf(ngx,ngy,lda,maxit,acc,iout,a,rhs,ub,ifail)
++ solves seven-diagonal systems of linear equations which
++ arise from the discretization of an elliptic partial differential
++ equation on a rectangular region. This routine uses a multigrid
++ technique.
++ See \downlink{Manual Page}{manpageXXd03edf}.

d03eef : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,String,Integer,Union(fn:FileName,fp:Asp73(PDEF)),Union(fn:FileName,fp:Asp74(BNDY))) -> Result
++ d03eef(xmin,xmax,ymin,ymax,ngx,ngy,lda,scheme,ifail,pdef,bndy)
++ discretizes a second order elliptic partial differential
++ equation (PDE) on a rectangular region.
++ See \downlink{Manual Page}{manpageXXd03eef}.

d03faf : (DoubleFloat,DoubleFloat,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Integer,Integer,ThreeDimensionalMatrix DoubleFloat,Integer) -> Result
++ d03faf(xs,xf,l,lbdcnd,bdxs,bdxf,ys,yf,m,mbdcnd,bdys,bdyf,zs,
++ zf,n,mbdcnd,bdzs,bdzf,lambda,ldimf,mdimf,lwrk,f,ifail)
++ solves the Helmholtz equation in Cartesian co-ordinates in
++ three dimensions using the standard seven-point finite difference
++ approximation. This routine is designed to be particularly
++ efficient on vector processors.
++ See \downlink{Manual Page}{manpageXXd03faf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(ThreeDimensionalMatrix DoubleFloat)
import FortranPackage
import Union(fn:FileName,fp:Asp73(PDEF))
import Union(fn:FileName,fp:Asp74(BNDY))

d03edf(ngxArg:Integer,ngyArg:Integer,ldaArg:Integer,_
maxitArg:Integer,accArg:DoubleFloat,ioutArg:Integer,_
aArg:Matrix DoubleFloat,rhsArg:Matrix DoubleFloat,_
ubArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"d03edf",_
)$Lisp,_
["us":S,"u":S,"numit":S]$Lisp,$Lisp,_
["double":S,"acc":S,["us":S,"ida":S]$Lisp,_
["rhs":S,"ida":S]$Lisp,$Lisp,_
[(ngxArg::Any,ngyArg::Any,ldaArg::Any,maxitArg::Any,accArg::Any,_
 ioutArg::Any,ifailArg::Any,aArg::Any,rhsArg::Any,ubArg::Any ])_
@List Any$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result


d03eef(xminArg:DoubleFloat,xmaxArg:DoubleFloat,yminArg:DoubleFloat,_
ymaxArg:DoubleFloat,ngxArg:Integer,ngyArg:Integer,_
ldaArg:Integer,schemeArg:String,ifailArg:Integer,_
pdefArg:Union(fn:FileName,fp:Asp73(PDEF)),bndyArg:Union(fn:FileName,_
fp:Asp74(BNDY))): Result ==
pushFortranOutputStack(pdefFilename := aspFilename "pdef")$FOP
if pdefArg case fn
  then outputAsFortran(pdefArg.fn)
  else outputAsFortran(pdefArg.fp)
popFortranOutputStack()$FOP
pushFortranOutputStack(bndyFilename := aspFilename "bndy")$FOP
if bndyArg case fn
  then outputAsFortran(bndyArg.fn)
  else outputAsFortran(bndyArg.fp)
popFortranOutputStack()$FOP
[(invokeNagman([pdefFilename,bndyFilename]$Lisp,_
"d03eef",_
"a"::S,"lda"::S,7$Lisp]$Lisp,_
["character"::S,"scheme"::S]$Lisp]$Lisp,_
["a"::S,"rhs"::S,"ifail"::S]$Lisp,_
[("xminArg::Any,"xmaxArg::Any,"yminArg::Any,"ymaxArg::Any,_
"ngxArg::Any,"ldaArg::Any,"schemeArg::Any,"ifailArg::Any ])_
@List Any][Lisp]@Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

d03faf(xsArg:DoubleFloat,xfArg:DoubleFloat,lArg:Integer,_
ldcndArg:Integer,bdxsArg:Matrix DoubleFloat,_
bdxfArg:Matrix DoubleFloat,ysArg:DoubleFloat,yfArg:DoubleFloat,mArg:Integer,_
mbcndArg:Integer,bdyysArg:Matrix DoubleFloat,_
bdfyArg:Matrix DoubleFloat,zsArg:DoubleFloat,zyfArg:DoubleFloat,nArg:Integer,_
nbdcndArg:Integer,bdysArg:Matrix DoubleFloat,_
bdfzArg:Matrix DoubleFloat,lwrkArg:Integer,ldimfArg:Integer,mdimfArg:Integer,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
["pertrb"::S,"w"::S]$Lisp,,
["pertrb"::S,"f"::S,"ifail"::S]$Lisp,,
@List Any$List$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

package NAGC02 NagPolynomialRootsPackage

--- NagPolynomialRootsPackage.input ---

)set break resume
)sys rm -f NagPolynomialRootsPackage.output
)spool NagPolynomialRootsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 13
)show NagPolynomialRootsPackage
--R
--R NagPolynomialRootsPackage is a package constructor
--R Abbreviation for NagPolynomialRootsPackage is NAGC02
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGC02
--R
--R----------------------------------- Operations -----------------------------------
--R c02aff : (Matrix(DoubleFloat),Integer,Boolean,Integer) -> Result
--R c02agf : (Matrix(DoubleFloat),Integer,Boolean,Integer) -> Result
--R
--E 1

)clear all

--S 2 of 13
showArrayValues true
--R
--R
--R (1) true
--E 2

--S 3 of 13
showScalarValues true
--R
--R (2) true
--E 3

--S 4 of 13
a: Matrix SF:=
[[5.0 ,30.0 , -0.2 ,50.0 , -2.0 ,10.0 ],
 [6.0 ,20.0 , -6.0 ,100000.0 , -40.0 ,1.0 ]]  
--R
--R
--R +5. 30. -0.19999999999999998 50. -2. 10. +
--R (3) |  
--E 4

--S 5 of 13
n:=ncols(a)-1
--R
--R (4) 5
--E 5

--S 6 of 13
scale:=true
--R
--R (5) true
--E 6

--S 7 of 13
result:=c02aff(a,n, scale,-1)
--E 7

)clear all

--S 8 of 13
showArrayValues true
--R
--R
This package uses the NAG Library to compute the zeros of a polynomial.
with real or complex coefficients.

Let \( f(z) \) be a polynomial of degree \( n \) with complex coefficients \( a \):

\[
f(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_2 z^2 + a_1 z + a_0, \quad a_0 \neq 0.
\]

A complex number \( z \) is called a zero of \( f(z) \) (or equivalently a root of the equation \( f(z) = 0 \)), if:

\[
f(z) = 0.
\]

If \( z \) is a zero, then \( f(z) \) can be divided by a factor \((z - z_1)\):

\[
f(z) = (z - z_1) f(z)
\]

(1)

where \( f(z) \) is a polynomial of degree \( n-1 \). By the Fundamental Theorem of Algebra, a polynomial \( f(z) \) always has a zero, and so the process of dividing out factors \((z - z_i)\) can be continued until we have a complete factorization of \( f(z) \):

\[
f(z) = (z - z_1)(z - z_2) \ldots (z - z_n).
\]

Here the complex numbers \( z_1, z_2, \ldots, z_n \) are the zeros of \( f(z) \); they may not all be distinct, so it is sometimes more convenient to
write:

\[ f(z) = a_1(z-z_1)(z-z_2)\ldots(z-z_k), \quad k\leq n, \]

with distinct zeros \( z_1, z_2, \ldots, z_k \) and multiplicities \( m_i \geq 1 \). If \( m_i = 1 \), \( z_i \) is called a single zero; if \( m_i > 1 \), \( z_i \) is called a multiple or repeated zero; a multiple zero is also a zero of the derivative of \( f(z) \).

If the coefficients of \( f(z) \) are all real, then the zeros of \( f(z) \) are either real or else occur as pairs of conjugate complex numbers \( x+iy \) and \( x-iy \). A pair of complex conjugate zeros are the zeros of a quadratic factor of \( f(z) \), \( (z^2+rz+s) \), with real coefficients \( r \) and \( s \).

Mathematicians are accustomed to thinking of polynomials as pleasantly simple functions to work with. However the problem of numerically computing the zeros of an arbitrary polynomial is far from simple. A great variety of algorithms have been proposed, of which a number have been widely used in practice; for a fairly comprehensive survey, see Householder [1]. All general algorithms are iterative. Most converge to one zero at a time; the corresponding factor can then be divided out as in equation (1) above — this process is called deflation or, loosely, dividing out the zero — and the algorithm can be applied again to the polynomial \( f(z) \). A pair of complex conjugate zeros can be divided out together — this corresponds to dividing \( f(z) \) by a quadratic factor.

Whatever the theoretical basis of the algorithm, a number of practical problems arise: for a thorough discussion of some of them see Peters and Wilkinson [2] and Wilkinson [3]. The most elementary point is that, even if \( z_1 \) is mathematically an exact zero of \( f(z) \), because of the fundamental limitations of computer arithmetic the computed value of \( f(z) \) will not necessarily be exactly 0.0. In practice there is usually a small region of values of \( z \) about the exact zero at which the computed value of \( f(z) \) becomes swamped by rounding errors. Moreover in many algorithms this inaccuracy in the computed value of \( f(z) \) results in a similar inaccuracy in the computed step from one iterate to the next. This limits the precision with which any zero can be
computed. Deflation is another potential cause of trouble, since, in the notation of equation (1), the computed coefficients of \( f(z) \) will not be completely accurate, especially if \( z \) is not an exact zero of \( f(z) \); so the zeros of the computed \( f(z) \) will deviate from the zeros of \( f(z) \).

A zero is called ill-conditioned if it is sensitive to small changes in the coefficients of the polynomial. An ill-conditioned zero is likewise sensitive to the computational inaccuracies just mentioned. Conversely a zero is called well-conditioned if it is comparatively insensitive to such perturbations. Roughly speaking a zero which is well separated from other zeros is well-conditioned, while zeros which are close together are ill-conditioned, but in talking about 'closeness' the decisive factor is not the absolute distance between neighbouring zeros but their ratio: if the ratio is close to 1 the zeros are ill-conditioned. In particular, multiple zeros are ill-conditioned. A multiple zero is usually split into a cluster of zeros by perturbations in the polynomial or computational inaccuracies.

2.1. References


3. Recommendations on Choice and Use of Routines

3.1. Discussion

Two routines are available: C02AFF for polynomials with complex coefficients and C02AGF for polynomials with real coefficients.

C02AFF and C02AGF both use a variant of Laguerre's Method due to BT Smith to calculate each zero until the degree of the deflated polynomial is less than 3, whereupon the remaining zeros are obtained using the 'standard' closed formulae for a quadratic or linear equation.

The accuracy of the roots will depend on how ill-conditioned they are. Peters and Wilkinson [2] describe techniques for estimating the errors in the zeros after they have been computed.
3.2. Index

Zeros of a complex polynomial C02AFF
Zeros of a real polynomial C02AGF

C02 -- Zeros of Polynomials
Chapter C02

Zeros of Polynomials

C02AFF All zeros of complex polynomial, modified Laguerre method
C02AGF All zeros of real polynomial, modified Laguerre method

1. Purpose

C02AFF finds all the roots of a complex polynomial equation, using a variant of Laguerre’s Method.

2. Specification

SUBROUTINE C02AFF (A, N, SCALE, Z, W, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION A(2,N+1), Z(2,N), W(4*(N+1))
LOGICAL SCALE

3. Description

The routine attempts to find all the roots of the nth degree complex polynomial equation

\[ P(z) = a_n z^n + a_{n-1} z^{n-1} + \ldots + a_1 z + a_0 = 0. \]

The roots are located using a modified form of Laguerre’s Method,
originally proposed by Smith [2].

The method of Laguerre [3] can be described by the iterative scheme

\[
L(z) = z - \frac{-n \cdot P(z)}{P'(z) + \sqrt{H(z)}},
\]

where

\[
H(z) = (n-1) \cdot [P'(z) \cdot P''(z)] - n \cdot P(z) \cdot P''(z),
\]

and \( z \) is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at \( z \), viz. \( |L(z)| \), is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots. The routine generates a sequence of iterates \( z_1, z_2, z_3, \ldots \), such that \( |P(z)| < |P(z_1)| \) and ensures that \( z_k + L(z_k) \) 'roughly' lies inside a circular region of radius \( |F| \) about \( z_k \) known to contain a zero of \( P(z) \); that is, \( |L(z_k)| \leq |F| \), where \( F \) denotes the Fejer bound (see Marden [1]) at the point \( z_k \).

Following Smith [2], \( F \) is taken to be \( \min(B, 1.445 \cdot n \cdot R) \), where \( B \) is an upper bound for the magnitude of the smallest zero given by

\[
B = 1.0001 \cdot \min(\sqrt{\frac{1}{n} \cdot L(z_k), |r_1|, |a_1/a_0|}),
\]

where \( r \) is the zero \( X \) of smaller magnitude of the quadratic equation

\[
2 \cdot P''(z) \cdot (2 \cdot n \cdot (n-1))X^2 + 2 \cdot P'(z) \cdot n \cdot P(z)X + P(z) = 0
\]

and the Cauchy lower bound \( R \) for the smallest zero is computed (using Newton's Method) as the positive root of the polynomial equation
\[ |a_0| z + |a_1| z + |a_2| z + \ldots + |a_{n-1}| z - |a_n| = 0. \]

Starting from the origin, successive iterates are generated according to the rule \( z_{k+1} = z_k + L(z_k) \) for \( k = 1, 2, 3, \ldots \) and \( L(z_k) \) is 'adjusted' so that \( |P(z_{k+1})| < |P(z_k)| \) and \( |L(z_k)| \leq |F| \). The iterative procedure terminates if \( P(z_{k+1}) \) is smaller in absolute value than the bound on the rounding error in \( P(z_k) \) and the current iterate \( z = z_k \) is taken to be a zero of \( P(z) \). The deflated polynomial \( P(z) = P(z)/(z-z_0) \) of degree \( n-1 \) is then formed, and the above procedure is repeated on the deflated polynomial until \( n \leq 3 \), whereupon the remaining roots are obtained via the 'standard' closed formulae for a linear (\( n = 1 \)) or quadratic (\( n = 2 \)) equation.

To obtain the roots of a quadratic polynomial, C02AHF(*) can be used.

4. References


5. Parameters

1: \( A(2,N+1) \) -- DOUBLE PRECISION array
   Input
   On entry: if \( A \) is declared with bounds \((2,0:N)\), then \( A(1,i) \)
   and \( A(2,i) \) must contain the real and imaginary parts of \( a_i \)
   (i.e., the coefficient of \( z^{n-i} \)), for \( i = 0, 1, \ldots, n \).
   Constraint: \( A(1,0) /= 0.0 \) or \( A(2,0) /= 0.0 \).

2: \( N \) -- INTEGER
   Input
On entry: the degree of the polynomial, \( n \). Constraint: \( n \geq 1 \).

3: SCALE -- LOGICAL Input
On entry: indicates whether or not the polynomial is to be scaled. See Section 8 for advice on when it may be preferable to set SCALE = .FALSE. and for a description of the scaling strategy. Suggested value: SCALE = .TRUE..

4: Z(2,N) -- DOUBLE PRECISION array Output
On exit: the real and imaginary parts of the roots are stored in \( Z(1,i) \) and \( Z(2,i) \) respectively, for \( i=1,2,\ldots,n \).

5: W(4*(N+1)) -- DOUBLE PRECISION array Workspace

6: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry \( A(1,0) = 0.0 \) and \( A(2,0) = 0.0 \),
or \( n < 1 \).

IFAIL= 2
The iterative procedure has failed to converge. This error is very unlikely to occur. If it does, please contact NAG immediately, as some basic assumption for the arithmetic has been violated. See also Section 8.

IFAIL= 3
Either overflow or underflow prevents the evaluation of \( P(z) \) near some of its zeros. This error is very unlikely to occur. If it does, please contact NAG immediately. See also Section 8.

7. Accuracy

All roots are evaluated as accurately as possible, but because of
the inherent nature of the problem complete accuracy cannot be guaranteed.

8. Further Comments

If SCALE = .TRUE., then a scaling factor for the coefficients is chosen as a power of the base B of the machine so that the largest coefficient in magnitude approaches $\text{THRESH} = B^{(EMAX-P)}$. Users should note that no scaling is performed if the largest coefficient in magnitude exceeds THRESH, even if SCALE = .TRUE. (For definition of B, EMAX and P see the Chapter Introduction X02.)

However, with SCALE = .TRUE., overflow may be encountered when the input coefficients $a_0, a_1, a_2, \ldots, a_n$ vary widely in magnitude, particularly on those machines for which $B$ overflows. In such cases, SCALE should be set to .FALSE. and the coefficients scaled so that the largest coefficient in magnitude does not exceed $B^{(EMAX-2*P)}$.

Even so, the scaling strategy used in C02AFF is sometimes insufficient to avoid overflow and/or underflow conditions. In such cases, the user is recommended to scale the independent variable (z) so that the disparity between the largest and smallest coefficient in magnitude is reduced. That is, use the routine to locate the zeros of the polynomial $d*P(cz)$ for some suitable values of $c$ and $d$. For example, if the original polynomial was $P(z) = 2i + 2z$, then choosing $c=2$ and $d=2$, for instance, would yield the scaled polynomial $i+z$, which is well-behaved relative to overflow and underflow and has 10 zeros which are 2 times those of $P(z)$.

If the routine fails with IFAIL = 2 or 3, then the real and imaginary parts of any roots obtained before the failure occurred are stored in Z in the reverse order in which they were found. Let $n$ denote the number of roots found before the failure occurred. Then $Z(1,n)$ and $Z(2,n)$ contain the real and imaginary parts of the 1st root found, $Z(1,n-1)$ and $Z(2,n-1)$ contain the real and imaginary parts of the 2nd root found, $\ldots$, $Z(1,n-1)$ and $Z(2,n-1)$ contain the real and imaginary parts of the $n$th root found.
found. After the failure has occurred, the remaining $2(n-n)$ elements of $Z$ contain a large negative number (equal to $-1/(X02AMF().\sqrt{2})$).

9. Example

To find the roots of the polynomial $a_5 z^5 + a_4 z^4 + a_3 z^3 + a_2 z^2 + a_1 z + a_0 = 0$, where $a_0 = (5.0+6.0i)$, $a_1 = (30.0+20.0i)$, $a_2 = -(0.2+6.0i)$, $a_3 = (50.0+100000.0i)$, $a_4 = -(2.0-40.0i)$ and $a_5 = (10.0+1.0i)$.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The roots are located using a modified form of Laguerre’s Method, originally proposed by Smith [2].

The method of Laguerre [3] can be described by the iterative scheme

\[
L(z) = \frac{-n*P(z)}{P'(z) + \sqrt{H(z)}},
\]

where

\[
H(z) = (n-1)*[(n-1)*(P'(z)P''(z)) - n*P(z)P''(z)]
\]

and \(z\) is specified.

The sign in the denominator is chosen so that the modulus of the Laguerre step at \(z\), viz. \(|L(z)|\), is as small as possible. The method can be shown to be cubically convergent for isolated roots (real or complex) and linearly convergent for multiple roots.

The routine generates a sequence of iterates \(z_1, z_2, z_3, \ldots\), such that \(|P(z_{k+1})| < |P(z_k)|\) and ensures that \(z + L(z)\) ‘roughly’ lies inside a circular region of radius \(|F|\) about \(z\) known to contain a zero of \(P(z)\); that is, \(|L(z) - z| = \epsilon = |F|\), where \(F\) denotes the Fejer bound (see Marden [1]) at the point \(z\). Following Smith [2], \(F\) is taken to be \(\min(B, 1.445*n*R)\), where \(B\) is an upper bound for the magnitude of the smallest zero given by

\[
B = 1.0001 \times \min(\sqrt{n*|L(z)|}, |r|, |a_1/a_0|)
\]

\(r\) is the zero \(X\) of smaller magnitude of the quadratic equation

\[
2(P'(z)/(2*n*(n-1))X + 2(P(z)/n)X + P(z) = 0)
\]
and the Cauchy lower bound $R$ for the smallest zero is computed (using Newton's Method) as the positive root of the polynomial equation

$$|z_0^n + a_1 z_0^{n-1} + a_2 z_0^{n-2} + \ldots + a_n| = 0.$$ 

Starting from the origin, successive iterates are generated according to the rule $z_k = z_{k-1} + L(z_k)$ for $k=1,2,\ldots$ and $L(z_k)$ is

$$L(z_k) = \frac{1}{k+1} z_k^k.$$ 

Iterative procedure terminates if $P(z_k)$ is smaller in absolute value than the bound on the rounding error in $P(z_k)$ and the current iterate $z_k$ is taken to be a zero of $P(z_k)$ (as is its conjugate $z_{k-1}$ if $z_k$ is complex). The deflated polynomial

$$P(z) = \frac{P(z)}{(z-z_0)}$$

of degree $n-1$ if $z$ is real

$$P(z) = \frac{P(z)}{(z-z_0)(z-z_1)}$$

of degree $n-2$ if $z$ is complex) is then formed, and the above procedure is repeated on the deflated polynomial until $n<3$, whereupon the remaining roots are obtained via the 'standard' closed formulae for a linear ($n=1$) or quadratic ($n=2$) equation.

To obtain the roots of a quadratic polynomial, C02AJF(*) can be used.

4. References


5. Parameters

1: A(N+1) -- DOUBLE PRECISION array  
   Input
   On entry: if A is declared with bounds (0:N), then A(i)
   must contain a (i.e., the coefficient of z^i), for
   i=0,1,...,n. Constraint: A(0) /= 0.0.

2: N -- INTEGER  
   Input

3: SCALE -- LOGICAL  
   Input
   On entry: indicates whether or not the polynomial is to be
   scaled. See Section 8 for advice on when it may be
   preferable to set SCALE = .FALSE. and for a description of
   the scaling strategy. Suggested value: SCALE = .TRUE..

4: Z(2,N) -- DOUBLE PRECISION array  
   Output
   On exit: the real and imaginary parts of the roots are
   stored in Z(1,i) and Z(2,i) respectively, for i=1,2,...,n.
   Complex conjugate pairs of roots are stored in consecutive
   pairs of elements of Z; that is, Z(1,i+1) = Z(1,i) and
   Z(2,i+1)=-Z(2,i).

5: W(2*(N+1)) -- DOUBLE PRECISION array  
   Workspace

6: IFAIL -- INTEGER  
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry A(0) = 0.0,
   or       N < 1.

IFAIL= 2
   The iterative procedure has failed to converge. This error
   is very unlikely to occur. If it does, please contact NAG
immediately, as some basic assumption for the arithmetic has been violated. See also Section 8.

IFAIL= 3
Either overflow or underflow prevents the evaluation of P(z) near some of its zeros. This error is very unlikely to occur. If it does, please contact NAG immediately. See also Section 8.

7. Accuracy
All roots are evaluated as accurately as possible, but because of the inherent nature of the problem complete accuracy cannot be guaranteed.

8. Further Comments
If SCALE = .TRUE., then a scaling factor for the coefficients is chosen as a power of the base B of the machine so that the largest coefficient in magnitude approaches THRESH = B^{EMAX-P}. Users should note that no scaling is performed if the largest coefficient in magnitude exceeds THRESH, even if SCALE = .TRUE.. (For definition of B, EMAX and P see the Chapter Introduction X02.)

However, with SCALE = .TRUE., overflow may be encountered when the input coefficients a_0, a_1, a_2, ..., a_n vary widely in magnitude, particularly on those machines for which B overflows. In such cases, SCALE should be set to .FALSE. and the coefficients scaled so that the largest coefficient in magnitude does not exceed B^{(EMAX-2*P)}. Even so, the scaling strategy used in C02AGF is sometimes insufficient to avoid overflow and/or underflow conditions. In such cases, the user is recommended to scale the independent variable (z) so that the disparity between the largest and smallest coefficient in magnitude is reduced. That is, use the routine to locate the zeros of the polynomial d*P(cz) for some suitable values of c and d. For example, if the original polynomial was P(z)=2 +2 z, then choosing c=2 and d=2, for instance, would yield the scaled polynomial 1+z, which is well-behaved relative to overflow and underflow and has zeros which are 2 times those of P(z).
If the routine fails with IFAIL = 2 or 3, then the real and imaginary parts of any roots obtained before the failure occurred are stored in Z in the reverse order in which they were found. Let n denote the number of roots found before the failure occurred. Then Z(1,n) and Z(2,n) contain the real and imaginary parts of the 1st root found, Z(1,n-1) and Z(2,n-1) contain the real and imaginary parts of the 2nd root found, ..., Z(1,n) and Z(2,n) contain the real and imaginary parts of the n th root found. After the failure has occurred, the remaining 2*(n-n) elements of Z contain a large negative number (equal to 
\[-1/(X02AMF().\sqrt{2})).\]

9. Example

To find the roots of the 5th degree polynomial
\[z^5 + 2z^4 + 3z^3 + 4z^2 + 5z + 6 = 0.\]

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

NagPolynomialRootsPackage (NAGC02)
Exports:
c02aff  c02agf

— package NAGC02 NagPolynomialRootsPackage —

)abbrev package NAGC02 NagPolynomialRootsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:27 1994
++ Description:
++ This package uses the NAG Library to compute the zeros of a
++ polynomial with real or complex coefficients.

NagPolynomialRootsPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==> with
  c02aff : (Matrix DoubleFloat,Integer,Boolean,Integer) -> Result
    ++ c02aff(a,n,scale,ifail)
    ++ finds all the roots of a complex polynomial equation,
    ++ using a variant of Laguerre’s Method.
    ++ See {downlink{Manual Page}{manpageXXc02aff}}.
  c02agf : (Matrix DoubleFloat,Integer,Boolean,Integer) -> Result
    ++ c02agf(a,n,scale,ifail)
    ++ finds all the roots of a real polynomial equation, using a
    ++ variant of Laguerre’s Method.
    ++ See {downlink{Manual Page}{manpageXXc02agf}}.

Implementation ==> add

import Lisp
import DoubleFloat
import Matrix DoubleFloat
import Any
import Record
import Integer
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(Boolean)

c02aff(aArg:Matrix DoubleFloat,nArg:Integer,scaleArg:Boolean,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c02aff",_
["z":S,"w":S]$Lisp,_
package NAGC05 NagRootFindingPackage

-- NagRootFindingPackage.input --

)clear all

-- S 1 of 26
show NagRootFindingPackage
--R
--R NagRootFindingPackage is a package constructor
--R Abbreviation for NagRootFindingPackage is NAGC05
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGC05
--R
--R----------------------------------- Operations -------------------------------------
--R c05adf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Union(fn: FileName,fp: Asp1(F))) -> Result
--R c05nbf : (Integer,Integer,Matrix(DoubleFloat),DoubleFloat,Integer,Union(fn: FileName,fp: Asp6(FCN))) -> Result
--R c05pbf : (Integer,Integer,Integer,Matrix(DoubleFloat),DoubleFloat,Integer,Union(fn: FileName,fp: Asp35(FCN))) -> Result

)clear all

)show ArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 2

)show ScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 3

f:ASP1(F):=exp(-X)-X
--R
--R (3) F
--R Type: Asp1(F)
--E 4

a:SF:=0.0
--R
--R (4) 0.
--R Type: DoubleFloat
--E 5

b:SF:=1.0
CHAPTER 15. CHAPTER N

--R
--R
--R (5) 1.
--R Type: DoubleFloat
--E 6

--S 7 of 26
eps:SF:=1.0e-5
--R
--R
--R (6) 9.9999999999999991E-6
--R Type: DoubleFloat
--E 7

--S 8 of 26
eta:SF:=0.0
--R
--R
--R (7) 0.
--R Type: DoubleFloat
--E 8

--S 9 of 26
result:=c0Sadf(a,b,eps,eta,-1,f)
--E 9

)clear all

--S 10 of 26
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 10

--S 11 of 26
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 11

--S 12 of 26
n:=9
--R
--R
--R (3) 9
--R Type: PositiveInteger
lwa := n*(3*n+13)/2

xtol:SF := 1.0e-9

fi := ASP6(FCN) := retract vector[

result := c05nbf(n, lwa, x, xtol, -1, fi)
showArrayValues true
--R
--R
--R  (1) true
--R Type: Boolean
--E 18

--S 19 of 26
showScalarValues true
--R
--R
--R  (2) true
--R Type: Boolean
--E 19

--S 20 of 26
n:=9
--R
--R
--R  (3) 9
--R Type: PositiveInteger
--E 20

--S 21 of 26
ldfjac:=n
--R
--R
--R  (4) 9
--R Type: PositiveInteger
--E 21

--S 22 of 26
lwa:=n*(n+13)/2
--R
--R
--R  (5) 99
--R Type: Fraction(Integer)
--E 22

--S 23 of 26
xtol:SF:=1.0e-9
--R
--R
--R  (6) 9.999999999999998E-10
--R Type: DoubleFloat
--E 23

--S 24 of 26
x:Matrix SF:= [[-1.0 , -1.0 , -1.0 , -1.0 , -1.0 , -1.0 , -1.0 , -1.0 , -1.0 ]]
--R
This package uses the NAG Library to calculate real zeros of continuous real functions of one or more variables. (Complex equations must be expressed in terms of the equivalent larger system of real equations.)

Chapter C05

Roots of One or More Transcendental Equations

1. Scope of the Chapter

This chapter is concerned with the calculation of real zeros of continuous real functions of one or more variables. (Complex equations must be expressed in terms of the equivalent larger
system of real equations.)

2. Background to the Problems

The chapter divides naturally into two parts.

2.1. A Single Equation

The first deals with the real zeros of a real function of a single variable $f(x)$.

At present, there is only one routine with a simple calling sequence. This routine assumes that the user can determine an initial interval $[a,b]$ within which the desired zero lies, that is $f(a)f(b)<0$, and outside which all other zeros lie. The routine then systematically subdivides the interval to produce a final interval containing the zero. This final interval has a length bounded by the user’s specified error requirements; the end of the interval where the function has smallest magnitude is returned as the zero. This routine is guaranteed to converge to a simple zero of the function. (Here we define a simple zero as a zero corresponding to a sign-change of the function.) The algorithm used is due to Bus and Dekker.

2.2. Systems of Equations

The routines in the second part of this chapter are designed to solve a set of nonlinear equations in $n$ unknowns

$$T \begin{align*}
    f_i(x) &= 0, \quad i=1,2,\ldots,n, \\
    x &= (x_1, x_2, \ldots, x_n)
\end{align*} \tag{1}
$$

where $T$ stands for transpose.

It is assumed that the functions are continuous and differentiable so that the matrix of first partial derivatives of the functions, the Jacobian matrix $J(x) = \frac{df_i}{dx_j}$ evaluated at the point $x$, exists, though it may not be possible to calculate it directly.

The functions $f_i$ must be independent, otherwise there will be an infinity of solutions and the methods will fail. However, even when the functions are independent the solutions may not be unique. Since the methods are iterative, an initial guess at the solution has to be supplied, and the solution located will usually be the one closest to this initial guess.
2.3. References


3. Recommendations on Choice and Use of Routines

3.1. Zeros of Functions of One Variable

There is only one routine (C05ADF) for solving a single nonlinear equation. This routine is designed for solving problems where the function \( f(x) \) whose zero is to be calculated, can be coded as a user-supplied routine.

C05ADF may only be used when the user can supply an interval \([a,b]\) containing the zero, that is \( f(a)\cdot f(b) < 0 \).

3.2. Solution of Sets of Nonlinear Equations

The solution of a set of nonlinear equations

\[
  f(x_1, x_2, \ldots, x_n) = 0, \quad i = 1, 2, \ldots, n
\]

can be regarded as a special case of the problem of finding a minimum of a sum of squares

\[
  s(x) = \sqrt{\sum_{i=1}^{m} f(x_1, x_2, \ldots, x_n)^2} \quad (m \geq n).
\]

So the routines in Chapter E04 of the Library are relevant as well as the special nonlinear equations routines.

There are two routines (C05NBF and C05PBF) for solving a set of nonlinear equations. These routines require the \( f \) (and possibly
their derivatives) to be calculated in user-supplied routines. These should be set up carefully so the Library routines can work as efficiently as possible.

The main decision which has to be made by the user is whether to supply the derivatives. It is advisable to do so if possible, since the results obtained by algorithms which use derivatives are generally more reliable than those obtained by algorithms which do not use derivatives.

C05PBF requires the user to provide the derivatives, whilst C05NBF does not. C05NBF and C05PBF are easy-to-use routines. A routine, C05ZAF, is provided for use in conjunction with C05PBF to check the user-provided derivatives for consistency with the functions themselves. The user is strongly advised to make use of this routine whenever C05PBF is used.

Firstly, the calculation of the functions and their derivatives should be ordered so that cancellation errors are avoided. This is particularly important in a routine that uses these quantities to build up estimates of higher derivatives.

Secondly, scaling of the variables has a considerable effect on the efficiency of a routine. The problem should be designed so that the elements of \( x \) are of similar magnitude. The same comment applies to the functions, all the \( f_i \) should be of comparable size.

The accuracy is usually determined by the accuracy parameters of the routines, but the following points may be useful:

(i) Greater accuracy in the solution may be requested by choosing smaller input values for the accuracy parameters. However, if unreasonable accuracy is demanded, rounding errors may become important and cause a failure.

(ii) Some idea of the accuracies of the \( x_i \) may be obtained by monitoring the progress of the routine to see how many figures remain unchanged during the last few iterations.

(iii) An approximation to the error in the solution \( x \), given by \( e \) where \( e \) is the solution to the set of linear equations

\[
J(x)e = -f(x)
\]
\[ T \]
\[ \text{where } f(x) = (f(x), f(x), \ldots, f(x)) \text{ (see Chapter F04).} \]

(iv) If the functions \( f_i(x) \) are changed by small amounts \( \epsilon_i \), for \( i = 1, 2, \ldots, n \), then the corresponding change in the solution \( x \) is given approximately by \( (\sigma) \), where \( (\sigma) \) is the solution of the set of linear equations

\[ J(x)(\sigma) = -\epsilon, \text{ (see Chapter F04).} \]

Thus one can estimate the sensitivity of \( x \) to any uncertainties in the specification of \( f_i(x) \), for \( i = 1, 2, \ldots, n \).

3.3. Index

Zeros of functions of one variable:
- Bus and Dekker algorithm C05ADF

Zeros of functions of several variables:
- easy-to-use C05NBF
- easy-to-use, derivatives required C05PBF

Checking Routine:
- Checks user-supplied Jacobian C05ZAF

C05 -- Roots of One or More Transcendental Equations

Contents -- C05

Chapter C05

Roots of One or More Transcendental Equations

C05ADF Zero of continuous function in given interval, Bus and Dekker algorithm

C05NBF Solution of system of nonlinear equations using function values only

C05PBF Solution of system of nonlinear equations using 1st derivatives

C05ZAF Check user's routine for calculating 1st derivatives
C05 -- Roots of One or More Transcendental Equations
C05ADF
C05ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C05ADF locates a zero of a continuous function in a given interval by a combination of the methods of linear interpolation, extrapolation and bisection.

2. Specification

SUBROUTINE C05ADF (A, B, EPS, ETA, F, X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION A, B, EPS, ETA, F, X
EXTERNAL F

3. Description

The routine attempts to obtain an approximation to a simple zero of the function \( f(x) \) given an initial interval \([a,b]\) such that \( f(a) \cdot f(b) \leq 0 \). The zero is found by calls to C05AZF(*) whose specification should be consulted for details of the method used.

The approximation \( x \) to the zero (alpha) is determined so that one or both of the following criteria are satisfied:

(i) \(|x-\alpha| < \text{EPS},\)

(ii) \(|f(x)| < \text{ETA}.

4. References

None.

5. Parameters

1: A -- DOUBLE PRECISION
   On entry: the lower bound of the interval, a.

2: B -- DOUBLE PRECISION
   On entry: the upper bound of the interval, b. Constraint: \( B \neq A \).

3: EPS -- DOUBLE PRECISION
   Input
On entry: the absolute tolerance to which the zero is required (see Section 3). Constraint: \( \text{EPS} > 0.0 \).

4: ETA -- DOUBLE PRECISION  
   Input  
   On entry: a value such that if \(|f(x)| < \text{ETA}\), \(x\) is accepted as the zero. ETA may be specified as 0.0 (see Section 7).

5: F -- DOUBLE PRECISION FUNCTION, supplied by the user.  
   External Procedure  
   F must evaluate the function \(f\) whose zero is to be determined.

Its specification is:

\[
\text{DOUBLE PRECISION FUNCTION } F(XX)  
\text{DOUBLE PRECISION } XX  
\]

1: XX -- DOUBLE PRECISION  
   Input  
   On entry: the point at which the function must be evaluated.
   F must be declared as EXTERNAL in the (sub)program from which C05ADF is called. Parameters denoted as Input must not be changed by this procedure.

6: X -- DOUBLE PRECISION  
   Output  
   On exit: the approximation to the zero.

7: IFAIL -- INTEGER  
   Input/Output  
   Before entry, IFAIL must be assigned a value. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1  
   On entry \( \text{EPS} \leq 0.0 \),
   or \( A = B \),
   or \( F(A) \cdot F(B) > 0.0 \).

IFAIL= 2  
   Too much accuracy has been requested in the computation, that is, \( \text{EPS} \) is too small for the computer being used. The final value of \(X\) is an accurate approximation to the zero.
CHAPTER 15. CHAPTER N

IFAIL = 3
A change in sign of f(x) has been determined as occurring near the point defined by the final value of X. However, there is some evidence that this sign-change corresponds to a pole of f(x).

IFAIL = 4
Indicates that a serious error has occurred in C05AZF(*).
Check all routine calls. Seek expert help.

7. Accuracy

This depends on the value of EPS and ETA. If full machine accuracy is required, they may be set very small, resulting in an error exit with IFAIL = 2, although this may involve more iterations than a lesser accuracy. The user is recommended to set ETA = 0.0 and to use EPS to control the accuracy, unless he has considerable knowledge of the size of f(x) for values of x near the zero.

8. Further Comments

The time taken by the routine depends primarily on the time spent evaluating F (see Section 5).

If it is important to determine an interval of length less than EPS containing the zero, or if the function F is expensive to evaluate and the number of calls to F is to be restricted, then use of C05AZF(*) is recommended. Use of C05AZF(*) is also recommended when the structure of the problem to be solved does not permit a simple function F to be written: the reverse communication facilities of C05AZF(*) are more flexible than the direct communication of F required by C05ADF.

9. Example

\[-x\]

The example program below calculates the zero of \( e^{-x} \) within the interval \([0,1]\) to approximately 5 decimal places.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C05NF(3NAG) Foundation Library (12/10/92) C05NF(3NAG)
C05 -- Roots of One or More Transcendental Equations C05NF
C05NBF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C05NBF is an easy-to-use routine to find a solution of a system of nonlinear equations by a modification of the Powell hybrid method.

2. Specification

SUBROUTINE C05NBF (FCN, N, X, FVEC, XTOL, WA, LWA, IFAIL)
INTEGER N, LWA, IFAIL
DOUBLE PRECISION X(N), FVEC(N), XTOL, WA(LWA)
EXTERNAL FCN

3. Description

The system of equations is defined as:

\[ f(x_1, x_2, ..., x_n) = 0, \text{ for } i=1,2,...,n. \]

C05NBF is based upon the MINPACK routine HYBRD1 (More et al [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is approximated by forward differences, but these are not used again until the rank-1 method fails to produce satisfactory progress. For more details see Powell [2].

4. References


5. Parameters
CHAPTER 15. CHAPTER N

1: FCN -- SUBROUTINE, supplied by the user.

   External Procedure

FCN must return the values of the functions \( f_i \) at a point \( x \).

Its specification is:

\[
\text{SUBROUTINE FCN}(N, X, FVEC, IFLAG)
\]
\[
\text{INTEGER} \quad N, \text{IFLAG}
\]
\[
\text{DOUBLE PRECISION} \quad X(N), \text{FVEC}(N)
\]

1: \( N \) -- INTEGER \hspace{1cm} \text{Input}
   On entry: the number of equations, \( n \).

2: \( X(N) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Input/Output}
   On entry: the components of the point \( x \) at which the functions must be evaluated.

3: \( FVEC(N) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Output}
   On exit: the function values \( f_i(x) \) (unless IFLAG is set to a negative value by FCN).

4: \( IFLAG \) -- INTEGER \hspace{1cm} \text{Input/Output}
   On entry: IFLAG > 0. On exit: in general, IFLAG should not be reset by FCN. If, however, the user wishes to terminate execution (perhaps because some illegal point \( X \) has been reached), then IFLAG should be set to a negative integer. This value will be returned through IFAIL. FCN must be declared as EXTERNAL in the (sub)program from which C05NBF is called. Parameters denoted as Input must not be changed by this procedure.

2: \( N \) -- INTEGER \hspace{1cm} \text{Input}
   On entry: the number of equations, \( n \). Constraint: \( N > 0 \).

3: \( X(N) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Input/Output}
   On entry: an initial guess at the solution vector. On exit: the final estimate of the solution vector.

4: \( FVEC(N) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Output}
   On exit: the function values at the final point, \( X \).

5: \( XTOL \) -- DOUBLE PRECISION \hspace{1cm} \text{Input}
   On entry: the accuracy in \( X \) to which the solution is required. Suggested value: the square root of the machine precision. Constraint: \( XTOL \geq 0.0 \).

6: \( WA(LWA) \) -- DOUBLE PRECISION array \hspace{1cm} \text{Workspace}
7: LWA -- INTEGER Input
    On entry: the dimension of the array WA. Constraint:
               LWA>=N*(3*N+13)/2.

8: IFAIL -- INTEGER Input/Output
    On entry: IFAIL must be set to 0, -1 or 1. For users not
    familiar with this parameter (described in the Essential
    Introduction) the recommended value is 0.
    On exit: IFAIL = 0 unless the routine detects an error (see
             Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL< 0
   The user has set IFLAG negative in FCN. The value of IFAIL
   will be the same as the user’s setting of IFLAG.

IFAIL= 1
   On entry N <= 0,
   or      XTOL < 0.0,
   or      LWA<N*(3*N+13)/2.

IFAIL= 2
   There have been at least 200*(N+1) evaluations of FCN.
   Consider restarting the calculation from the final point
   held in X.

IFAIL= 3
   No further improvement in the approximate solution X is
   possible; XTOL is too small.

IFAIL= 4
   The iteration is not making good progress. This failure exit
   may indicate that the system does not have a zero, or that
   the solution is very close to the origin (see Section 7).
   Otherwise, rerunning COSNB from a different starting point
   may avoid the region of difficulty.

7. Accuracy
If \( x \) is the true solution, \texttt{CO5NBF} tries to ensure that

\[
\|x-x\| \leq \text{XTOL} \|x\|. 
\]

- \( k \)

If this condition is satisfied with \( \text{XTOL}=10^{-k} \), then the larger components of \( x \) have \( k \) significant decimal digits. There is a danger that the smaller components of \( x \) may have large relative errors, but the fast rate of convergence of \texttt{COSNBF} usually avoids this possibility.

If \( \text{XTOL} \) is less than machine precision, and the above test is satisfied with the machine precision in place of \( \text{XTOL} \), then the routine exits with \( \text{IFAIL} = 3 \).

Note: this convergence test is based purely on relative error, and may not indicate convergence if the solution is very close to the origin.

The test assumes that the functions are reasonably well behaved. If this condition is not satisfied, then \texttt{CO5NBF} may incorrectly indicate convergence. The validity of the answer can be checked, for example, by rerunning \texttt{CO5NBF} with a tighter tolerance.

8. Further Comments

The time required by \texttt{CO5NBF} to solve a given problem depends on \( n \), the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by \texttt{CO5NBF} to process each call of \texttt{FCN} is about \( 11.5 \times n \). Unless \texttt{FCN} can be evaluated quickly, the timing of \texttt{CO5NBF} will be strongly influenced by the time spent in \texttt{FCN}.

Ideally the problem should be scaled so that at the solution the function values are of comparable magnitude.

9. Example

To determine the values \( x_1, \ldots, x_9 \) which satisfy the tridiagonal equations:

\[
(3-2x_i)x_{i+1} - 2x_i = -1, \quad i=1,2,\ldots,8
\]

where \( x_9 = 0 \).
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

The system of equations is defined as:

\[-x + (3-2x)x = -1, \quad 8 \quad 9 \quad 9\]

The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C05PBF is an easy-to-use routine to find a solution of a system of nonlinear equations by a modification of the Powell hybrid method. The user must provide the Jacobian.

2. Specification

```
SUBROUTINE C05PBF (FCN, N, X, FVEC, FJAC, LDFJAC, XTOL, WA, LWA, IFAIL)

INTEGER N, LDFJAC, LWA, IFAIL
DOUBLE PRECISION X(N), FVEC(N), FJAC(LDFJAC,N), XTOL, WA(LWA)
EXTERNAL FCN
```

3. Description

The system of equations is defined as:

\[ f(x, x, \ldots, x) = 0, \quad i=1,2,\ldots,n. \]

C05PBF is based upon the MINPACK routine HYBRJ1 (More et al [1]). It chooses the correction at each step as a convex combination of the Newton and scaled gradient directions. Under reasonable conditions this guarantees global convergence for starting points far from the solution and a fast rate of convergence. The Jacobian is updated by the rank-1 method of Broyden. At the starting point the Jacobian is calculated, but it is not recalculated until the rank-1 method fails to produce satisfactory progress. For more details see Powell [2].
CHAPTER 15. CHAPTER N

4. References


5. Parameters

1: FCN -- SUBROUTINE, supplied by the user.

   External Procedure
   Depending upon the value of IFLAG, FCN must either return
   the values of the functions \( f \) at a point \( x \) or return the
   Jacobian at \( x \).

   Its specification is:

   ```
   SUBROUTINE FCN (N, X, FVEC, FJAC, LDFJAC, IFLAG)
   INTEGER N, LDFJAC, IFLAG
   DOUBLE PRECISION X(N), FVEC(N), FJAC(LDFJAC,N)
   ```

   1: \( N \) -- INTEGER
   On entry: the number of equations, \( n \).

   2: \( X(N) \) -- DOUBLE PRECISION array
   On entry: the components of the point \( x \) at which the
   functions or the Jacobian must be evaluated.

   3: \( FVEC(N) \) -- DOUBLE PRECISION array
   On exit: if IFLAG = 1 on entry, \( FVEC \) must contain the
   function values \( f(x) \) (unless IFLAG is set to a
   negative value by FCN). If IFLAG = 2 on entry, \( FVEC \)
   must not be changed.

   4: \( FJAC(LDFJAC,N) \) -- DOUBLE PRECISION array
   On exit: if IFLAG = 2 on entry, \( FJAC(i,j) \) must contain
   the value of \( \frac{df}{dx} \) at the point \( x \), for \( i,j=1,2,\ldots,n \)
   (unless IFLAG is set to a negative value by FCN).

   If IFLAG = 1 on entry, \( FJAC \) must not be changed.
5: LDFJAC -- INTEGER
   Input
   On entry: the first dimension of FJAC.

6: IFLAG -- INTEGER
   Input/Output
   On entry: IFLAG = 1 or 2:
       if IFLAG = 1, FVEC is to be updated;
       if IFLAG = 2, FJAC is to be updated.
   On exit: in general, IFLAG should not be reset by FCN.
   If, however, the user wishes to terminate execution
   (perhaps because some illegal point x has been reached)
   then IFLAG should be set to a negative integer. This
   value will be returned through IFAIL.
   FCN must be declared as EXTERNAL in the (sub)program
   from which C05PBF is called. Parameters denoted as
   Input must not be changed by this procedure.

2: N -- INTEGER
   Input
   On entry: the number of equations, n. Constraint: N > 0.

3: X(N) -- DOUBLE PRECISION array
   Input/Output
   On entry: an initial guess at the solution vector. On
   exit: the final estimate of the solution vector.

4: FVEC(N) -- DOUBLE PRECISION array
   Output
   On exit: the function values at the final point, X.

5: FJAC(LDFJAC,N) -- DOUBLE PRECISION array
   Output
   On exit: the orthogonal matrix Q produced by the QR
   factorization of the final approximate Jacobian.

6: LDFJAC -- INTEGER
   Input
   On entry: the first dimension of the array FJAC as declared in the
   (sub)program from which C05PBF is called.
   Constraint: LDFJAC >= N.

7: XTOL -- DOUBLE PRECISION
   Input
   On entry: the accuracy in X to which the solution is
   required. Suggested value: the square root of the machine
   precision. Constraint: XTOL >= 0.0.

8: WA(LWA) -- DOUBLE PRECISION array
   Workspace

9: LWA -- INTEGER
   Input
   On entry: the dimension of the array WA. Constraint:
   LWA>=N*(N+13)/2.

10: IFAIL -- INTEGER
    Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL< 0
A negative value of IFAIL indicates an exit from C05PBF because the user has set IFLAG negative in FCN. The value of IFAIL will be the same as the user’s setting of IFLAG.

IFAIL= 1
On entry N <= 0,
or     LDFJAC < N,
or     XTOL < 0.0,
or     LWA<N*(N+13)/2.

IFAIL= 2
There have been 100*(N+1) evaluations of the functions. Consider restarting the calculation from the final point held in X.

IFAIL= 3
No further improvement in the approximate solution X is possible; XTOL is too small.

IFAIL= 4
The iteration is not making good progress. This failure exit may indicate that the system does not have a zero or that the solution is very close to the origin (see Section 7). Otherwise, rerunning C05PBF from a different starting point may avoid the region of difficulty.

7. Accuracy

If x is the true solution, C05PBF tries to ensure that
\[\|x-x\| \leq \text{XTOL}\|x\|.\]

If this condition is satisfied with XTOL = 10^{-k}, then the larger components of \(x\) have \(k\) significant decimal digits. There is a danger that the smaller components of \(x\) may have large relative errors, but the fast rate of convergence of C05PBF usually avoids the possibility.

If XTOL is less than machine precision and the above test is satisfied with the machine precision in place of XTOL, then the routine exits with IFAIL = 3.

Note: this convergence test is based purely on relative error, and may not indicate convergence if the solution is very close to the origin.

The test assumes that the functions and Jacobian are coded consistently and that the functions are reasonably well behaved. If these conditions are not satisfied then C05PBF may incorrectly indicate convergence. The coding of the Jacobian can be checked using C05ZAF. If the Jacobian is coded correctly, then the validity of the answer can be checked by rerunning C05PBF with a tighter tolerance.

8. Further Comments

The time required by C05PBF to solve a given problem depends on \(n\), the behaviour of the functions, the accuracy requested and the starting point. The number of arithmetic operations executed by C05PBF is about 11.5\(n\) to process each evaluation of the functions and about 1.3\(n\) to process each evaluation of the Jacobian. Unless FCN can be evaluated quickly, the timing of C05PBF will be strongly influenced by the time spent in FCN.

Ideally the problem should be scaled so that, at the solution, the function values are of comparable magnitude.

9. Example

To determine the values \(x_1, \ldots, x_9\) which satisfy the tridiagonal equations:

\[
(3-2x_1)x_1 - 2x_2 = -1
\]

\[
1 1 2
\]

\[
(3-2x_2)x_2 - 2x_3 = -1
\]

\[
1 1 2
\]

\[
(3-2x_3)x_3 - 2x_4 = -1
\]

\[
1 1 2
\]

\[
(3-2x_4)x_4 - 2x_5 = -1
\]

\[
1 1 2
\]

\[
(3-2x_5)x_5 - 2x_6 = -1
\]

\[
1 1 2
\]

\[
(3-2x_6)x_6 - 2x_7 = -1
\]

\[
1 1 2
\]

\[
(3-2x_7)x_7 - 2x_8 = -1
\]

\[
1 1 2
\]

\[
(3-2x_8)x_8 - 2x_9 = -1
\]

\[
1 1 2
\]

\[
(3-2x_9)x_9 - 2x_1 = -1
\]

\[
1 1 2
\]
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

NagRootFindingPackage (NAGC05)

Exports:
  c05adf  c05nbf  c05pbf

— package NAGC05 NagRootFindingPackage —

)abbrev package NAGC05 NagRootFindingPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:28 1994
++ Description:
++ This package uses the NAG Library to calculate real zeros of
++ continuous real functions of one or more variables. (Complex
++ equations must be expressed in terms of the equivalent larger
++ system of real equations.)

NagRootFindingPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage
Exports ==> with
  c05adf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer,Union(fn:FileName,fp:Asp1(F))) -> Result
++ c05adf(a,b,eps,eta,ifail,f)
++ locates a zero of a continuous function in a given
++ interval by a combination of the methods of linear interpolation,
++ extrapolation and bisection.
++ See \downlink{Manual Page}{manpageXXc05adf}.
  c05nbf : (Integer,Integer,Matrix DoubleFloat,DoubleFloat,Integer,Union(fn:FileName,fp:Asp6(FCN))) -> Result
++ c05nbf(n,lwa,x,xtol,ifail,fcn)
++ is an easy-to-use routine to find a solution of a system
++ of nonlinear equations by a modification of the Powell hybrid
++ method.
++ See \downlink{Manual Page}{manpageXXc05nbf}.
  c05pbf : (Integer,Integer,Integer,Matrix DoubleFloat,Integer,Union(fn:FileName,fp:Asp35(FCN))) -> Result
++ c05pbf(n,ldfjac,lwa,x,xtol,ifail,fcn)
++ is an easy-to-use routine to find a solution of a system
++ of nonlinear equations by a modification of the Powell hybrid
++ method. The user must provide the Jacobian.
++ See \downlink{Manual Page}{manpageXXc05pbf}.
Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import FortranPackage
import Union(fn:FileName,fp:Asp1(F))
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(Matrix DoubleFloat)
import AnyFunctions1(Integer)

c05adf(aArg:DoubleFloat,bArg:DoubleFloat,epsArg:DoubleFloat,etaArg:DoubleFloat,ifailArg:Integer, _
  fArg:Union(fn:FileName,fp:Asp1(F))): Result ==
pushFortranOutputStack(ifFilename := aspFilename "f")$FOP
  if fArg case fn
    then outputAsFortran(fArg.fn)
    else outputAsFortran(fArg.fp)
popFortranOutputStack()$FOP
  ["c05adf", _
  ["a":S,"b":S,"eps":S,"eta":S,"x":S_,
   "ifail":S,"f":S]$Lisp, _
c05nbf(nArg:Integer,lwaArg:Integer,xArg:Matrix DoubleFloat,_
xtolArg:DoubleFloat,ifailArg:Integer,_
fcnArg:Union(fn:FileName,fp:Asp6(FCN))): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
    then outputAsFortran(fcnArg.fn)
    else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
[[invokeNagman([fcnFilename]$Lisp,_
  "c05nbf",_
  "fvec"::S,"x"::S,"wa"::S]$Lisp,_
  ["fvec"::S,"wa"::S,"fcn"::S]$Lisp_,
  ["double"::S,["fvec"::S,"n"::S]$Lisp,["x"::S,"n"::S]$Lisp_,
  "xtol1"::S,["x"::S,"lwa"::S,$Lisp,"fcn"::S]$Lisp_,
]$Lisp_,
  [[nArg::Any,lwaArg::Any,xtolArg::Any,ifailArg::Any ]]_  
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

c05pbf(nArg:Integer,ldfjacArg:Integer,lwaArg:Integer,_
xArg:Matrix DoubleFloat,xtolArg:DoubleFloat,ifailArg:Integer,_
fcnArg:Union(fn:FileName,fp:Asp35(FCN))): Result ==
pushFortranOutputStack(fcnFilename := aspFilename "fcn")$FOP
if fcnArg case fn
    then outputAsFortran(fcnArg.fn)
    else outputAsFortran(fcnArg.fp)
popFortranOutputStack()$FOP
[[invokeNagman([fcnFilename]$Lisp,_
  "c05pbf",_
  ["double"::S,["fvec"::S,"n"::S]$Lisp,["x"::S,"n"::S]$Lisp_,
  ["fjac"::S,"ldfjac"::S,"n"::S]$Lisp_,
  "ifail"::S]$Lisp_]}_
package NAGC06 NagSeriesSummationPackage

-- NagSeriesSummationPackage.input --

)set break resume
)sys rm -f NagSeriesSummationPackage.output
)spool NagSeriesSummationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 88
)show NagSeriesSummationPackage
--R
--R NagSeriesSummationPackage is a package constructor
--R Abbreviation for NagSeriesSummationPackage is NAGC06
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NAGC06
--R
--R--------------------------------- Operations ---------------------------------
--R c06eaf : (Integer, Matrix(DoubleFloat), Integer) -> Result
--R c06ebf : (Integer, Matrix(DoubleFloat), Integer) -> Result
--R c06ecf : (Integer, Matrix(DoubleFloat), Matrix(DoubleFloat), Integer) -> Result
--R c06ekf : (Integer, Integer, Matrix(DoubleFloat), Matrix(DoubleFloat), Integer) -> Result
--R c06fpf : (Integer, Integer, String, Matrix(DoubleFloat), Matrix(DoubleFloat), Integer) -> Result
--R c06fqf : (Integer, Integer, String, Matrix(DoubleFloat), Matrix(DoubleFloat), Integer) -> Result
--R c06frf : (Integer, Integer, String, Matrix(DoubleFloat), Matrix(DoubleFloat), Matrix(DoubleFloat), Integer) -> Result
clear all

showArrayValues true

(1) true

Type: Boolean

showScalarValues true

(2) true

Type: Boolean

x:Matrix SF:=[[0.34907, 0.54890, 0.74776, 0.94459, 1.13850, 1.32850, 1.51370]]

(3)

[0.34906999999999999, 0.54889999999999994, 0.74775999999999998, 0.94459000000000004, 1.1385000000000001, 1.3285, 1.5137]

Type: Matrix(DoubleFloat)

n:=ncols x

(4) 7

Type: PositiveInteger
)clear all

--S 7 of 88
showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 7

--S 8 of 88
showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 8

--S 9 of 88
x:Matrix SF:=
[[0.34907 ,0.54890 ,0.74776 ,0.94459 ,1.13850 ,1.32850 ,1.51370 ]]
--R
--R (3)
--R [0.34906999999999999, 0.54889999999999994, 0.74775999999999998,
--R 0.94459000000000004, 1.1385000000000001, 1.3285, 1.5137]
--R ]
--R Type: Matrix(DoubleFloat)
--E 9

--S 10 of 88
n:=ncols x
--R
--R (4) 7
--R Type: PositiveInteger
--E 10

--S 11 of 88
result:=c06ebf(n,x,-1)
--E 11

)clear all

--S 12 of 88
showArrayValues true
--R
--R
--R (1) true
--E 12

--S 13 of 88
showScalarValues true
--R
--R (2) true
--E 13

--S 14 of 88
x:Matrix SF:=
[[0.34907, 0.54890, 0.74776, 0.94459, 1.13850, 1.32850, 1.51370]]
--R
--R
--R (3)
--R [  
--R [0.34906999999999999, 0.54889999999999994, 0.74775999999999998, 
--R 0.94459000000000004, 1.1385000000000001, 1.3285, 1.5137]
--R ]
--R Type: Matrix(DoubleFloat)
--E 14

--S 15 of 88
y:Matrix SF:=
[[0.37168, -0.35669, -0.31175, -0.23702, -0.13274, 0.00074, 0.16298]]
--R
--R
--R (4)
--R [  
--R [- 0.37167999999999995, - 0.35668999999999995, - 0.31174999999999997, 
--R - 0.23701999999999998, - 0.13274000000000002, 7.3999999999999999E-4, 
--R 0.16298000000000001]
--R ]
--R Type: Matrix(DoubleFloat)
--E 15

--S 16 of 88
n: ncols x
--R
--R (5) 7
--R Type: PositiveInteger
--E 16

--S 17 of 88
result:=c06ecf(n,x,y,-1)
--E 17
)clear all

--S 18 of 88
showArrayValues true
--R
--R  (1) true
--R  Type: Boolean
--E 18

--S 19 of 88
showScalarValues true
--R
--R  (2) true
--R  Type: Boolean
--E 19

--S 20 of 88
x:Matrix SF:= [[1.00 ,1.00 ,1.00 ,1.00 ,1.00 ,0.00 ,0.00 ,0.00 ,0.00 ]]
--R
--R  (3) [1. 1. 1. 1. 1. 0. 0. 0. 0.]
--R  Type: Matrix(DoubleFloat)
--E 20

--S 21 of 88
y:Matrix SF:= [[0.50 ,0.50 ,0.50 ,0.50 ,0.00 ,0.00 ,0.00 ,0.00 ,0.00 ]]
--R
--R  (4) [0.5 0.5 0.5 0.5 0. 0. 0. 0. 0.]
--R  Type: Matrix(DoubleFloat)
--E 21

--S 22 of 88
n:=ncols x
--R
--R  (5) 9
--R  Type: PositiveInteger
--E 22

--S 23 of 88
job:=1
--R
--R  (6) 1
--R  Type: PositiveInteger
--E 23
--S 24 of 88
-- c06ekf(job,n,x,y,-1)
--E 24

--S 25 of 88
--R
job:=2
--R
--R (7) 2
--R Type: PositiveInteger
--E 25

--S 26 of 88
-- result:=c06ekf(job,n,x,y,-1)
--E 26

)clear all

--S 27 of 88
showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 27

--S 28 of 88
showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 28

--S 29 of 88
x:Matrix SF:=
[[0.3854 ,0.6772 ,0.1138 ,0.6751 ,0.6362 ,0.1424] ,
[0.5417 ,0.2983 ,0.1181 ,0.7255 ,0.8638 ,0.8723] ,
[0.9172 ,0.0644 ,0.6037 ,0.6430 ,0.0428 ,0.4815 ]]
--R
--R (3)
--R [ 
--R [0.38539999999999996, 0.67720000000000002, 0.1138, 0.67510000000000003, 
--R 0.63619999999999999, 0.1424]
--R ,
--R [0.54169999999999996, 0.29830000000000001, 0.1181, 0.72550000000000003, 
--R 0.86379999999999999, 0.87229999999999996]
m:=nrows x

n:=ncols x

trig:Matrix SF:=new(1,2*n,0.0@SF)

init:="i"

result:=c06fpf(m,n,init,x,trig,-1)

)clear all

showArrayValues true
--R (1) true
--E 35

--S 36 of 88
showScalarValues true
--R
--R (2) true
--R
--E 36

--S 37 of 88
x:Matrix SF:=
[[0.3854, 0.6772, 0.1138, 0.6751, 0.6362, 0.1424],
 [0.5417, 0.2983, 0.1181, 0.7255, 0.8638, 0.8723],
 [0.9172, 0.0644, 0.6037, 0.6430, 0.0428, 0.4815]]
--R
--R
--R (3)
--R [0.38539999999999996, 0.67720000000000002, 0.1138, 0.67510000000000003,
  0.63619999999999999, 0.1424],
--R ,
--R [0.54169999999999996, 0.29830000000000001, 0.1181, 0.72550000000000003,
  0.8637999999999999, 0.87229999999999996],
--R ,
--R [0.91720000000000002, 6.439999999999999E-2, 0.063699999999999999,
  0.64300000000000002, 4.279999999999999E-2, 0.48149999999999998],
--R ]
--E 37

--S 38 of 88
m:=nrows x
--R
--R
--R (4) 3
--R
--E 38

--S 39 of 88
n:=ncols x
--R
--R
--R (5) 6
--E 39
trig: Matrix SF:=new(1,2*n,0.0@SF)

init:="i"

result:=c06fqf(m,n,init,x,trig,-1)

x: Matrix SF:=
[[0.3854 ,0.6772 ,0.1138 ,0.6751 ,0.6362 ,0.1424],
[0.9172 ,0.0644 ,0.6037 ,0.6430 ,0.0428 ,0.4815],
[0.1156 ,0.0685 ,0.2060 ,0.8630 ,0.6967 ,0.2792 ]]

result:=
[0.38539999999999996, 0.67720000000000002, 0.1138, 0.67510000000000003, 0.63619999999999999, 0.1424]
Type: Matrix(DoubleFloat)

\[
\begin{bmatrix}
0.5417 & 0.2983 & 0.1181 & 0.7255 & 0.8638 & 0.8723 \\
0.9089 & 0.3118 & 0.3465 & 0.6198 & 0.2668 & 0.1614 \\
0.6214 & 0.8681 & 0.7060 & 0.8652 & 0.9190 & 0.3355
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
\begin{bmatrix}
0.5417 & 0.2983 & 0.1181 & 0.7255 & 0.8638 & 0.8723 \\
0.9089 & 0.3118 & 0.3465 & 0.6198 & 0.2668 & 0.1614 \\
0.6214 & 0.8681 & 0.7060 & 0.8652 & 0.9190 & 0.3355
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)

\[
\begin{bmatrix}
0.5417 & 0.2983 & 0.1181 & 0.7255 & 0.8638 & 0.8723 \\
0.9089 & 0.3118 & 0.3465 & 0.6198 & 0.2668 & 0.1614 \\
0.6214 & 0.8681 & 0.7060 & 0.8652 & 0.9190 & 0.3355
\end{bmatrix}
\]

Type: Matrix(DoubleFloat)
trig: Matrix SF := new(1,2*n,0.0@SF)
--R
--R (7) [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
--R Type: Matrix(DoubleFloat)
--E 49

init:="i"
--R
--R (8) "i"
--R Type: String
--E 50

result:=c06frf(m,n,init,x,y,trig,-1)
--E 51

)clear all

showArrayValues true
--R
--R (1) true
--R Type: Boolean
--E 52

showScalarValues true
--R
--R (2) true
--R Type: Boolean
--E 53

x: Matrix SF :=
[[1.000,0.999,0.987,0.936,0.802],
[0.994,0.989,0.963,0.891,0.731],
[0.903,0.885,0.823,0.694,0.467]]
--R
--R (3)
--R [[1.,0.999,0.98699999999999999,0.93599999999999994,0.80200000000000005],
--R
--R [0.99399999999999999, 0.98899999999999999, 0.96299999999999997,
y:Matrix SF:=
[[0.000,-0.040,-0.159,-0.352,-0.597],
[-0.111,-0.151,-0.268,-0.454,-0.682],
[-0.430,-0.466,-0.568,-0.720,-0.884]]

trigm:Matrix SF:=new(1,2*m,0.0@SF)
PACKAGE NAGC06 NAGSERIESSUMMATIONPACKAGE

--R
--R
--R (7) [0. 0. 0. 0. 0. 0.]
--R Type: Matrix(DoubleFloat)
--E 58

--S 59 of 88
trign:Matrix SF:=new(1,2*n,0.0@SF)
--R
--R
--R (8) [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
--R Type: Matrix(DoubleFloat)
--E 59

--S 60 of 88
init:"i"
--R
--R
--R (9) "i"
--R Type: String
--E 60

--S 61 of 88
result:=c06fuf(m,n,init,x,y,trigm,trign,-1)
-- E 61
)
clear all

--S 62 of 88
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
--E 62

--S 63 of 88
showScalarValues true
--R
--R
--R (2) true
--R Type: Boolean
--E 63

--S 64 of 88
x:Matrix SF:=
[[0.34907, 0.54890, 0.74776, 0.94459, 1.13850, 1.32850, 1.51370]]
n := ncols x
---R
---R (4) 7
---R Type: PositiveInteger
---E 65

result := c06gbf(n, x, -1)
---E 66

clear all

showArrayValues true
---R
---R
---R (1) true
---R Type: Boolean
---E 67

showScalarValues true
---R
---R
---R (2) true
---R Type: Boolean
---E 68

y := Matrix SF :=
[[-0.37168, -0.35669, -0.31175, -0.23702, 0.00074, 0.16298, 1.51370]]
---R
---R
---R (3)
---R
---R
---R [- 0.3716799999999995, - 0.3566899999999995, - 0.3117499999999997,
---R - 0.23701999999999998, 7.399999999999999E-4, 0.1629800000000001, 1.5137]
---R
---R Type: Matrix(DoubleFloat)
---E 69
n:=ncols y
--R
--R (4) 7
Type: PositiveInteger
--E 70

result:=c06gcf(n,y,-1)
--E 71
)
clear all

showArrayValues true
--R
--R (1) true
Type: Boolean
--E 72

showScalarValues true
--R
--R (2) true
Type: Boolean
--E 73

y:Matrix SF:=
[[[-0.37168,-0.35669,-0.31175,-0.23702,0.00074,0.16298,1.51370]]
--R
--R (3)
--R [
--R [- 0.3716799999999995, - 0.3566899999999995, - 0.3117499999999997, 
--R - 0.2370199999999998, 7.399999999999999E-4, 0.1629800000000001, 
--R 1.5137]]
--R ]
Type: Matrix(DoubleFloat)
--E 74

n:=ncols y
--R
--R (4) 7
x: Matrix SF :=
[[0.3854, 0.6772, 0.1138, 0.6751, 0.6362, 0.1424],
 [0.5417, 0.2983, 0.1181, 0.7255, 0.8638, 0.8723],
 [0.9172, 0.0644, 0.6037, 0.6430, 0.0428, 0.4815]]

m := nrows x
n:=ncols x

result:=c06gqf(m,n,x,-1)
This package uses the NAG Library to calculate the discrete Fourier transform of a sequence of real or complex data values, and applies it to calculate convolutions and correlations.
1. Scope of the Chapter

This chapter is concerned with calculating the discrete Fourier transform of a sequence of real or complex data values, and applying it to calculate convolutions and correlations.

2. Background to the Problems

2.1. Discrete Fourier Transforms

2.1.1. Complex transforms

Most of the routines in this chapter calculate the finite discrete Fourier transform (DFT) of a sequence of n complex numbers \( z_j \), for \( j=0,1,...,n-1 \). The transform is defined by:

\[
\begin{align*}
  z_k &= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(-i \frac{2\pi jk}{n}\right) \\
  &\text{for } k=0,1,...,n-1.
\end{align*}
\]

Note that equation (1) makes sense for all integral \( k \) and with this extension \( z \) is periodic with period \( n \), i.e. \( z_k = z_{k+n} \), and in particular \( z_k = z_{n-k} \).

If we write \( z_j = x_j + iy_j \) and \( z_k = a_k + ib_k \), then the definition of \( z_j \) may be written in terms of sines and cosines as:

\[
\begin{align*}
  a_k &= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} (x_j \cos\left(\frac{2\pi jk}{n}\right) + y_j \sin\left(\frac{2\pi jk}{n}\right)) \\
  b_k &= \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} (y_j \cos\left(\frac{2\pi jk}{n}\right) - x_j \sin\left(\frac{2\pi jk}{n}\right)).
\end{align*}
\]

The original data values \( z_j \) may conversely be recovered from the
transform \( z \) by an inverse discrete Fourier transform:

\[
\begin{align*}
1 & \quad \frac{1}{\sqrt{n}} \quad 2(\pi jk) \\
\sum_{j=0}^{n-1} z & \exp(+i \frac{2\pi jk}{n}) \quad (2) \\
\end{align*}
\]

for \( j=0,1,\ldots,n-1 \). If we take the complex conjugate of (2), we find that the sequence \( z \) is the DFT of the sequence \( z \). Hence the inverse DFT of the sequence \( z \) may be obtained by: taking the complex conjugates of the \( z \); performing a DFT; and taking the complex conjugates of the result.

Notes: definitions of the discrete Fourier transform vary. Sometimes (2) is used as the definition of the DFT, and (1) as the definition of the inverse. Also the scale-factor of \( 1/\sqrt{n} \) may be omitted in the definition of the DFT, and replaced by \( 1/n \) in the definition of the inverse.

2.1.2. Real transforms

If the original sequence is purely real valued, i.e. \( z = x \), then

\[
\begin{align*}
1 & \quad \frac{1}{\sqrt{n}} \quad 2(\pi jk) \\
\sum_{j=0}^{n-1} x & \exp(-i \frac{2\pi jk}{n}) \\
\end{align*}
\]

and \( z \) is the complex conjugate of \( z \). Thus the DFT of a real sequence is a particular type of complex sequence, called a Hermitian sequence, or half-complex or conjugate symmetric with the properties:
Thus a Hermitian sequence of $n$ complex data values can be represented by only $n$, rather than $2n$, independent real values. This can obviously lead to economies in storage, the following scheme being used in this chapter: the real parts $a$ for $0 \leq k < n/2$ are stored in normal order in the first $n/2+1$ locations of an array $X$ of length $n$; the corresponding non-zero imaginary parts are stored in reverse order in the remaining locations of $X$. In other words, if $X$ is declared with bounds $(0:n-1)$ in the user's (sub)program, the real and imaginary parts of $z$ are stored as follows:

```
if n=2s

X(0)   a  a
      0  0
X(1)   a  a
      1  1
X(2)   a  a
      2  2
.
.
.
.
.
.
X(s-1) a  a
      s-1 s-1
X(s)   a  b
      s  s-1
X(s+1) b  b
      s-1 s-2
.
.
.
.
X(n-2) b  b
```
2.1.3. Fourier integral transforms

The usual application of the discrete Fourier transform is that of obtaining an approximation of the Fourier integral transform

\[
F(s) = \int_{-\infty}^{\infty} f(t) \exp(-i2\pi st) dt
\]

when \( f(t) \) is negligible outside some region \((0,c)\). Dividing the region into \( n \) equal intervals we have

\[
F(s) \approx \sum_{j=0}^{n-1} f \exp(-i2\pi sjc/n)
\]

and so

\[
F(s) \approx \sum_{j=0}^{n-1} f \exp(-i2\pi jk/n)
\]

for \( k=0,1,\ldots,n-1 \), where \( f = f(jc/n) \) and \( F = F(k/c) \).

Hence the discrete Fourier transform gives an approximation to the Fourier integral transform in the region \( s=0 \) to \( s=n/c \).
If the function \( f(t) \) is defined over some more general interval \((a,b)\), then the integral transform can still be approximated by the discrete transform provided a shift is applied to move the point \( a \) to the origin.

2.1.4. Convolutions and correlations

One of the most important applications of the discrete Fourier transform is to the computation of the discrete convolution or correlation of two vectors \( x \) and \( y \) defined (as in Brigham [1]) by:

\[
\begin{align*}
\text{convolution: } z & = \sum_{k=0}^{n-1} x_k y_{k-j} \\
\text{correlation: } w & = \sum_{k=0}^{n-1} x_k y_{k+j}
\end{align*}
\]

(Here \( x \) and \( y \) are assumed to be periodic with period \( n \).)

Under certain circumstances (see Brigham [1]) these can be used as approximations to the convolution or correlation integrals defined by:

\[
\begin{align*}
\text{convolution integral: } z(s) & = \int_{-\infty}^{\infty} x(t) y(s-t) \, dt \\
\text{correlation integral: } w(s) & = \int_{-\infty}^{\infty} x(t) y(s+t) \, dt, \quad -\infty < s < +\infty.
\end{align*}
\]

For more general advice on the use of Fourier transforms, see Hamming [2]; more detailed information on the fast Fourier transform algorithm can be found in Van Loan [3] and Brigham [1].

2.2. References
3. Recommendations on Choice and Use of Routines

3.1. One-dimensional Fourier Transforms

The choice of routine is determined first of all by whether the data values constitute a real, Hermitian or general complex sequence. It is wasteful of time and storage to use an inappropriate routine.

Two groups, each of three routines, are provided

<table>
<thead>
<tr>
<th>Group 1</th>
<th>Group 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real sequences</td>
<td>C06EAF</td>
</tr>
<tr>
<td>Hermitian sequences</td>
<td>C06EBF</td>
</tr>
<tr>
<td>General complex sequences</td>
<td>C06ECF</td>
</tr>
</tbody>
</table>

Group 1 routines each compute a single transform of length n, without requiring any extra working storage. The Group 1 routines impose some restrictions on the value of n, namely that no prime factor of n may exceed 19 and the total number of prime factors (including repetitions) may not exceed 20 (though the latter restriction only becomes relevant when n>10).

Group 2 routines are designed to perform several transforms in a single call, all with the same value of n. They do however require more working storage. Even on scalar processors, they may be somewhat faster than repeated calls to Group 1 routines because of reduced overheads and because they pre-compute and store the required values of trigonometric functions. Group 2 routines impose no practical restrictions on the value of n; however the fast Fourier transform algorithm ceases to be 'fast' if applied to values of n which cannot be expressed as a product of small prime factors. All the above routines are particularly efficient if the only prime factors of n are 2, 3 or 5.

If extensive use is to be made of these routines, users who are
concerned about efficiency are advised to conduct their own
timing tests.

To compute inverse discrete Fourier transforms the above routines
should be used in conjunction with the utility routines C06GBF,
C06GCF and C06GQF which form the complex conjugate of a Hermitian
or general sequence of complex data values.

3.2. Multi-dimensional Fourier Transforms

C06FUF computes a 2-dimensional discrete Fourier transform of a
2-dimensional sequence of complex data values. This is defined by

\[
\sum_{k=0}^{n-1} \sum_{j=0}^{n-1} z_{j,k} e^{-i \frac{2\pi j k}{n}} e^{-i \frac{2\pi j k}{n}}
\]

3.3. Convolution and Correlation

C06EKF computes either the discrete convolution or the discrete
Correlation of two real vectors.

3.4. Index

Complex conjugate,
complex sequence C06GCF
Hermitian sequence C06GBF
multiple Hermitian sequences C06GQF
Complex sequence from Hermitian sequences C06GSF
Convolution or Correlation
real vectors C06EKF
Discrete Fourier Transform
two-dimensional
complex sequence C06FUF
one-dimensional, multiple transforms
complex sequence C06FRF
Hermitian sequence C06FQF
real sequence C06FPF
one-dimensional, single transforms
complex sequence C06ECF
Hermitian sequence C06EBF
real sequence C06EAF
Chapter C06

Summation of Series

C06EAF Single 1-D real discrete Fourier transform, no extra workspace

C06EBF Single 1-D Hermitian discrete Fourier transform, no extra workspace

C06ECF Single 1-D complex discrete Fourier transform, no extra workspace

C06EKF Circular convolution or correlation of two real vectors, no extra workspace

C06FPF Multiple 1-D real discrete Fourier transforms

C06FQF Multiple 1-D Hermitian discrete Fourier transforms

C06FRF Multiple 1-D complex discrete Fourier transforms

C06FUF 2-D complex discrete Fourier transform

C06GBF Complex conjugate of Hermitian sequence

C06GCF Complex conjugate of complex sequence

C06GQF Complex conjugate of multiple Hermitian sequences

C06GSF Convert Hermitian sequences to general complex sequences

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06EAF calculates the discrete Fourier transform of a sequence of n real data values. (No extra workspace required.)

2. Specification
SUBROUTINE C06EAF (X, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N)

3. Description

Given a sequence of n real data values x, for \( j=0,1,...,n-1 \),

this routine calculates their discrete Fourier transform defined by:

\[
\begin{align*}
\frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp(-i \frac{2\pi ij}{n}), \quad k=0,1,...,n-1.
\end{align*}
\]

(Note the scale factor of \( \frac{1}{\sqrt{n}} \) in this definition.) The

\[
\frac{1}{\sqrt{n}}
\]

transformed values \( z \) are complex, but they form a Hermitian

\[
\frac{1}{\sqrt{n}}
\]

sequence (i.e., \( z_k \) is the complex conjugate of \( z_{n-k} \), so they are

\[
\frac{1}{\sqrt{n}}
\]

completely determined by \( n \) real numbers (see also the Chapter

Introduction).

To compute the inverse discrete Fourier transform defined by:

\[
\begin{align*}
\frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} x_j \exp(i \frac{2\pi ij}{n}),
\end{align*}
\]

this routine should be followed by a call of C06GBF to form the

\[
\frac{1}{\sqrt{n}}
\]

complex conjugates of the \( z_k \).

The routine uses the fast Fourier transform (FFT) algorithm

(B Brigham [1]). There are some restrictions on the value of \( n \) (see

Section 5).

4. References
CHAPTER 15. CHAPTER N


5. Parameters

1: X(N) -- DOUBLE PRECISION array    Input/Output
On entry: if X is declared with bounds (0:N-1) in the (sub)program from which C06EAF is called, then X(j) must contain x^j, for j=0,1,...,n-1. On exit: the discrete Fourier transform stored in Hermitian form. If the components of the transform z are written as a +ib, and if X is declared with bounds (0:N-1) in the (sub)program from which C06EAF is called, then for 0<=k<=n/2, a is contained in X(k), and for k 1<=k<=(n-1)/2, b is contained in X(n-k). (See also Section 2.1.2 of the Chapter Introduction, and the Example Program.)

2: N -- INTEGER    Input
On entry: the number of data values, n. The largest prime factor of N must not exceed 19, and the total number of prime factors of N, counting repetitions, must not exceed 20. Constraint: N > 1.

3: IFAIL -- INTEGER    Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one of the prime factors of N is greater than 19.

IFAIL= 2
N has more than 20 prime factors.

IFAIL= 3
N <= 1.

7. Accuracy
Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to \( n\log n \), but also depends on the factorization of \( n \). The routine is somewhat faster than average if the only prime factors of \( n \) are 2, 3 or 5; and fastest of all if \( n \) is a power of 2.

On the other hand, the routine is particularly slow if \( n \) has several unpaired prime factors, i.e., if the 'square-free' part of \( n \) has several factors. For such values of \( n \), routine C06FAF(*) (which requires an additional \( n \) elements of workspace) is considerably faster.

9. Example

This program reads in a sequence of real data values, and prints their discrete Fourier transform (as computed by C06EAF), after expanding it from Hermitian form into a full complex sequence.

It then performs an inverse transform using C06GBF and C06EBF, and prints the sequence so obtained alongside the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

1. Purpose

C06EBF calculates the discrete Fourier transform of a Hermitian sequence of \( n \) complex data values. (No extra workspace required.)

2. Specification
SUBROUTINE C06EBF (X, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N)

3. Description

Given a Hermitian sequence of n complex data values \( z \) (i.e., a sequence such that \( z \) is real and \( z_j \) is the complex conjugate of \( z \), for \( j=1,2,\ldots,n-1 \)) this routine calculates their discrete Fourier transform defined by:

\[
x_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(-i \frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1.
\]

(Note the scale factor of \( \frac{1}{\sqrt{n}} \) in this definition.) The transformed values \( x_k \) are purely real (see also the the Chapter Introduction).

To compute the inverse discrete Fourier transform defined by:

\[
y_j = \frac{1}{\sqrt{n}} \sum_{k=0}^{n-1} x_k \exp\left(+i \frac{2\pi jk}{n}\right), \quad j=0,1,\ldots,n-1.
\]

this routine should be preceded by a call of C06GBF to form the complex conjugates of the \( z_j \).

4. References

5. Parameters

1: \( X(N) \) -- DOUBLE PRECISION array \quad \text{Input/Output}

On entry: the sequence to be transformed stored in Hermitian form. If the data values \( z \) are written as \( x + iy \),

\[
X(j) = \begin{array}{ll}
  & \text{for } 0 \leq j \leq n/2, \quad x \\
  & \text{for } 1 \leq j \leq (n-1)/2, \quad y
\end{array}
\]

and if \( X \) is declared with bounds \((0:N-1)\) in the subroutine from which C06EBF is called, then for \( 0 \leq j \leq n/2 \), \( x \) is contained in \( X(j) \), and for \( 1 \leq j \leq (n-1)/2 \), \( y \) is contained in \( X(n-j) \). (See also Section 2.1.2 of the Chapter Introduction and the Example Program.) On exit: the components of the discrete Fourier transform \( x \). If \( X \) is declared with bounds \( (0:N-1) \) in the (sub)program from which C06EBF is called, then \( x \) is stored in \( X(k) \), for \( k=0,1,\ldots,n-1 \).

2: \( N \) -- INTEGER \quad \text{Input}

On entry: the number of data values, \( n \). The largest prime factor of \( N \) must not exceed 19, and the total number of prime factors of \( N \), counting repetitions, must not exceed 20. Constraint: \( N > 1 \).

3: \( IFAIL \) -- INTEGER \quad \text{Input/Output}

On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
At least one of the prime factors of \( N \) is greater than 19.

IFAIL= 2
\( N \) has more than 20 prime factors.

IFAIL= 3
\( N \leq 1 \).
7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to n*logn, but also depends on the factorization of n. The routine is somewhat faster than average if the only prime factors of n are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has several unpaired prime factors, i.e., if the 'square-free' part of n has several factors. For such values of n, routine C06FBF(*) (which requires an additional n elements of workspace) is considerably faster.

9. Example

This program reads in a sequence of real data values which is assumed to be a Hermitian sequence of complex data values stored in Hermitian form. The input sequence is expanded into a full complex sequence and printed alongside the original sequence. The discrete Fourier transform (as computed by C06EBF) is printed out.

The program then performs an inverse transform using C06EAF and C06GBF, and prints the sequence so obtained alongside the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
C06ECF calculates the discrete Fourier transform of a sequence of n complex data values. (No extra workspace required.)

2. Specification

SUBROUTINE C06ECF (X, Y, N, IFAIL)
INTEGER N, IFAIL
DOUBLE PRECISION X(N), Y(N)

3. Description

Given a sequence of n complex data values \( z_j \) for \( j=0,1,\ldots,n-1 \), this routine calculates their discrete Fourier transform defined by:

\[
\hat{z}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(\frac{-2\pi i jk}{n}\right), \quad k=0,1,\ldots,n-1.
\]

(Note the scale factor of \( \frac{1}{\sqrt{n}} \) in this definition.)

1/
\sqrt{n}

To compute the inverse discrete Fourier transform defined by:

\[
\hat{z}_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} w_j \exp\left(\frac{2\pi i jk}{n}\right), \quad k=0,1,\ldots,n-1.
\]

this routine should be preceded and followed by calls of C06GCF to form the complex conjugates of the \( z_j \) and the \( \hat{z}_k \).

The routine uses the fast Fourier transform (FFT) algorithm (Brigham [1]). There are some restrictions on the value of \( n \) (see Section 5).

4. References

5. Parameters

1: \( X(N) \) -- DOUBLE PRECISION array

   On entry: if \( X \) is declared with bounds \((0:N-1)\) in the (sub)
   program from which C06ECF is called, then \( X(j) \) must contain
   \( x_j \), the real part of \( z_j \), for \( j=0,1,...,n-1 \). On exit: the
   \( j \)
   real parts \( a_k \) of the components of the discrete Fourier
   \( k \)
   transform. If \( X \) is declared with bounds \((0:N-1)\) in the (sub)
   program from which C06ECF is called, then \( a_k \) is contained in
   \( k \)
   \( X(k) \), for \( k=0,1,...,n-1 \).

2: \( Y(N) \) -- DOUBLE PRECISION array

   On entry: if \( Y \) is declared with bounds \((0:N-1)\) in the (sub)
   program from which C06ECF is called, then \( Y(j) \) must contain
   \( j \)
   \( y_j \), the imaginary part of \( z_j \), for \( j=0,1,...,n-1 \). On exit: the
   imaginary parts \( b_k \) of the components of the discrete Fourier
   \( k \)
   transform. If \( Y \) is declared with bounds \((0:N-1)\) in the (sub)
   program from which C06ECF is called, then \( b_k \) is
   contained in \( Y(k) \), for \( k=0,1,...,n-1 \).

3: \( N \) -- INTEGER

   On entry: the number of data values, \( n \). The largest prime
   factor of \( N \) must not exceed 19, and the total number of
   prime factors of \( N \), counting repetitions, must not exceed
   20. Constraint: \( N > 1 \).

4: \( IFAIL \) -- INTEGER

   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   At least one of the prime factors of \( N \) is greater than 19.

IFAIL= 2
   \( N \) has more than 20 prime factors.
IFAIL = 3
N <= 1.

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to n*logn, but also depends on the factorization of n. The routine is somewhat faster than average if the only prime factors of n are 2, 3 or 5; and fastest of all if n is a power of 2.

On the other hand, the routine is particularly slow if n has several unpaired prime factors, i.e., if the 'square-free' part of n has several factors. For such values of n, routine C06FCF(*) (which requires an additional n real elements of workspace) is considerably faster.

9. Example

This program reads in a sequence of complex data values and prints their discrete Fourier transform.

It then performs an inverse transform using C06GCF and C06ECF, and prints the sequence so obtained alongside the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
real vectors of period n. No extra workspace is required.

2. Specification

SUBROUTINE C06EKF (JOB, X, Y, N, IFAIL)
INTEGER JOB, N, IFAIL
DOUBLE PRECISION X(N), Y(N)

3. Description

This routine computes:

if JOB =1, the discrete convolution of x and y, defined by:

\[
\sum_{j=0}^{n-1} x_{j+k} y_j = z_k
\]

if JOB =2, the discrete correlation of x and y defined by:

\[
\sum_{j=0}^{n-1} x_{j+k} y_{j+k} = w_{j+k}
\]

Here x and y are real vectors, assumed to be periodic, with period n, i.e., \( x = x = x = \ldots \); \( z \) and \( w \) are then also periodic with period n.

Note: this usage of the terms 'convolution' and 'correlation' is taken from Brigham [1]. The term 'convolution' is sometimes used to denote both these computations.

If x, y, z and w are the discrete Fourier transforms of these sequences,

\[
\sum_{j=0}^{n-1} x_j \exp(-i2\pi jk/n)
\]

then \( z = \sum_{j=0}^{n-1} x_j y_j \)
and \( w = \overline{\sum_{k} x y} \)

(the bar denoting complex conjugate).

This routine calls the same auxiliary routines as \texttt{C06EAF} and \texttt{C06EBF} to compute discrete Fourier transforms, and there are some restrictions on the value of \( n \).

4. References


5. Parameters

1: \texttt{JOB} -- INTEGER Input
   On entry: the computation to be performed:
   \[
   \begin{align*}
   n-1 \\
   - n-1
   \end{align*}
   \]

   --

   if \( \text{JOB} = 1 \), \( z = \sum_{k} x y \) (convolution);
   \[
   \begin{align*}
   n-1 \\
   - n-1
   \end{align*}
   \]

   if \( \text{JOB} = 2 \), \( w = \sum_{k} x y \) (correlation).
   \[
   \begin{align*}
   n-1 \\
   - n-1
   \end{align*}
   \]

   --

   Constraint: \( \text{JOB} = 1 \) or 2.

2: \texttt{X(N)} -- DOUBLE PRECISION array Input/Output
   On entry: the elements of one period of the vector \( x \). If \( X \) is declared with bounds \((0:N-1)\) in the (sub)program from which \texttt{C06EKF} is called, then \( X(j) \) must contain \( x \), for \( j=0,1,...,n-1 \).
   On exit: the corresponding elements of the discrete convolution or correlation.

3: \texttt{Y(N)} -- DOUBLE PRECISION array Input/Output
   On entry: the elements of one period of the vector \( y \). If \( Y \) is declared with bounds \((0:N-1)\) in the (sub)program from which \texttt{C06EKF} is called, then \( Y(j) \) must contain \( y \), for \( j=0,1,...,n-1 \).
   On exit: the discrete Fourier transform of the convolution or correlation returned in the array \( X \); the
transform is stored in Hermitian form, exactly as described in the document C06EAF.

4: N -- INTEGER Input
On entry: the number of values, n, in one period of the vectors X and Y. The largest prime factor of N must not exceed 19, and the total number of prime factors of N, counting repetitions, must not exceed 20. Constraint: N > 1.

5: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

IFAIL= 1
At least one of the prime factors of N is greater than 19.

IFAIL= 2
N has more than 20 prime factors.

IFAIL= 3
N <= 1.

IFAIL= 4
JOB /= 1 or 2.

7. Accuracy
The results should be accurate to within a small multiple of the machine precision.

8. Further Comments
The time taken by the routine is approximately proportional to n*logn, but also depends on the factorization of n. The routine is faster than average if the only prime factors are 2, 3 or 5; and fastest of all if n is a power of 2.

The routine is particularly slow if n has several unpaired prime factors, i.e., if the 'square free' part of n has several factors. For such values of n, routine C06FKF(*) is considerably faster (but requires an additional workspace of n elements).
9. Example

This program reads in the elements of one period of two real vectors x and y and prints their discrete convolution and correlation (as computed by C06EKF). In realistic computations the number of data values would be much larger.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

SUBROUTINE C06FPF (M, N, X, INIT, TRIG, WORK, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N), TRIG(2*N), WORK(M*N)
CHARACTER*1 INIT

Given m sequences of n real data values x , for j=0,1,...,n-1; j
p=1,2,...,m, this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

\[ z = \sum_{k=0}^{n-1} x_k \exp(-i \frac{2\pi j k}{n}), \quad k=0,1,...,n-1; \quad p=1,2,...,m. \]
The transformed values $z$ are complex, but for each value of $p$ the $z^p$ form a Hermitian sequence (i.e., $z^p$ is the complex conjugate of $z$), so they are completely determined by $m n$ real numbers (see also the Chapter Introduction).

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term:

$$
\frac{1}{n} \sum_{j=0}^{n-1} x^j \exp(i \frac{2\pi}{n} jk)
$$

To compute this form, this routine should be followed by a call to C06GQF to form the complex conjugates of the $z$.

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special coding is provided for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as $M$, the number of transforms to be computed in parallel, increases.

4. References


5. Parameters

1: M -- INTEGER Input
On entry: the number of sequences to be transformed, m.  
Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of real values in each sequence, n.  
Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
On entry: the data must be stored in X as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of the array. In other words, the data values of the pth sequence to be transformed are denoted by x

Then the mn elements of the array X must contain the values

```
1 2  m 1 2  m 1 2  m
x , x ,...,x , x ,..., x ,...,x ,...,x .
0 0 0 1 1 1  n-1 n-1 n-1
```

On exit: the m discrete Fourier transforms stored as if in a two-dimensional array of dimension (1:M,0:N-1). Each of the m transforms is stored in a row of the array in Hermitian form, overwriting the corresponding original sequence. If the n components of the discrete Fourier transform \( z^p \) are written as \( a_k + ib_k \), then for \( 0 \leq k \leq n/2 \), \( a_k \) is contained in \( X(p,k) \), and for \( 1 \leq k \leq (n-1)/2 \), \( b_k \) is contained in \( X(p,n-k) \). (See also Section 2.1.2 of the Chapter Introduction.)

4: INIT -- CHARACTER*1 Input
On entry: if the trigonometric coefficients required to compute the transforms are to be calculated by the routine and stored in the array TRIG, then INIT must be set equal to 'I' (Initial call).

If INIT contains 'S' (Subsequent call), then the routine assumes that trigonometric coefficients for the specified value of n are supplied in the array TRIG, having been calculated in a previous call to one of C06FPF, C06FQF or C06FRF.

If INIT contains 'R' (Restart then the routine assumes that trigonometric coefficients for the particular value of n are supplied in the array TRIG, but does not check that C06FPF, C06FQF or C06FRF have previously been called. This option allows the TRIG array to be stored in an external file, read
in and re-used without the need for a call with INIT equal to 'I'. The routine carries out a simple test to check that the current value of n is consistent with the array TRIG. Constraint: INIT = 'I', 'S' or 'R'.

5: TRIG(2*N) -- DOUBLE PRECISION array Input/Output
On entry: if INIT = 'S' or 'R', TRIG must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIG need not be set. On exit: TRIG contains the required coefficients (computed by the routine if INIT = 'I').

6: WORK(M*N) -- DOUBLE PRECISION array Workspace

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry M < 1.

IFAIL = 2
N < 1.

IFAIL = 3
INIT is not one of 'I', 'S' or 'R'.

IFAIL = 4
INIT = 'S', but none of C06FPF, C06FQF or C06FRF has previously been called.

IFAIL = 5
INIT = 'S' or 'R', but the array TRIG and the current value of N are inconsistent.

7. Accuracy

Some indication of accuracy can be obtained by performing a
subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to $mn \log n$, but also depends on the factors of $n$. The routine is fastest if the only prime factors of $n$ are 2, 3 and 5, and is particularly slow if $n$ is a large prime, or has large prime factors.

9. Example

This program reads in sequences of real data values and prints their discrete Fourier transforms (as computed by C06FPF). The Fourier transforms are expanded into full complex form using C06GSF and printed. Inverse transforms are then calculated by calling C06GQF followed by C06FQF showing that the original sequences are restored.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Given \( m \) Hermitian sequences of \( n \) complex data values \( z_j \), for \( j=0,1,\ldots,n-1; \ p=1,2,\ldots,m \), this routine simultaneously calculates the Fourier transforms of all the sequences defined by:

\[
x_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(-i \frac{2\pi jk}{n}\right), \quad k=0,1,\ldots,n-1; \ p=1,2,\ldots,m.
\]

(Note the scale factor \( \frac{1}{\sqrt{n}} \) in this definition.)

The transformed values are purely real (see also the Chapter Introduction).

The discrete Fourier transform is sometimes defined using a positive sign in the exponential term

\[
x_k = \frac{1}{\sqrt{n}} \sum_{j=0}^{n-1} z_j \exp\left(+i \frac{2\pi jk}{n}\right),
\]

To compute this form, this routine should be preceded by a call to \( \text{C06GQF} \) to form the complex conjugates of the \( z_j \).

The routine uses a variant of the fast Fourier transform (FFT) algorithm (Brigham [1]) known as the Stockham self-sorting algorithm, which is described in Temperton [2]. Special code is included for the factors 2, 3, 4, 5 and 6. This routine is designed to be particularly efficient on vector processors, and it becomes especially fast as \( m \), the number of transforms to be computed in parallel, increases.

4. References


5. Parameters

1: M -- INTEGER Input
   On entry: the number of sequences to be transformed, m.
   Constraint: M >= 1.

2: N -- INTEGER Input
   On entry: the number of data values in each sequence, n.
   Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
   On entry: the data must be stored in X as if in a two-
   dimensional array of dimension (1:M,0:N-1); each of the m
   sequences is stored in a row of the array in Hermitian form.
   \[ \text{If the n data values } z_j \text{ are written as } x_j +iy_j, \text{ then for} \]
   \[ 0 \leq j \leq n/2, x_j \text{ is contained in } X(p,j), \text{ and for } 1 \leq j \leq (n-1)/2, \]
   \[ y_j \text{ is contained in } X(p,n-j). \text{(See also Section 2.1.2 of the} \]
   \[ \text{Chapter Introduction.) On exit: the components of the m} \]
   \[ \text{discrete Fourier transforms, stored as if in a two-} \]
   \[ \text{dimensional array of dimension (1:M,0:N-1). Each of the m} \]
   \[ \text{transforms is stored as a row of the array, overwriting} \]
   \[ \text{the corresponding original sequence. If the n components of the} \]
   \[ \text{p} \]
   \[ \text{discrete Fourier transform are denoted by } x_k, \text{ for} \]
   \[ k=0,1,...,n-1, \text{ then the mn elements of the array } X \text{ contain} \]
   \[ \text{the values} \]
   \[ x_0, x_1, ..., x_{n-1}, x_0, x_1, ..., x_{n-1}, x_0, x_1, ..., x_{n-1}. \]

4: INIT -- CHARACTER*1 Input
   On entry: if the trigonometric coefficients required to
   compute the transforms are to be calculated by the routine
   and stored in the array TRIG, then INIT must be set equal to
   'I' (Initial call).
   If INIT contains 'S' (Subsequent call), then the routine
   assumes that trigonometric coefficients for the specified
   value of n are supplied in the array TRIG, having been
   calculated in a previous call to one of C06FPF, C06FQF or
If INIT contains 'R' (Restart), then the routine assumes that trigonometric coefficients for the particular value of N are supplied in the array TRIG, but does not check that C06FPF, C06FQF or C06FRF have previously been called. This option allows the TRIG array to be stored in an external file, read in and re-used without the need for a call with INIT equal to 'I'. The routine carries out a simple test to check that the current value of n is compatible with the array TRIG. Constraint: INIT = 'I', 'S' or 'R'.

5: TRIG(2*N) -- DOUBLE PRECISION array Input/Output
   On entry: if INIT = 'S' or 'R', TRIG must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIG need not be set. On exit: TRIG contains the required coefficients (computed by the routine if INIT = 'I').

6: WORK(M*N) -- DOUBLE PRECISION array Workspace

7: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry M < 1.

IFAIL= 2
   On entry N < 1.

IFAIL= 3
   On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL= 4
   On entry INIT = 'S', but none of C06FPF, C06FQF and C06FRF has previously been called.

IFAIL= 5
On entry INIT = 'S' or 'R', but the array TRIG and the current value of n are inconsistent.

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to nm*logn, but also depends on the factors of n. The routine is fastest if the only prime factors of n are 2, 3 and 5, and is particularly slow if n is a large prime, or has large prime factors.

9. Example

This program reads in sequences of real data values which are assumed to be Hermitian sequences of complex data stored in Hermitian form. The sequences are expanded into full complex form using C06GSF and printed. The discrete Fourier transforms are then computed (using C06FQF) and printed out. Inverse transforms are then calculated by calling C06FPF followed by C06GQF showing that the original sequences are restored.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
SUBROUTINE C06FRF (M, N, X, Y, INIT, TRIG, WORK, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N), Y(M*N), TRIG(2*N), WORK(2*M*N)
CHARACTER*1 INIT

3. Description

Given m sequences of n complex data values $z_j$, for $j=0,1,...,n-1$;

$p=1,2,...,m$, this routine simultaneously calculates the Fourier
transforms of all the sequences defined by:

\[
\sum_{k=0}^{n-1} \frac{1}{n} z_k \exp(-i \frac{2\pi jk}{n}), \quad k=0,1,...,n-1; \quad p=1,2,...,m.
\]

(Note the scale factor \( \frac{1}{n} \) in this definition.)

The discrete Fourier transform is sometimes defined using a
positive sign in the exponential term

\[
\sum_{k=0}^{n-1} \frac{1}{n} z_k \exp(+i \frac{2\pi jk}{n}), \quad k=0,1,...,n-1; \quad p=1,2,...,m.
\]

To compute this form, this routine should be preceded and
followed by a call of C06GCF to form the complex conjugates of

$p$  
the $z$ and the $z$.

The routine uses a variant of the fast Fourier transform (FFT)
algorithm (Brigham [1]) known as the Stockham self-sorting
algorithm, which is described in Temperton [2]. Special code is
provided for the factors 2, 3, 4, 5 and 6. This routine is
designed to be particularly efficient on vector processors, and
it becomes especially fast as m, the number of transforms to be
computed in parallel, increases.

4. References
5. Parameters

1: M -- INTEGER Input
   On entry: the number of sequences to be transformed, m.
   Constraint: M >= 1.

2: N -- INTEGER Input
   On entry: the number of complex values in each sequence, n.
   Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output

4: Y(M,N) -- DOUBLE PRECISION array Input/Output
   On entry: the real and imaginary parts of the complex data
   must be stored in X and Y respectively as if in a two-
   dimensional array of dimension (1:M,0:N-1); each of the m
   sequences is stored in a row of each array. In other words,
   if the real parts of the pth sequence to be transformed are
   denoted by x_p, for j=0,1,...,n-1, then the mn elements of
   the array X must contain the values
   \[ x_0, x_1, \ldots, x_n, x_0, x_1, \ldots, x_n, \ldots, x_0, x_1, \ldots, x_n \]
   On exit: X and Y are overwritten by the real and imaginary
   parts of the complex transforms.

5: INIT -- CHARACTER*1 Input
   On entry: if the trigonometric coefficients required to
   compute the transforms are to be calculated by the routine
   and stored in the array TRIG, then INIT must be set equal to
   'I' (Initial call).

   If INIT contains 'S' (Subsequent call), then the routine
   assumes that trigonometric coefficients for the specified
   value of n are supplied in the array TRIG, having been
   calculated in a previous call to one of C06FPF, C06FQF or
   C06FRF.

   If INIT contains 'R' (Restart) then the routine assumes that
   trigonometric coefficients for the particular value of n are
   supplied in the array TRIG, but does not check that C06FPF,
CO6FQF or CO6FRF have previously been called. This option allows the TRIG array to be stored in an external file, read in and re-used without the need for a call with INIT equal to 'I'. The routine carries out a simple test to check that the current value of n is compatible with the array TRIG. Constraint: INIT = 'I', 'S' or 'R'.

6: TRIG(2*N) -- DOUBLE PRECISION array Input/Output
   On entry: if INIT = 'S' or 'R', TRIG must contain the required coefficients calculated in a previous call of the routine. Otherwise TRIG need not be set. On exit: TRIG contains the required coefficients (computed by the routine if INIT = 'I').

7: WORK(2*M*N) -- DOUBLE PRECISION array Workspace

8: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   On entry M < 1.

IFAIL = 2
   On entry N < 1.

IFAIL = 3
   On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL = 4
   On entry INIT = 'S', but none of CO6FPF, CO6FQF and CO6FRF has previously been called.

IFAIL = 5
   On entry INIT = 'S' or 'R', but the array TRIG and the current value of n are inconsistent.

7. Accuracy
Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to \( \text{nm} \log n \), but also depends on the factors of \( n \). The routine is fastest if the only prime factors of \( n \) are 2, 3 and 5, and is particularly slow if \( n \) is a large prime, or has large prime factors.

9. Example

This program reads in sequences of complex data values and prints their discrete Fourier transforms (as computed by C06FRF). Inverse transforms are then calculated using C06GCF and C06FRF and printed out, showing that the original sequences are restored.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
3. Description

This routine computes the two-dimensional discrete Fourier transform of a bivariate sequence of complex data values $z_{j,k}$, where $j = 0,1,\ldots,m-1$, $k = 0,1,\ldots,n-1$.

The discrete Fourier transform is here defined by:

$$z_{j,k} = \frac{1}{\sqrt{mn}} \sum_{j=0}^{m-1} \sum_{k=0}^{n-1} z_{j,k} \exp(-2\pi i \left( \frac{j}{m} + \frac{k}{n} \right)),$$

where $k = 0,1,\ldots,m-1$, $j = 0,1,\ldots,n-1$.

(Note the scale factor of $\frac{1}{\sqrt{mn}}$ in this definition.)

4. References


5. Parameters

1: M -- INTEGER

Input
On entry: the number of rows, $m$, of the arrays $X$ and $Y$.
Constraint: $M \geq 1$.

2: $N$ -- INTEGER  Input
On entry: the number of columns, $n$, of the arrays $X$ and $Y$.
Constraint: $N \geq 1$.

3: $X( M,N )$ -- DOUBLE PRECISION array  Input/Output
On entry: the real and imaginary parts of the complex data
values must be stored in arrays $X$ and $Y$ respectively. If $X$
and $Y$ are regarded as two-dimensional arrays of dimension
$(0:M-1,0:N-1)$, then $X(j, j)$ and $Y(j, j)$ must contain the
real and imaginary parts of $z$. On exit: the real and
imaginary parts respectively of the corresponding elements
of the computed transform.

4: $Y( M,N )$ -- DOUBLE PRECISION array  Input/Output
On entry: the real and imaginary parts of the complex data
values must be stored in arrays $X$ and $Y$ respectively. If $X$
and $Y$ are regarded as two-dimensional arrays of dimension
$(0:M-1,0:N-1)$, then $X(j, j)$ and $Y(j, j)$ must contain the
real and imaginary parts of $z$. On exit: the real and
imaginary parts respectively of the corresponding elements
of the computed transform.

5: $INIT$ -- CHARACTER*1  Input
On entry: if the trigonometric coefficients required to
compute the transforms are to be calculated by the routine
and stored in the arrays $TRIGM$ and $TRIGN$, then $INIT$ must be
set equal to 'I', (Initial call).

If $INIT$ contains 'S', (Subsequent call), then the routine
assumes that trigonometric coefficients for the specified
values of $m$ and $n$ are supplied in the arrays $TRIGM$ and
$TRIGN$, having been calculated in a previous call to the
routine.

If $INIT$ contains 'R', (Restart), then the routine assumes
that trigonometric coefficients for the particular values of
$m$ and $n$ are supplied in the arrays $TRIGM$ and $TRIGN$, but does
not check that the routine has previously been called. This
option allows the $TRIGM$ and $TRIGN$ arrays to be stored in an
external file, read in and re-used without the need for a
call with $INIT$ equal to 'I'. The routine carries out a
simple test to check that the current values of $m$ and $n$ are
compatible with the arrays $TRIGM$ and $TRIGN$. Constraint: $INIT$
= 'I', 'S' or 'R'.

6: $TRIGM( 2*M )$ -- DOUBLE PRECISION array  Input/Output

7: $TRIGN( 2*N )$ -- DOUBLE PRECISION array  Input/Output
On entry: if $INIT$ = 'S' or 'R', $TRIGM$ and $TRIGN$ must contain
the required coefficients calculated in a previous call of
the routine. Otherwise $TRIGM$ and $TRIGN$ need not be set.
If \( m=n \) the same array may be supplied for TRIGM and TRIGN. On exit: TRIGM and TRIGN contain the required coefficients (computed by the routine if INIT = 'I').

8: WORK(2*M*N) -- DOUBLE PRECISION array Workspace

9: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AUF).

IFAIL = 1
On entry \( M < 1 \).

IFAIL = 2
On entry \( N < 1 \).

IFAIL = 3
On entry INIT is not one of 'I', 'S' or 'R'.

IFAIL = 4
On entry INIT = 'S', but C06FUF has not previously been called.

IFAIL = 5
On entry INIT = 'S' or 'R', but at least one of the arrays TRIGM and TRIGN is inconsistent with the current value of \( M \) or \( N \).

7. Accuracy

Some indication of accuracy can be obtained by performing a subsequent inverse transform and comparing the results with the original sequence (in exact arithmetic they would be identical).

8. Further Comments

The time taken by the routine is approximately proportional to \( mn \log(mn) \), but also depends on the factorization of the
individual dimensions \( m \) and \( n \). The routine is somewhat faster than average if their only prime factors are 2, 3 or 5; and fastest of all if they are powers of 2.

9. Example

This program reads in a bivariate sequence of complex data values and prints the two-dimensional Fourier transform. It then performs an inverse transform and prints the sequence so obtained, which may be compared to the original data values.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1: X(N) -- DOUBLE PRECISION array Input/Output
On entry: if the data values z are written as x +iy and
               j
if X is declared with bounds (0:N-1) in the (sub)program
from which C06GBF is called, then for 0<=j<=n/2, X(j) must
contain x (=x ), while for n/2<j<=n-1, X(j) must contain
               j  n-j
-y (=y ). In other words, X must contain the Hermitian
               j  n-j
sequence in Hermitian form. (See also Section 2.1.2 of the
Chapter Introduction). On exit: the imaginary parts y are
               j
negated. The real parts x are not referenced.
               j

2: N -- INTEGER Input
On entry: the number of data values, n. Constraint: N >= 1.

3: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   N < 1.

7. Accuracy

Exact.

8. Further Comments

The time taken by the routine is negligible.

9. Example

This program reads in a sequence of real data values, calls
C06EAF followed by C06GBF to compute their inverse discrete
Fourier transform, and prints this after expanding it from
Hermitian form into a full complex sequence.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

C06GCF(3NAG) Foundation Library (12/10/92) C06GCF(3NAG)

C06 -- Summation of Series
C06GCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06GCF forms the complex conjugate of a sequence of n data values.

2. Specification

```
SUBROUTINE C06GCF (Y, N, IFAIL)
 INTEGER N, IFAIL
 DOUBLE PRECISION Y(N)
```

3. Description

This is a utility routine for use in conjunction with C06ECF or C06FCF(*) to calculate inverse discrete Fourier transforms (see the Chapter Introduction).

4. References

None.

5. Parameters

1: Y(N) -- DOUBLE PRECISION array Input/Output
   On entry: if Y is declared with bounds (0:N-1) in the (sub)program which C06GCF is called, then Y(j) must contain the imaginary part of the jth data value, for 0<=j<=n-1. On exit: these values are negated.

2: N -- INTEGER Input
   On entry: the number of data values, n. Constraint: N >= 1.

3: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
CHAPTER 15. CHAPTER N

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
N < 1.

7. Accuracy

Exact.

8. Further Comments

The time taken by the routine is negligible.

9. Example

This program reads in a sequence of complex data values and prints their inverse discrete Fourier transform as computed by calling C06GCF, followed by C06ECF and C06GCF again.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C06GQF(3NAG)  Foundation Library (12/10/92)  C06GQF(3NAG)

C06 -- Summation of Series  C06GQF
C06GQF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06GQF forms the complex conjugates of m Hermitian sequences, each containing n data values.

2. Specification

SUBROUTINE C06GQF (M, N, X, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N)

3. Description

This is a utility routine for use in conjunction with C06FPF and C06FQF to calculate inverse discrete Fourier transforms (see the Chapter Introduction).

4. References

None.

5. Parameters

1: M -- INTEGER Input
   On entry: the number of Hermitian sequences to be conjugated, m. Constraint: M >= 1.

2: N -- INTEGER Input
   On entry: the number of data values in each Hermitian sequence, n. Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input/Output
   On entry: the data must be stored in array X as if in a two-dimensional array of dimension (1:M,0:N-1); each of the m sequences is stored in a row of the array in Hermitian form. If the n data values z are written as x +iy, then for 0<=j<=n/2, x is contained in X(p,j), and for 1<=j<=(n-1)/2, y is contained in X(p,n-j). (See also Section 2.1.2 of the Chapter Introduction.) On exit: the imaginary parts y are negated. The real parts x are not referenced.

4: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry M < 1.

IFAIL = 2
On entry N < 1.

7. Accuracy
Exact.

8. Further Comments
None.

9. Example

This program reads in sequences of real data values which are assumed to be Hermitian sequences of complex data stored in Hermitian form. The sequences are expanded into full complex form using C06GSF and printed. The sequences are then conjugated (using C06GQF) and the conjugated sequences are expanded into complex form using C06GSF and printed out.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

C06GSF(3NAG)  Foundation Library (12/10/92)  C06GSF(3NAG)

C06 -- Summation of Series  C06GSF
C06GSF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

C06GSF takes m Hermitian sequences, each containing n data values, and forms the real and imaginary parts of the m corresponding complex sequences.
2. Specification

SUBROUTINE C06GSF (M, N, X, U, V, IFAIL)
INTEGER M, N, IFAIL
DOUBLE PRECISION X(M*N), U(M*N), V(M*N)

3. Description

This is a utility routine for use in conjunction with C06FPF and C06FQF (see the Chapter Introduction).

4. References

None.

5. Parameters

1: M -- INTEGER Input
On entry: the number of Hermitian sequences, m, to be
converted into complex form. Constraint: M >= 1.

2: N -- INTEGER Input
On entry: the number of data values, n, in each sequence.
Constraint: N >= 1.

3: X(M,N) -- DOUBLE PRECISION array Input
On entry: the data must be stored in X as if in a two-
dimensional array of dimension (1:M,0:N-1); each of the m
sequences is stored in a row of the array in Hermitian form.

If the n data values z are written as x +iy , then for
j j j
0<=j<=n/2, x is contained in X(p,j), and for 1<=j<=(n-1)/2,

p p p
j j j
y is contained in X(p,n-j). (See also Section 2.1.2 of the
Chapter Introduction.)

4: U(M,N) -- DOUBLE PRECISION array Output

5: V(M,N) -- DOUBLE PRECISION array Output
On exit: the real and imaginary parts of the m sequences of
length n, are stored in U and V respectively, as if in two-
dimensional arrays of dimension (1:M,0:N-1); each of the m
sequences is stored as if in a row of each array. In other
words, if the real parts of the pth sequence are denoted by
p
x , for j=0,1,...,n-1 then the mn elements of the array U
contain the values
\[
\begin{array}{cccccccc}
1 & 2 & m & 1 & 2 & m & 1 & 2 & m \\
x & x & \ldots & x & x & \ldots & x & x & \ldots & x \\
0 & 0 & 0 & 1 & 1 & 1 & n-1 & n-1 & n-1 \\
\end{array}
\]

6: IFAIL -- INTEGER
Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry \( M < 1 \).

IFAIL = 2
On entry \( N < 1 \).

7. Accuracy
Exact.

8. Further Comments
None.

9. Example
This program reads in sequences of real data values which are assumed to be Hermitian sequences of complex data stored in Hermitian form. The sequences are then expanded into full complex form using C06GSF and printed.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
NagSeriesSummationPackage (NAGC06)

Exports:
  c06eaf  c06ebf  c06ecf  c06ekf  c06fpf
  c06fqf  c06frf  c06fuf  c06gbf  c06gcf
  c06gsf

--- package NAGC06 NagSeriesSummationPackage ---

)abbrev package NAGC06 NagSeriesSummationPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:44:30 1994
++ Description:
++ This package uses the NAG Library to calculate the discrete Fourier
++ transform of a sequence of real or complex data values, and
++ applies it to calculate convolutions and correlations.

NagSeriesSummationPackage(): Exports == Implementation where
  S ==> Symbol
  FOP ==> FortranOutputStackPackage

Exports ==
  with
  c06eaf : (Integer,Matrix DoubleFloat,Integer) -> Result
    ++ c06eaf(n,x,ifail)
    ++ calculates the discrete Fourier transform of a sequence of
    ++ n real data values. (No extra workspace required.)
    ++ See \downlink{Manual Page}{manpageXXc06eaf}.
  c06ebf : (Integer,Matrix DoubleFloat,Integer) -> Result
    ++ c06ebf(n,x,ifail)
    ++ calculates the discrete Fourier transform of a Hermitian
    ++ sequence of n complex data values. (No extra workspace required.)
    ++ See \downlink{Manual Page}{manpageXXc06ebf}.
  c06ecf : (Integer,Matrix DoubleFloat,Matrix DoubleFloat,Integer) -> Result
    ++ c06ecf(n,x,y,ifail)
    ++ calculates the discrete Fourier transform of a sequence of
    ++ n complex data values. (No extra workspace required.)
++ See \downlink{Manual Page}{manpageXXc06ecf}.
c06ekf : (Integer,Integer,Matrix DoubleFloat,Matrix DoubleFloat, Integer) -> Result
c06ekf(job,n,x,y,ifail)
c06ekf calculates the circular convolution of two real vectors of period n. No extra workspace is required.
++ See \downlink{Manual Page}{manpageXXc06ekf}.
c06fpf : (Integer,Integer,String,Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
c06fpf(m,n,init,x,trig,ifail)
c06fpf computes the discrete Fourier transforms of m sequences, each containing n real data values. This routine is designed to be particularly efficient on vector processors.
++ See \downlink{Manual Page}{manpageXXc06fpf}.
c06fqf : (Integer,Integer,String,Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
c06fqf(m,n,init,x,trig,ifail)
c06fqf computes the discrete Fourier transforms of m Hermitian sequences, each containing n complex data values. This routine is designed to be particularly efficient on vector processors.
++ See \downlink{Manual Page}{manpageXXc06fqf}.
c06frf : (Integer,Integer,String,Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
c06frf(m,n,init,x,y,trig,ifail)
c06frf computes the discrete Fourier transforms of m sequences, each containing n complex data values. This routine is designed to be particularly efficient on vector processors.
++ See \downlink{Manual Page}{manpageXXc06frf}.
c06fuf : (Integer,Integer,String,Matrix DoubleFloat, Matrix DoubleFloat, Matrix DoubleFloat, Integer) -> Result
c06fuf(m,n,init,x,y,trigm,trign,ifail)
c06fuf computes the two-dimensional discrete Fourier transform of a bivariate sequence of complex data values. This routine is designed to be particularly efficient on vector processors.
++ See \downlink{Manual Page}{manpageXXc06fuf}.
c06gbf : (Integer,Matrix DoubleFloat,Integer) -> Result
c06gbf(n,x,ifail)
c06gbf forms the complex conjugate of n data values.
++ See \downlink{Manual Page}{manpageXXc06gbf}.
c06gcf : (Integer,Matrix DoubleFloat,Integer) -> Result
c06gcf(n,y,ifail)
c06gcf forms the complex conjugate of a sequence of n data values.
++ See \downlink{Manual Page}{manpageXXc06gcf}.
c06gqf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
c06gqf(m,n,x,ifail)
c06gqf forms the complex conjugates, each containing n data values.
++ See \downlink{Manual Page}{manpageXXc06gsf}.
c06gsf : (Integer,Integer,Matrix DoubleFloat,Integer) -> Result
  ++ c06gsf(m,n,x,ifail)
  ++ takes m Hermitian sequences, each containing n data
  ++ values, and forms the real and imaginary parts of the m
  ++ corresponding complex sequences.
  ++ See \downlink{Manual Page}{manpageXXc06gsf}.
Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Matrix Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Integer)
import AnyFunctions1(String)
import AnyFunctions1(Matrix DoubleFloat)

c06eaf(nArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "c06eaf",_
    ["n"::S,"ifail"::S,"x"::S]$Lisp,_
    []$Lisp,_
    [["double"::S,"x"::S,"n"::S]$Lisp]$Lisp_,
    ["integer"::S,"n"::S,"ifail"::S]$Lisp_]
    )$Lisp_,
  ["x"::S,"ifail"::S]$Lisp_,
  [(nArg::Any,ifailArg::Any,xArg::Any )]
  @List Any($Lisp)@$Lisp_)
pretend List (Record(key:Symbol,entry:Any))$Result

c06ebf(nArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp,_
    "c06ebf",_
    ["n"::S,"ifail"::S,"x"::S]$Lisp,_
    []$Lisp_,
    [["double"::S,"x"::S,"n"::S]$Lisp]$Lisp_,
    ["integer"::S,"n"::S,"ifail"::S]$Lisp_]
    )$Lisp_,
  ["x"::S,"ifail"::S]$Lisp_,
  [(nArg::Any,ifailArg::Any,xArg::Any )]
  @List Any($Lisp)@$Lisp_)
pretend List (Record(key:Symbol,entry:Any))$Result

c06ecf(nArg:Integer,xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
  ifailArg:Integer): Result ==
c06ekf(jobArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
yArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c06ekf",_ 
[])$Lisp,_ 
["double":S,["x":S,"n":S]$Lisp,["y":S,"n":S]$Lisp_ ]$Lisp_ ,
["integer":S,"n":S,"ifail":S]$Lisp_ ]$Lisp_ 
["x":S,"y":S,"ifail":S]$Lisp_,
[(nArg::Any,ifailArg::Any,xArg::Any,yArg::Any ])
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

c06fpf(mArg:Integer,nArg:Integer,initArg:String,_
xArg:Matrix DoubleFloat,trigArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"c06fpf",_ 
["work":S]$Lisp_ ,
["double":S,["x":S,"m":S,"n":S]$Lisp]$Lisp_ ,
["trig":S,["*":S,"m":S,"n":S]$Lisp]$Lisp_ ,
["work":S,["*":S,"m":S,"n":S]$Lisp]$Lisp_ ,
["integer":S,"m":S,"n":S,"ifail":S]$Lisp_ ]$Lisp_ ,
["character":S,"init":S]$Lisp_ ]$Lisp_ 
["x":S,"trig":S,"ifail":S]$Lisp_ ,
[(mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
xArg::Any,tryArg::Any ])
@List Any]$Lisp)$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result

c06fqf(mArg:Integer,nArg:Integer,initArg:String,_
xArg:Matrix DoubleFloat,trigArg:Matrix DoubleFloat,_
ifailArg:Integer): Result ==
c06fqf(mArg:Integer,nArg:Integer,initArg:String,_
xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
trigArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "c06fqf",_[
  ["work"::S]$Lisp,_
  ["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,
  ["trig"::S,["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,_
  ["work"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,$Lisp,
  ["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp,
  ["character"::S,"init"::S]$Lisp_]
)$Lisp,
["x"::S,"trig"::S,"ifail"::S]$Lisp,_
[(\[mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,_
  trigArg::Any \])_@List Any]$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result


c06frf(mArg:Integer,nArg:Integer,initArg:String,_
xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
trigArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
  "c06frf",_[
  ["work"::S]$Lisp,_
  ["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,
  ["y"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,["trig"::S,_
  ["*"::S,2$Lisp,"n"::S]$Lisp]$Lisp,$Lisp,["work"::S,_
  ["*"::S,["*"::S,2$Lisp,"m"::S]$Lisp,"n"::S]$Lisp]$Lisp,$Lisp,
  ["integer"::S,"m"::S,"n"::S,"ifail"::S]$Lisp,
  ["character"::S,"init"::S]$Lisp]_\[mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
xArg::Any,yArg::Any,trigArg::Any \])_@List Any]$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result


c06fuf(mArg:Integer,nArg:Integer,initArg:String,_
xArg:Matrix DoubleFloat,yArg:Matrix DoubleFloat,_
[(invokeNagman(NIL$Lisp,_
  "c06fuf",_[
  ["work"::S]$Lisp,_
  ["double"::S,["x"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,
  ["y"::S,["*"::S,"m"::S,"n"::S]$Lisp]$Lisp,["trig"::S,_
  ["*"::S,2$Lisp,"m"::S]$Lisp,"n"::S]$Lisp]$Lisp,$Lisp,
  ["*"::S,["*"::S,2$Lisp,"m"::S]$Lisp]$Lisp,$Lisp,["trig"::S,_
  ["*"::S,2$Lisp,"m"::S]$Lisp]$Lisp,$Lisp,$Lisp]_\[mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,_
xArg::Any,yArg::Any,trigArg::Any ]_@List Any]$Lisp)_
pretend List (Record(key:Symbol,entry:Any))$Result
c06gbf(nArg:Integer,xArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
  ((invokeNagman(NIL$Lisp_,
    "c06gbf",_)
    ["n":S,"ifail":S,"x":S]$Lisp_,
    [])$Lisp_,
    [["double":S,["x":S,"n":S]$Lisp]$Lisp_,
      ["integer":S,"n":S,"ifail":S]$Lisp_]
    $Lisp_,
    ["x":S,"ifail":S]$Lisp_,
    [([mArg::Any,nArg::Any,initArg::Any,ifailArg::Any,xArg::Any,_
        yArg::Any,trimArg::Any,trignArg::Any ])
     @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))$Result

c06gcf(nArg:Integer,yArg:Matrix DoubleFloat,ifailArg:Integer): Result ==
  ((invokeNagman(NIL$Lisp_,
    "c06gcf",_)
    ["n":S,"ifail":S,"y":S]$Lisp_,
    [])$Lisp_,
    [["double":S,["y":S,"n":S]$Lisp]$Lisp_,
      ["integer":S,"n":S,"ifail":S]$Lisp_]
    $Lisp_,
    ["y":S,"ifail":S]$Lisp_,
    [([nArg::Any,ifailArg::Any,yArg::Any ])
     @List Any]$Lisp)$Lisp)_
    pretend List (Record(key:Symbol,entry:Any))$Result

c06gqf(mArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_,
 ifailArg:Integer): Result ==
  ((invokeNagman(NIL$Lisp_,
    "c06gqf",_)
    ["m":S,"n":S,"ifail":S,"x":S]$Lisp_,
    [])$Lisp_,
    [["double":S,["x":S,["*":S,"m":S,"n":S]$Lisp]$Lisp_]
    $Lisp_,
    ["integer":S,"m":S,"n":S,"ifail":S]$Lisp_]
    $Lisp_,
    ["x":S,"ifail":S]$Lisp_,
    [([mArg::Any,nArg::Any,ifailArg::Any,xArg::Any ])
     @List Any]$Lisp)$Lisp)_
package NAGS NagSpecialFunctionsPackage

pretend List (Record(key:Symbol,entry:Any))$Result

c06gsf(mArg:Integer,nArg:Integer,xArg:Matrix DoubleFloat,_
   ifailArg:Integer): Result ==
   ([invokelnagman(NIL$Lisp,_
     "c06gsf",_
     ["u":S,"v":S]$Lisp,_
     [["double":S,"x":S,"*:S","m":S,"n":S]$Lisp]$Lisp$_
     ,["u":S,"*:S","m":S,"n":S]$Lisp]$Lisp$_
     ,["v":S,"*:S","m":S,"n":S]$Lisp]$Lisp$_
     ,["integer":S,"m":S,"n":S,"ifail":S]$Lisp$_
   ])$Lisp$_
   ,["u":S,"v":S,"ifail":S]$Lisp$_
   ,[(mArg::Any,nArg::Any,ifailArg::Any,xArg::Any )]_@
   pretend List (Record(key:Symbol,entry:Any))$Result

— NAGC06.dotabb —

"NAGC06" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGC06"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGC06" -> "ALIST"

— NagSpecialFunctionsPackage.input —

)set break resume
)sys rm -f NagSpecialFunctionsPackage.output
)spool NagSpecialFunctionsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 41
)show NagSpecialFunctionsPackage
--R
--R NagSpecialFunctionsPackage is a package constructor
--R Abbreviation for NagSpecialFunctionsPackage is NAGS
CHAPTER 15. CHAPTER N

--R This constructor is exposed in this frame.
--R Issue \texttt{)edit bookvol10.4.pamphlet} to see algebra source code for NAGS
--R
--R------------------------------- Operations -------------------------------
--R s01eaf : (Complex(DoubleFloat),Integer) -> Result
--R s13aaf : (DoubleFloat,Integer) -> Result
--R s13acf : (DoubleFloat,Integer) -> Result
--R s13adf : (DoubleFloat,Integer) -> Result
--R s14aaf : (DoubleFloat,Integer) -> Result
--R s14abf : (DoubleFloat,Integer) -> Result
--R s14abf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
--R s15adf : (DoubleFloat,Integer) -> Result
--R s15aef : (DoubleFloat,Integer) -> Result
--R s17acf : (DoubleFloat,Integer) -> Result
--R s17adf : (DoubleFloat,Integer) -> Result
--R s17aef : (DoubleFloat,Integer) -> Result
--R s17aff : (DoubleFloat,Integer) -> Result
--R s17agf : (DoubleFloat,Integer) -> Result
--R s17ahf : (DoubleFloat,Integer) -> Result
--R s17ajf : (DoubleFloat,Integer) -> Result
--R s17aef : (DoubleFloat,Integer) -> Result
--R s17acf : (DoubleFloat,Integer) -> Result
--E 1

--S 2 of 41
showArrayValues true
--R
--R
--R (1) true
--R Type: Boolean
showScalarValues true

--R

(2) true

Type: Boolean

result:=s01eaf(complex(-0.5,2.0),-1)

result:=s13aaf(2.0,-1)

result:=s13acf(0.2,-1)

result:=s13adf(0.2,-1)

result:=s14aaf(1.25,-1)

result:=s14abf(1.25,-1)

result:=s14baf(2.0,3.0,1.1102230246251600E-16,-1)

result:=s15adf(-10.0,-1)

result:=s15aef(-6.0,-1)

result:=s17acf(0.5,-1)
--S 14 of 41
-- result:=s17adf(0.5,-1)
--E 14

--S 15 of 41
-- result:=s17aef(0.5,-1)
--E 15

--S 16 of 41
-- result:=s17aff(0.5,-1)
--E 16

--S 17 of 41
-- result:=s17agf(-10.0,-1)
--E 17

--S 18 of 41
-- result:=s17ahf(-10.0,-1)
--E 18

--S 19 of 41
-- result:=s17ajf(-10.0,-1)
--E 19

--S 20 of 41
-- result:=s17akf(-10.0,-1)
--E 20

--S 21 of 41
-- result:=s17dcf(0.0,complex(0.3,0.4),2,"u", -1)
--E 21

--S 22 of 41
-- result:=s17def(0.0,complex(0.3,0.4),2,"u", -1)
--E 22

--S 23 of 41
-- result:=s17dgf("f",complex(0.3,0.4),"u", -1)
--E 23

--S 24 of 41
-- result:=s17dhf("f",complex(0.3,0.4),"u", -1)
--E 24

--S 25 of 41
-- result:=s17dlf(1, 0.0,complex(0.3,0.4),2,"u", -1)
--E 25

--S 26 of 41
-- result:=s18acf(0.4,-1)
--E 26

--S 27 of 41
-- result:=s18adf(0.4,-1)
--E 27

--S 28 of 41
-- result:=s18aef(0.5,-1)
--E 28

--S 29 of 41
-- result:=s18aff(0.5,-1)
--E 29

--S 30 of 41
-- result:=s18dcf(0.0,complex(0.3,0.4),2,"u", -1)
--E 30

--S 31 of 41
-- result:=s18def(0.0,complex(0.3,-0.4),2,"u", -1)
--E 31

--S 32 of 41
-- result:=s19aaf(1.0,-1)
--E 32

--S 33 of 41
-- result:=s19abf(0.1,-1)
--E 33

--S 34 of 41
-- result:=s19acf(0.1,-1)
--E 34

--S 35 of 41
-- result:=s19adf(0.0,-1)
--E 35

--S 36 of 41
-- result:=s20acf(0.5,-1)
--E 36

--S 37 of 41
-- result:=s20adf(0.5,-1)
--E 37

--S 38 of 41
-- result:=s21baf(0.5,1.0,-1)
--E 38
This package uses the NAG Library to compute some commonly occurring physical and mathematical functions.

1. Scope of the Chapter

This chapter is concerned with the provision of some commonly occurring physical and mathematical functions.

2. Background to the Problems

The majority of the routines in this chapter approximate real-valued functions of a single real argument, and the techniques involved are described in Section 2.1. In addition the chapter contains routines for elliptic integrals (see Section 2.2), Bessel and Airy functions of a complex argument (see Section 2.3), and exponential of a complex argument.

2.1. Functions of a Single Real Argument

Most of the routines for functions of a single real argument have been based on truncated Chebyshev expansions. This method of approximation was adopted as a compromise between the conflicting requirements of efficiency and ease of implementation on many different machine ranges. For details of the reasons behind this choice and the production and testing procedures followed in
constructing this chapter see Schonfelder [7].

Basically if the function to be approximated is \( f(x) \), then for \( x \) is in \([a,b]\) an approximation of the form

\[
f(x) = g(x) > \sum_{r=0}^{\infty} C_r T_r(t)
\]

is used, ( > denotes, according to the usual convention, a summation in which the first term is halved), where \( g(x) \) is some suitable auxiliary function which extracts any singularities, asymptotes and, if possible, zeros of the function in the range in question and \( t = t(x) \) is a mapping of the general range \([a,b]\) to the specific range \([-1,+1]\) required by the Chebyshev polynomials, \( T_r(t) \). For a detailed description of the properties of the Chebyshev polynomials see Clenshaw [5] and Fox and Parker [6].

The essential property of these polynomials for the purposes of function approximation is that \( T_r(t) \) oscillates between \( +1 \) and \( -1 \) it takes its extreme values \( n+1 \) times in the interval \([-1,+1]\). Therefore, provided the coefficients \( C_r \) decrease in magnitude sufficiently rapidly the error made by truncating the Chebyshev expansion after \( n \) terms is approximately given by

\[
E(t) \approx C_n T_n(t)
\]

That is the error oscillates between \( +C_n \) and takes its extreme value \( n+1 \) times in the interval in question. Now this is just the condition that the approximation be a mini-max representation, one which minimizes the maximum error. By suitable choice of the interval, \([a,b]\), the auxiliary function, \( g(x) \), and the mapping of the independent variable, \( t(x) \), it is almost always possible to obtain a Chebyshev expansion with rapid convergence and hence truncations that provide near mini-max polynomial approximations to the required function. The difference between the true mini-max polynomial and the truncated Chebyshev expansion is seldom sufficiently great to be of significance.

The evaluation of the Chebyshev expansions follows one of two
methods. The first and most efficient, and hence most commonly used, works with the equivalent simple polynomial. The second method, which is used on the few occasions when the first method proves to be unstable, is based directly on the truncated Chebyshev series and uses backward recursion to evaluate the sum. For the first method, a suitably truncated Chebyshev expansion (truncation is chosen so that the error is less than the machine precision) is converted to the equivalent simple polynomial. That is we evaluate the set of coefficients \( b \) such that

\[
y(t) = \sum_{r=0}^{n-1} b_r t^r = \sum_{r=0}^{n-1} c_r T_r(t).
\]

The polynomial can then be evaluated by the efficient Horner’s method of nested multiplications,

\[
y(t) = (b_0 + t(b_1 + t(b_2 + \ldots + t(b_{n-2} + b_{n-1})))).
\]

This method of evaluation results in efficient routines but for some expansions there is considerable loss of accuracy due to cancellation effects. In these cases the second method is used. It is well known that if

\[
b = \sum_{r=0}^{n-1} c_r T_r(t)
\]

then

\[
y(t) = (b_0 - b_1 + b_2 - b_3 + \ldots - b_{n-1} + b_{n-2})
\]

and this is always stable. This method is most efficiently implemented by using three variables cyclically and explicitly constructing the recursion.

That is,
(alpha) = C
n-1
(beta) = 2t(alpha)+C
n-2
(gamma) = 2t(beta)-(alpha)+C
n-3
(alpha) = 2t(gamma)-(beta)+C
n-4
(beta) = 2t(alpha)-(gamma)+C
n-5
...
...
(alpha) = 2t(gamma)-(beta)+C (say)
2
(beta) = 2t(alpha)-(gamma)+C
1
1
y(t) = t(beta)-(alpha)+ -C
2 0

The auxiliary functions used are normally functions compounded of simple polynomial (usually linear) factors extracting zeros, and
the primary compiler-provided functions, sin, cos, ln, exp, sqrt,
which extract singularities and/or asymptotes or in some cases
basic oscillatory behaviour, leaving a smooth well-behaved
function to be approximated by the Chebyshev expansion which can
therefore be rapidly convergent.

The mappings of \([a,b]\) to \([-1,+1]\) used, range from simple linear mappings to the case when \(b\) is infinite and considerable
improvement in convergence can be obtained by use of a bilinear
form of mapping. Another common form of mapping is used when the
function is even, that is it involves only even powers in its
expansion. In this case an approximation over the whole interval
\([-a,a]\) can be provided using a mapping \(t=2(-a) -1\). This embodies
the evenness property but the expansion in \(t\) involves all powers and hence removes the necessity of working with an expansion with
half its coefficients zero.

For many of the routines an analysis of the error in principle is
given, viz, if \(E\) and \((nabla)\) are the absolute errors in function
and argument and \((epsilon)\) and \((delta)\) are the corresponding
relative errors, then

\[ E^* = |f'(x)|(nabla) \]
\[ E^* = |xf'(x)|(delta) \]
If we ignore errors that arise in the argument of the function by propagation of data errors etc and consider only those errors that result from the fact that a real number is being represented in the computer in floating-point form with finite precision, then $\delta$ is bounded and this bound is independent of the magnitude of $x$; e.g. on an 11-digit machine

$$10^{-11}$$

$|\delta| \leq 10$ .

(This of course implies that the absolute error $\nabla = x\delta$ is also bounded but the bound is now dependent on $x$). However because of this the last two relations above are probably of more interest. If possible the relative error propagation is discussed; that is the behaviour of the error amplification factor $|xf'(x)/f(x)|$ is described, but in some cases, such as near zeros of the function which cannot be extracted explicitly, absolute error in the result is the quantity of significance and here the factor $|xf'(x)|$ is described. In general, testing of the functions has shown that their error behaviour follows fairly well these theoretical error behaviours. In regions, where the error amplification factors are less than or of the order of one, the errors are slightly larger than the above predictions. The errors are here limited largely by the finite precision of arithmetic in the machine but $\varepsilon$ is normally no more than a few times greater than the bound on $\delta$. In regions where the amplification factors are large, order of ten or greater, the theoretical analysis gives a good measure of the accuracy obtainable.

It should be noted that the definitions and notations used for the functions in this chapter are all taken from Abramowitz and Stegun [1]. Users are strongly recommended to consult this book for details before using the routines in this chapter.

2.2. Approximations to Elliptic Integrals

The functions provided here are symmetrised variants of the classic elliptic integrals. These alternative definitions have been suggested by Carlson (see [2], [3] and [4]) and he also developed the basic algorithms used in this chapter.

The standard integral of the first kind is represented by $\infty$
\[ \frac{1}{R(t)} = \int_{0}^{1} \frac{dt}{F(t+x)(t+y)(t+z)} \]

where \( x, y, z \geq 0 \) and at most one may be equal to zero.

The normalisation factor, \( \frac{1}{F} \), is chosen so as to make

\[ R(x,x,x) = \frac{1}{\sqrt{x}}. \]

If any two of the variables are equal, \( R \) degenerates into the second function

\[ \frac{1}{R(t)} = \int_{0}^{\infty} \frac{dt}{C(t+x)(t+y)} \]

where the argument restrictions are now \( x > 0 \) and \( y \neq 0 \).

This function is related to the logarithm or inverse hyperbolic functions if \( 0 < y < x \), and to the inverse circular functions if \( 0 \leq x < y \).

The integrals of the second kind are defined by

\[ \frac{1}{R(t)} = \int_{0}^{3} \frac{dt}{D(t+x)(t+y)(t+z)} \]

with \( z > 0 \), \( x \geq 0 \) and \( y \geq 0 \) but only one of \( x \) or \( y \) may be zero.

The function is a degenerate special case of the integral of the third kind
with \( \rho \neq 0 \), \( x, y, z \geq 0 \) with at most one equality holding. Thus \( R_{J}(x,y,z) = R_{J}(x,y,z,z) \). The normalisation of both these functions is chosen so that

\[
R_{D}(x,x,x) = R_{J}(x,x,x) = \frac{1}{\sqrt{x}}
\]

The algorithms used for all these functions are based on duplication theorems. These allow a recursion system to be established which constructs a new set of arguments from the old using a combination of arithmetic and geometric means. The value of the function at the original arguments can then be simply related to the value at the new arguments. These recursive reductions are used until the arguments differ from the mean by an amount small enough for a Taylor series about the mean to give sufficient accuracy when retaining terms of order less than six. Each step of the recurrences reduces the difference from the mean by a factor of four, and as the truncation error is of order six, the truncation error goes like \( 4^{n} \), where \( n \) is the number of iterations.

The above forms can be related to the more traditional canonical forms (see Abramowitz and Stegun [1], 17.2).

If we write \( q = \cos(\phi), r = 1 - m \sin(\phi), s = 1 + n \sin(\phi) \), where

\[
0 < \phi \leq -\pi,
\]

we have: the elliptic integral of the first kind:

\[
\begin{align*}
\frac{\sin(\phi)}{2} & - \frac{1}{2} \quad 1 - \frac{1}{2} \\
F((\phi)|m) &= \int_{0}^{1} \left(1 - t^{2}ight) \left(1 - mt^{2}\right) \frac{dt}{\sin(\phi) R(q,r,1)} \quad F
\end{align*}
\]

the elliptic integral of the second kind:

\[
\begin{align*}
\frac{\sin(\phi)}{2} & - \frac{1}{2} \quad - \frac{1}{2} \\
E((\phi)|m) &= \int_{0}^{1} \left(1 - t^{2}ight) \left(1 - mt^{2}\right) dt
\end{align*}
\]
the elliptic integral of the third kind:

\[\frac{1}{2} -\frac{1}{2}
\]

\[\frac{2}{2} -\frac{2}{2} -\frac{1}{2} -\frac{1}{2}\]

\[\int_{(\pi)(n;(\phi)|m)}^{(\pi)(n)} \frac{(1-t) (1-mt) (1+nt)}{dt} \]

\[\frac{1}{3} -\frac{3}{3} -\frac{3}{3} -\frac{3}{3}\]

Also the complete elliptic integral of the first kind:

\[\frac{(\pi)/2}{2} -\frac{1}{2}\]

\[\int_{K(m)}^{(1-m.\sin(\theta))} d(\theta) = R(0,1-m,1)\]

Also the complete elliptic integral of the second kind:

\[\frac{(\pi)/2}{2} -\frac{1}{2}\]

\[\int_{E(m)}^{(1-m.\sin(\theta))} d(\theta) = R(0,1-m,1) - mR(0,1-m,1)\]

2.3. Bessel and Airy Functions of a Complex Argument

The routines for Bessel and Airy functions of a real argument are based on Chebyshev expansions, as described in Section 2.1. The routines for functions of a complex argument, however, use different methods. These routines relate all functions to the modified Bessel functions \(I_{(\nu)}(z)\) and \(K_{(\nu)}(z)\) computed in the right-half complex plane, including their analytic continuations. \(I_{(\nu)}(z)\) and \(K_{(\nu)}(z)\) are computed by different methods according to the values of \(z\) and \(\nu\). The methods include power series, asymptotic expansions and Wronskian evaluations. The relations between functions are based on well known formulae (see Abramowitz and Stegun [1]).
2.4. References


3. Recommendations on Choice and Use of Routines

3.1. Elliptic Integrals

IMPORTANT ADVICE: users who encounter elliptic integrals in the course of their work are strongly recommended to look at transforming their analysis directly to one of the Carlson forms, rather than the traditional canonical Legendre forms. In general, the extra symmetry of the Carlson forms is likely to simplify the analysis, and these symmetric forms are much more stable to calculate.

The routine S21BAF for R is largely included as an auxiliary to the other routines for elliptic integrals. This integral essentially calculates elementary functions, e.g.

\[
\begin{align*}
\ln(x) &= (x-1)R((\ldots),x), x > 0; \\
C((2)) &= \arcsin x = xR(1-x,1), |x| \leq 1;
\end{align*}
\]
2
arcsinhx=x.R (1+x ,1), etc
C

In general this method of calculating these elementary functions
is not recommended as there are usually much more efficient
specific routines available in the Library. However, S21BAF may
be used, for example, to compute lnx/(x-1) when x is close to 1,
without the loss of significant figures that occurs when lnx and
x-1 are computed separately.

3.2. Bessel and Airy Functions

For computing the Bessel functions \( J_\nu(x) \), \( Y_\nu(x) \), \( I_\nu(x) \)
and \( K_\nu(x) \) where \( x \) is real and \( \nu=0 \) or 1, special routines
are provided, which are much faster than the more general
routines that allow a complex argument and arbitrary real \( \nu\geq0 \)
functions and their derivatives \( Ai(x), Bi(x), Ai'(x), Bi'(x) \) for
a real argument which are much faster than the routines for
complex arguments.

3.3. Index

Airy function, \( Ai \), real argument S17AGF
Airy function, \( Ai' \), real argument S17AJF
Airy function, \( Ai \) or \( Ai' \), complex argument, optionally
scaled S17DGF
Airy function, \( Bi \), real argument S17AHF
Airy function, \( Bi' \), real argument S17AKF
Airy function, \( Bi \) or \( Bi' \), complex argument, optionally
scaled S17DHF
Bessel function, \( J_0(x) \), real argument S17AEF
Bessel function, \( J_1(x) \), real argument S17AFF
Bessel function, \( J_{\nu}(x) \), complex argument, optionally
scale d S17DEF
Bessel function, \( Y_0(x) \), real argument S17ACF
Bessel function, \( Y_1(x) \), real argument S17ADF
Bessel function, \( Y_{\nu}(x) \), complex argument, optionally
scale d S17DCF
Complement of the Error function S15ADF
Cosine Integral S13ACF
Elliptic integral, symmetrised, degenerate of 1st kind, S21BAF
<table>
<thead>
<tr>
<th>Function/Incorporated</th>
<th>Description</th>
<th>Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>S21BBF</td>
<td>Elliptic integral, symmetrised, of 1st kind, R</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S21BCF</td>
<td>Elliptic integral, symmetrised, of 2nd kind, R</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S21BDF</td>
<td>Elliptic integral, symmetrised, of 3rd kind, R</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S15AEF</td>
<td>Erf, real argument</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S15ADF</td>
<td>Erfc, real argument</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S15AEF</td>
<td>Error function</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S01EAF</td>
<td>Exponential, complex</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S13AAF</td>
<td>Exponential Integral</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S20ADF</td>
<td>Fresnel Integral, C</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S20ACF</td>
<td>Fresnel Integral, S</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S14AAAF</td>
<td>Gamma function</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S14BAF</td>
<td>Generalized Factorial function</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S17DLF</td>
<td>Hankel function H or H, complex argument, optionally scaled</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S14BAF</td>
<td>Incomplete Gamma function</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S21CAF</td>
<td>Jacobian elliptic functions, sn, cn, dn</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S19ABF</td>
<td>Kelvin function, bei x</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S19AAF</td>
<td>Kelvin function, ber x</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S19ADF</td>
<td>Kelvin function, kei x</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S19ACF</td>
<td>Kelvin function, ker x</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S14ABF</td>
<td>Logarithm of Gamma function</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S18AEF</td>
<td>Modified Bessel function, I, real argument 0</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S18AFF</td>
<td>Modified Bessel function, I, real argument 1</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S18SDEF</td>
<td>Modified Bessel function, I, complex argument, optionally scaled</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S18ACF</td>
<td>Modified Bessel function, K, real argument 0</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S18ADF</td>
<td>Modified Bessel function, K, real argument 1</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S18DCF</td>
<td>Modified Bessel function, K, complex argument, optionally scaled</td>
<td>Chapter 15.</td>
</tr>
<tr>
<td>S13ADF</td>
<td>Sine integral</td>
<td>Chapter 15.</td>
</tr>
</tbody>
</table>
S01EAF Complex exponential, $e^z$
S13AAF Exponential integral $E_1(x)$
S13ACF Cosine integral $\text{Ci}(x)$
S13ADF Sine integral $\text{Si}(x)$
S14AAF Gamma function
S14ABF Log Gamma function
S14BAF Incomplete gamma functions $P(a,x)$ and $Q(a,x)$
S15ADF Complement of error function $\text{erfc}(x)$
S15AEF Error function $\text{erf}(x)$
S17ACF Bessel function $Y_0(x)$
S17ADF Bessel function $Y_1(x)$
S17AEF Bessel function $J_0(x)$
S17AFF Bessel function $J_1(x)$
S17AGF Airy function $\text{Ai}(x)$
S17AHF Airy function $\text{Bi}(x)$
S17AJF Airy function $\text{Ai}'(x)$
S17AKF Airy function $\text{Bi}'(x)$
S17DCF Bessel functions $Y_{\nu+a}(z)$, real $a \geq 0$, complex $z$, $\nu=0,1,2,...$
S17DEF Bessel functions $J_{\nu+a}(z)$, real $a \geq 0$, complex $z$, $\nu=0,1,2,...$
S17DGF Airy functions $\text{Ai}(z)$ and $\text{Ai}'(z)$, complex $z$
S17DHF Airy functions $Bi(z)$ and $Bi'(z)$, complex $z$

$(j)$
S17DLF Hankel functions $H^{(j)}(z)$, $j=1,2$, real $a>=0$, complex $z$, $(\nu)+a$
$(\nu)=0,1,2,...$

S18ACF Modified Bessel function $K_0(x)$

S18ADF Modified Bessel function $K_1(x)$

S18AEF Modified Bessel function $I_0(x)$

S18AFF Modified Bessel function $I_1(x)$

S18DCF Modified Bessel functions $K_\nu(z)$, real $a>=0$, complex $z$, $(\nu)+a$
$(\nu)=0,1,2,...$

S18DEF Modified Bessel functions $I_\nu(z)$, real $a>=0$, complex $z$, $(\nu)+a$
$(\nu)=0,1,2,...$

S19AAF Kelvin function $ber x$

S19ABF Kelvin function $bei x$

S19ACF Kelvin function $ker x$

S19ADF Kelvin function $kei x$

S20ACF Fresnel integral $S(x)$

S20ADF Fresnel integral $C(x)$

S21BAF Degenerate symmetrised elliptic integral of 1st kind $R(x,y)$

S21BBF Symmetrised elliptic integral of 1st kind $R(x,y,z)$

S21BCF Symmetrised elliptic integral of 2nd kind $R(x,y,z)$
S01EAF evaluates the exponential function $e^z$, for complex $z$.

2. Specification

```fortran
COMPLEX(KIND(1.0D0)) FUNCTION S01EAF (Z, IFAIL)
    INTEGER IFAIL
    COMPLEX(KIND(1.0D0)) Z
```

3. Description

This routine evaluates the exponential function $e^z$, taking care to avoid machine overflow, and giving a warning if the result cannot be computed to more than half precision. The function is $z^x$ evaluated as $e^z = e^{(\cos y + i \sin y)}$, where $x$ and $y$ are the real and imaginary parts respectively of $z$.

Since $\cos y$ and $\sin y$ are less than or equal to 1 in magnitude, it is possible that $e^z$ may overflow although $e^{\cos y}$ or $e^{\sin y}$ does not. In this case the alternative formula $\text{sign}(\cos y)e^x\ln|\cos y|$ is used for the real part of the result, and $\text{sign}(\sin y)e^x\ln|\sin y|$ for the imaginary part. If either part of the result still overflows, a warning is returned through parameter IFAIL.

If $\text{Im } z$ is too large, precision may be lost in the evaluation of $\sin y$ and $\cos y$. Again, a warning is returned through IFAIL.

4. References

None.
5. Parameters

1: Z -- COMPLEX(KIND(1.0D0))  Input
   On entry: the argument z of the function.

2: IFAIL -- INTEGER  Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.

   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   The real part of the result overflows, and is set to the
   largest safe number with the correct sign. The imaginary
   part of the result is meaningful.

IFAIL= 2
   The imaginary part of the result overflows, and is set to
   the largest safe number with the correct sign. The real part
   of the result is meaningful.

IFAIL= 3
   Both real and imaginary parts of the result overflow, and
   are set to the largest safe number with the correct signs.

IFAIL= 4
   The computed result is accurate to less than half precision,
   due to the size of Im z.

IFAIL= 5
   The computed result has no precision, due to the size of Im
   z, and is set to zero.

7. Accuracy

Accuracy is limited in general only by the accuracy of the
Fortran intrinsic functions in the computation of siny, cosy and
x e , where x=Re z, y=Im z. As y gets larger, precision will
probably be lost due to argument reduction in the evaluation of
the sine and cosine functions, until the warning error IFAIL = 4
occurs when \( y \) gets larger than \( \sqrt{1/(\text{epsilon})} \), where (epsilon) is the machine precision. Note that on some machines, the intrinsic functions \( \text{SIN} \) and \( \text{COS} \) will not operate on arguments larger than about \( \sqrt{1/(\text{epsilon})} \), and so IFAIL can never return as 4.

In the comparatively rare event that the result is computed by
\[
\frac{x + \ln|\cos y|}{x + \ln|\sin y|}
\]
the formulae \( \text{sign}(\cos y)e \) and \( \text{sign}(\sin y)e \), a further small loss of accuracy may be expected due to rounding errors in the logarithmic function.

8. Further Comments
None.

9. Example
The example program reads values of the argument \( z \) from a file, evaluates the function at each value of \( z \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
The routine calculates an approximate value for

$$E(x) = |\frac{\infty}{e^{-u}} \int_{x} du, \quad x > 0.$$ 

For $0 < x \leq 4$, the approximation is based on the Chebyshev expansion

$$E(x) = y(t) - \ln x = \sum_{r=1}^{\infty} a_T(t) - \ln x, \quad t = -x - 1.$$ 

For $x > 4$,

$$E(x) = \frac{e^{-x}}{x} \sum_{r=1}^{\infty} a_T(t), \quad t = -1.0 + 14.5/(x+3.25) = \frac{11.25-x}{3.25+x}.$$ 

In both cases, $-1 \leq t \leq +1$.

To guard against producing underflows, if $x > x_{\text{hi}}$ the result is set directly to zero. For the value $x_{\text{hi}}$ see the Users’ Note for your implementation.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION

   Input:
   On entry: the argument $x$ of the function. Constraint: $X > 0$.

2: $\text{IFAIL}$ -- INTEGER

   Input/Output:
   Before entry, $\text{IFAIL}$ must be assigned a value. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
Unless the routine detects an error (see Section 6), IFAIL contains 0 on exit.

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
The routine has been called with an argument less than or equal to zero for which the function is not defined. The result returned is zero.

7. Accuracy

If (delta) and (epsilon) are the relative errors in argument and result respectively, then in principle,

<table>
<thead>
<tr>
<th>-x</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>(epsilon)</td>
<td>----*(delta)</td>
</tr>
<tr>
<td>E (x)</td>
<td>1</td>
</tr>
</tbody>
</table>

so the relative error in the argument is amplified in the result by at least a factor e^-x /E (x). The equality should hold if (delta) is greater than the machine precision ((delta) due to data errors etc) but if (delta) is simply a result of round-off in the machine representation, it is possible that an extra figure may be lost in internal calculation and round-off.

The behaviour of this amplification factor is shown in Figure 1.

Figure 1
Please see figure in printed Reference Manual

It should be noted that, for small x, the amplification factor tends to zero and eventually the error in the result will be limited by machine precision.

For large x,

(epsilon)^-x(delta)=Delta,

the absolute error in the argument.

8. Further Comments
9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S13 -- Approximations of Special Functions

S13ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S13ACF returns the value of the cosine integral

$$
Ci(x) = (\text{gamma}) + \ln x + \int_0^x \frac{1}{u} \, du, \quad x > 0
$$

via the routine name, where $(\text{gamma})$ denotes Euler's constant.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S13ACF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for $Ci(x)$.

For $0 < x \leq 16$ it is based on the Chebyshev expansion

$$
Ci(x) = \ln x + \sum_{r=0}^{\infty} a T_r(t), \quad t = 2(\frac{x}{16}) - 1
$$
For $16 < x < x_{\text{hi}}$ where the value of $x_{\text{hi}}$ is given in the Users’ Note for your implementation,

$$\frac{f(x) \sin x}{x^2} = \frac{g(x) \cos x}{x}$$

where $f(x) = \sum f(T(t))$ and $g(x) = \sum g(T(t))$, $t = 2(-1)^{r} - 1$.

$r = 0$  \hspace{1cm} r = 0$

For $x \geq x_{\text{hi}}$, $Ci(x) = 0$ to within the accuracy possible (see Section 7).

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument x of the function. Constraint: $X > 0$.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   The routine has been called with an argument less than or equal to zero for which the function is not defined. The result returned is zero.

7. Accuracy

If E and (epsilon) are the absolute and relative errors in the
result and (delta) is the relative error in the argument then in principle these are related by

\[ |E| = |(\delta)\cos x| \text{ and } |(\epsilon)| = | \frac{1}{C_\iota(x)} |. \]

That is accuracy will be limited by machine precision near the origin and near the zeros of \( \cos x \), but near the zeros of \( C_\iota(x) \) only absolute accuracy can be maintained.

The behaviour of this amplification is shown in Figure 1.

Figure 1
Please see figure in printed Reference Manual

For large values of \( x \), \( \sin x \approx \frac{1}{x} \) therefore

\[ (\epsilon) \approx (\delta) \cot x \text{ and since } (\delta) \text{ is limited by the finite precision of the machine it becomes impossible to return results which have any relative accuracy. That is, when } x = \frac{1}{(\delta)} \text{ we have that } |C_\iota(x)| < 1/x^E \text{ and hence is not significantly different from zero.} \]

Hence \( x \) is chosen such that for values of \( x \geq x \), \( C_\iota(x) \) in principle would have values less than the machine precision and so is essentially zero.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S13ADF returns the value of the sine integral

\[
\frac{x}{\sin u} \quad \text{Si}(x) = \int_{0}^{x} \frac{\sin u}{u} \, du,
\]

via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S13ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

The routine calculates an approximate value for \( \text{Si}(x) \).

For \(|x|\leq 16.0\) it is based on the Chebyshev expansion

\[
\text{Si}(x) = x + \sum_{r=0}^{\infty} a_r T_r(x), \quad r = 2(\frac{x}{16}) - 1.
\]

For \(16<|x|<x\), where \(x\) is an implementation dependent number,

\[
\text{Si}(x) = \text{sign}(x) \left\{ \frac{\pi f(x) \cos x}{2} + \frac{g(x) \sin x}{x} \right\},
\]

where \(f(x) = \sum_{r=0}^{\infty} f_r T_r(x)\) and \(g(x) = \sum_{r=0}^{\infty} g_r T_r(x)\), \(r = 2(\frac{x}{16}) - 1\).

For \(|x|\geq x\), \(\text{Si}(x) = -\frac{\pi}{2} \text{sign} x\) to within machine precision.
4. References


5. Parameters

1: X -- DOUBLE PRECISION  
   Input  
   On entry: the argument x of the function.

2: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy

If (delta) and (epsilon) are the relative errors in the argument and result, respectively, then in principle

\[
\frac{|(\delta)\sin x|}{|S(x)|} \leq \epsilon
\]

The equality may hold if (delta) is greater than the machine precision ((delta) due to data errors etc) but if (delta) is simply due to round-off in the machine representation, then since the factor relating (delta) to (epsilon) is always less than one, the accuracy will be limited by machine precision.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.
The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S14 -- Approximations of Special Functions

S14AAF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S14AAF returns the value of the Gamma function \( \Gamma(x) \), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S14AAF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Gamma function \( \Gamma(x) \). The routine is based on the Chebyshev expansion:

\[
\Gamma(1+u) = \sum_{r=0}^{\infty} a_r T_r(t), \quad 0 \leq u < 1, \quad t = 2u - 1,
\]

and uses the property \( \Gamma(1+x) = x\Gamma(x) \). If \( x = N+1+u \) where \( N \) is integral and \( 0 \leq u < 1 \) then it follows that:

- For \( N > 0 \):
  \[ \Gamma(x) = (x-1)(x-2)\ldots(x-N)\Gamma(1+u), \]

- For \( N = 0 \):
  \[ \Gamma(x) = \Gamma(1+u), \]

- For \( N < 0 \):
  \[ \Gamma(x) = \frac{\Gamma(1+u)}{x(x+1)(x+2)\ldots(x-N-1)} \]

There are four possible failures for this routine:

1. if \( x \) is too large, there is a danger of overflow since \( \Gamma(x) \) could become too large to be represented in the
(i) if $x$ is too large and negative, there is a danger of underflow;

(iii) if $x$ is equal to a negative integer, $\Gamma(x)$ would overflow since it has poles at such points;

(iv) if $x$ is too near zero, there is again the danger of overflow on some machines. For small $x$, $\Gamma(x) \approx -\frac{1}{x}$, and on some machines there exists a range of non-zero but small values of $x$ for which $1/x$ is larger than the greatest representable value.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument $x$ of the function. Constraint: X must not be a negative integer.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   The argument is too large. On soft failure the routine returns the approximate value of $\Gamma(x)$ at the nearest valid argument.

IFAIL= 2
   The argument is too large and negative. On soft failure the routine returns zero.

IFAIL= 3
   The argument is too close to zero. On soft failure the
routine returns the approximate value of $(\Gamma)(x)$ at the nearest valid argument.

IFAIL = 4
The argument is a negative integer, at which value $(\Gamma)(x)$ is infinite. On soft failure the routine returns a large positive value.

7. Accuracy

Let $(\delta)$ and $(\epsilon)$ be the relative errors in the argument and the result respectively. If $(\delta)$ is somewhat larger than the machine precision (i.e., is due to data errors etc), then $(\epsilon)$ and $(\delta)$ are approximately related by:

$$(\epsilon) = |x(\psi)(x)| \cdot (\delta)$$

(provided $(\epsilon)$ is also greater than the representation $(\Gamma)'(x)$ error). Here $(\psi)(x)$ is the digamma function.

Figure 1 shows the behaviour of the error amplification factor $|x(\psi)(x)|$.

Figure 1
Please see figure in printed Reference Manual

If $(\delta)$ is of the same order as machine precision, then rounding errors could make $(\epsilon)$ slightly larger than the above relation predicts.

There is clearly a severe, but unavoidable, loss of accuracy for arguments close to the poles of $(\Gamma)(x)$ at negative integers. However relative accuracy is preserved near the pole at $x=0$ right up to the point of failure arising from the danger of overflow.

Also accuracy will necessarily be lost as $x$ becomes large since in this region

$$(\epsilon) = (\delta) \cdot x \cdot \ln x.$$ 

However since $(\Gamma)(x)$ increases rapidly with $x$, the routine must fail due to the danger of overflow before this loss of accuracy is too great. (e.g. for $x=20$, the amplification factor $\approx 60$.)

8. Further Comments

For details of the time taken by the routine see the Users’ Note.
for your implementation.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S14 -- Approximations of Special Functions
S14ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S14ABF returns a value for the logarithm of the Gamma function, $\ln(\Gamma(x))$, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S14ABF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to $\ln(\Gamma(x))$. It is based on two Chebyshev expansions.

For $0 < x \leq x_{\text{small}}$, $\ln(\Gamma(x)) = -\ln(x)$ to within machine accuracy.

For $x < x_{\text{small}} \leq 15.0$, the recursive relation

\[
\Gamma(1+u) = x \Gamma(u) \quad \text{for} \quad 0 \leq u < 1
\]

is used to reduce the calculation to one involving $\Gamma(1+u)$, $0 \leq u < 1$ which is evaluated as:

\[
\Gamma(1+u) = \sum_{r=0}^{\infty} a_r T_r(t), \quad t = 2u - 1.
\]
Once \((\Gamma)(x)\) has been calculated, the required result is produced by taking the logarithm.

For \(15.0 \leq x \leq x\),

\[
\ln(\Gamma)(x) = (x - -)\ln x + -\ln 2(\pi) + y(x)/x
\]

\[
\begin{align*}
&\frac{1}{2} \quad\frac{1}{2} \\
\end{align*}
\]

where \(y(x) = > b T (t), t = 2 -> -1.
\]

\[
\begin{align*}
r &= 0
\end{align*}
\]

For \(x < x \leq x\) the term \(y(x)/x\) is negligible and so its calculation is omitted.

For \(x > x\) there is a danger of setting overflow so the routine must fail.

For \(x = 0.0\) the function is not defined and the routine fails.

Note: \(x\) is calculated so that if \(x < x\), \((\Gamma)(x) = 1/x\) to within machine accuracy. \(x\) is calculated so that if \(x > x\),

\[
\ln(\Gamma)(x) = (x - -)\ln x + -\ln 2(\pi)
\]

\[
\begin{align*}
&\frac{1}{2} \quad\frac{1}{2} \\
\end{align*}
\]

to within machine accuracy. \(x\) is calculated so that \(\ln(\Gamma)(x)\) is close to the value returned by X02ALF(*).

4. References


5. Parameters

1: \(X\) -- DOUBLE PRECISION
   Input
   On entry: the argument \(x\) of the function. Constraint: \(X > 0.0\).
2: IFAIL -- INTEGER  
   Input/Output  
On entry: IFAIL must be set to 0, -1 or 1. For users not  
familiar with this parameter (described in the Essential  
Introduction) the recommended value is 0.  
   On exit: IFAIL = 0 unless the routine detects an error (see  
Section 6).  

6. Error Indicators and Warnings  

Errors detected by the routine:  

IFAIL= 1  
X <= 0.0, the function is undefined. On soft failure, the  
routine returns zero.  

IFAIL= 2  
X is too large, the function would overflow. On soft  
failure, the routine returns the value of the function at  
the largest permissible argument.  

7. Accuracy  

Let (delta) and (epsilon) be the relative errors in the argument  
and result respectively, and E be the absolute error in the  
result.  

If (delta) is somewhat larger than the relative machine  
precision, then  

\[ | x*(\Psi)(x) | \]  
\[ E = | x*(\Psi)(x) | (\delta) \]  
\[ | ln(\Gamma)(x) | \]  
\[ (\Gamma)'(x) \]  

where (\Psi)(x) is the digamma function. Figure 1 and  
(\Gamma)(x)  
Figure 2 show the behaviour of these error amplification factors.  

Figure 1  
Please see figure in printed Reference Manual  

Figure 2  
Please see figure in printed Reference Manual  

These show that relative error can be controlled, since except  
near x=1 or 2 relative error is attenuated by the function or at  
least is not greatly amplified.
For large $x$, $(\epsilon) = (1 + \frac{1}{\ln x})(\delta)$ and for small $x$,
\[
\frac{1}{\ln x}
\]
$(\epsilon) = \frac{1}{\ln x}(\delta)$.

The function $\ln(\Gamma(x))$ has zeros at $x=1$ and 2 and hence relative accuracy is not maintainable near those points. However, absolute accuracy can still be provided near those zeros as is shown above.

If however, $(\delta)$ is of the order of the machine precision, then rounding errors in the routine’s internal arithmetic may result in errors which are slightly larger than those predicted by the equalities. It should be noted that even in areas where strong attenuation of errors is predicted the relative precision is bounded by the effective machine precision.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
SUBROUTINE S14BAF (A, X, TOL, P, Q, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION A, X, TOL, P, Q

3. Description

This subroutine evaluates the incomplete gamma functions in the
normalised form

\[ P(a,x) = \frac{1}{(\Gamma(a))} \int_0^1 t^{-a+1} e^{-t} dt, \]
\[ Q(a,x) = \frac{1}{(\Gamma(a))} \int_0^\infty t^{-a+1} e^{-t} dt, \]

with x \geq 0 and a > 0, to a user-specified accuracy. With this
normalisation, P(a,x) + Q(a,x) = 1.

Several methods are used to evaluate the functions depending on
the arguments a and x, the methods including Taylor expansion for
P(a,x), Legendre's continued fraction for Q(a,x), and power
series for Q(a,x). When both a and x are large, and a \approx x, the
uniform asymptotic expansion of Temme [3] is employed for greater
efficiency - specifically, this expansion is used when a \geq 20 and
0.7a \leq x \leq 1.4a.

Once either of P or Q is computed, the other is obtained by
subtraction from 1. In order to avoid loss of relative precision
in this subtraction, the smaller of P and Q is computed first.

This routine is derived from subroutine GAM in Gautschi [2].

4. References

Gamma Functions. ACM Trans. Math. Softw. 5 466--481.

ACM Trans. Math. Softw. 5 482--489.

Functions for Large Values of the Parameters. Algorithms for
Approximation. (ed J C Mason and M G Cox) Oxford University
Press.
5. Parameters

1: A -- DOUBLE PRECISION Input
   On entry: the argument a of the functions. Constraint: A > 0.0.

2: X -- DOUBLE PRECISION Input
   On entry: the argument x of the functions. Constraint: X >= 0.0.

3: TOL -- DOUBLE PRECISION Input
   On entry: the relative accuracy required by the user in the results. If S14BAF is entered with TOL greater than 1.0 or less than machine precision, then the value of machine precision is used instead.

4: P -- DOUBLE PRECISION Output

5: Q -- DOUBLE PRECISION Output
   On exit: the values of the functions P(a,x) and Q(a,x) respectively.

6: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry A <= 0.0.

IFAIL= 2
   On entry X < 0.0.

IFAIL= 3
   Convergence of the Taylor series or Legendre continued fraction fails within 600 iterations. This error is extremely unlikely to occur; if it does, contact NAG.

7. Accuracy

There are rare occasions when the relative accuracy attained is somewhat less than that specified by parameter TOL. However, the
error should never exceed more than one or two decimal places.
Note also that there is a limit of 18 decimal places on the
achievable accuracy, because constants in the routine are given
to this precision.

8. Further Comments

The time taken for a call of S14BAF depends on the precision
requested through TOL, and also varies slightly with the input
arguments a and x.

9. Example

The example program reads values of the argument a and x from a
file, evaluates the function and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

S15 -- Approximations of Special Functions
S15ADF
S15ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for
your implementation to check implementation-dependent details.
The symbol (*) after a NAG routine name denotes a routine that is
not included in the Foundation Library.

1. Purpose

S15ADF returns the value of the complementary error function,
erfcx, via the routine name.

2. Specification

DOUBLE PRECISION FUNCTION S15ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

The routine calculates an approximate value for the complement of
the error function

\[
2 \int_{-u}^{\infty} e^{-\frac{u^2}{2}} du = 1 - \text{erf} x.
\]
For \( x \geq 0 \), it is based on the Chebyshev expansion

\[
\text{erfc } x = e^{-x} y(x),
\]

where \( y(x) = \sum_{r=0}^{\infty} a_r T_r(t) \) and \( t = \frac{x-3.75}{x+3.75}, \, -1 \leq t \leq +1 \).

For \( x \geq x_h \), where there is a danger of setting underflow, the hi
result is returned as zero.

\[
\text{erfc } x = 2 - e^{-x} y(|x|).
\]

For \( x < 0 \), the result is returned as 2.0 which is correct to low
within machine precision. The values of \( x \) and \( x_l \) are given in the Users' Note for your implementation.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument \( x \) of the function.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this
chapter.

7. Accuracy

If \((\delta)\) and \((\epsilon)\) are relative errors in the argument and result, respectively, then in principle

\[
\frac{| \epsilon |}{\sqrt{\pi} \text{erfc} x} \approx \frac{| \epsilon |}{\delta}
\]

That is, the relative error in the argument, \(x\), is amplified by a factor \(\frac{2x}{\sqrt{\pi}}\) in the result.

\(\sqrt{\pi} \text{erfc} x\)

The behaviour of this factor is shown in Figure 1.

Figure 1

Please see figure in printed Reference Manual

It should be noted that near \(x=0\) this factor behaves as \(\frac{2x}{\sqrt{\pi}}\) and hence the accuracy is largely determined by the machine precision. Also for large negative \(x\), where the factor is \(\frac{2x}{\sqrt{\pi}}\), accuracy is mainly limited by machine precision.

\(\sqrt{\pi}\)

However, for large positive \(x\), the factor becomes \(\frac{2x}{\sqrt{\pi}}\) and to an extent relative accuracy is necessarily lost. The absolute accuracy \(E\) is given by
so absolute accuracy is guaranteed for all \( x \).

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S15 -- Approximations of Special Functions
S15AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S15AEF returns the value of the error function \( \text{erfx} \), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S15AEF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

Evaluates the error function,
\[
\text{erf } x = \frac{2}{\sqrt{\pi}} \int_0^t e^{-t^2} \, dt.
\]

For \(|x| \leq 2\),
\[
\text{erf } x = x > a \, T(t), \quad \text{where } t = -x - 1,
\]
\[
\text{erf } x = \text{sign}(x) \left[1 - \sum_{r=0}^{\infty} \frac{e^{-r^2}}{r!} \right], \quad \text{where } t = \frac{x}{\sqrt{\pi}}.
\]

For \(2 < |x| < x\),
\[
\text{erf } x = \text{sign}(x) \left[1 - \sum_{r=0}^{\infty} \frac{e^{-r^2}}{r!} \right], \quad \text{where } t = \frac{x}{\sqrt{\pi}}.
\]

For \(|x| > x\),
\[
\text{erf } x = \text{sign}(x).
\]

The value \(x\) is the value above which \(\text{erf } x = \pm 1\) within machine precision. Its value is given in the Users' Note for your implementation.

4. References


5. Parameters

1: \(X\) -- DOUBLE PRECISION
   Input
   On entry: the argument \(x\) of the function.

2: \(IFAIL\) -- INTEGER
   Input/Output
   On entry: \(IFAIL\) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   On exit: \(IFAIL = 0\) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter \(IFAIL\)
has been included for consistency with other routines in this chapter.

7. Accuracy

On a machine with approximately 11 significant figures the routine agrees with available tables to 10 figures and consistency checking with S15ADF of the relation

\[ \text{erf } x + \text{erfc } x = 1.0 \]

shows errors in at worst the 11th figure.

8. Further Comments

None.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
CHAPTER 15. CHAPTER N

This routine evaluates an approximation to the Bessel Function of the second kind \( Y_0(x) \).

Note: \( Y_0(x) \) is undefined for \( x \leq 0 \) and the routine will fail for such arguments.

The routine is based on four Chebyshev expansions:

For \( 0 < x \leq 8 \),
\[
Y_0(x) = \frac{\ln x}{4} \sum_{r=0}^{2} a_r T_r(t) + b_r T_r(t), \quad \text{with} \quad t = 2\left(\frac{x}{\pi}\right) - 1,
\]
where \( T_r \) are Chebyshev polynomials.

For \( x > 8 \),
\[
Y_0(x) = \sum_{r=0}^{2} c_r T_r(t),
\]
where \( t = 2\left(\frac{x}{\pi}\right) - 1 \) and \( T_r \) are Chebyshev polynomials.

For \( x \) near zero, \( Y_0(x) \approx \frac{\ln x}{\pi} \left( 1 + \gamma \right) \), where \( \gamma \) denotes Euler's constant. This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision.

For very large \( x \), it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of \( Y_0(x) \); only the amplitude, \( \sqrt{x} \), can be calculated.
determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision: the routine will fail if x>1/machine precision (see the Users’ Note for your implementation for details).

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument x of the function. Constraint: X > 0.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   X is too large. On soft failure the routine returns the amplitude of the Y oscillation, \sqrt{2/(\pi)x}.

IFAIL= 2
   X <= 0.0, Y is undefined. On soft failure the routine returns zero.

7. Accuracy

Let (delta) be the relative error in the argument and E be the absolute error in the result. (Since Y (x) oscillates about zero, absolute error and not relative error is significant, except for very small x.)

If (delta) is somewhat larger than the machine representation
error (e.g. if \( \delta \) is due to data errors etc), then \( E \) and \( \delta \) are approximately related by

\[
E^* = |x Y(x)| \delta
\]

provided \( E \) is also within machine bounds. Figure 1 displays the behaviour of the amplification factor \( |x Y(x)| \).

Figure 1
Please see figure in printed Reference Manual

However, if \( \delta \) is of the same order as the machine representation errors, then rounding errors could make \( E \) slightly larger than the above relation predicts.

For very small \( x \), the errors are essentially independent of \( \delta \) and the routine should provide relative accuracy bounded by the machine precision.

For very large \( x \), the above relation ceases to apply. In this region, \( Y(x) \approx \frac{\pi}{\pi x} \). The amplitude

\[
\sin(x- \frac{\pi}{4})
\]

can be calculated with reasonable accuracy for all \( x \), but

\[
\sin(x- \frac{\pi}{4})
\]

cannot. If \( x- \frac{\pi}{4} \) is written as \( 2N\pi+(\theta) \)

\[
\begin{align*}
\theta & \in \left[0, 2\pi\right) \\
N & \in \mathbb{Z}
\end{align*}
\]

where \( N \) is an integer and \( 0 \leq \theta < 2\pi \), then \( \sin(x- \frac{\pi}{4}) \) is determined by \( \theta \) only. If \( x > \frac{\pi}{4} \), \( \theta \) cannot be determined with any accuracy at all. Thus if \( x \) is greater than, or of the order of the inverse of machine precision, it is impossible to calculate the phase of \( Y(x) \) and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example
The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S17 -- Approximations of Special Functions

S17ADF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17ADF returns the value of the Bessel Function $Y_1(x)$, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S17ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Bessel Function of the second kind $Y_1(x)$.

Note: $Y_1(x)$ is undefined for $x \leq 0$ and the routine will fail for such arguments.

The routine is based on four Chebyshev expansions:

For $0 < x \leq 8$,

\[
Y_1(x) = -\frac{\ln x}{x} + \sum_{r=0}^{\infty} a_r T_r(t) - \frac{\ln x}{x} + \sum_{r=0}^{\infty} b_r T_r(t), \quad \text{with } t = \frac{1}{\pi (x)} - 1;
\]
For $x > 8$,

$$Y(x) = \frac{2}{\pi x} \left\{ P(x) \sin(x-3) + Q(x) \cos(x-3) \right\}$$

where $P(x) = \sum_{r=0}^{2} c_r T(t)$, $Q(x) = -\sum_{r=0}^{2} d_r T(t)$, with $t = 2(\frac{x}{\pi}) - 1$.

For $x$ near zero, $Y(x) \approx -\frac{1}{\pi x}$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision. For extremely small $x$, there is a danger of overflow in calculating $\frac{1}{\pi x}$ and for such arguments the routine will fail.

For very large $x$, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of $Y(x)$, only the amplitude, $\frac{2}{\pi x}$, can be determined and this is returned on soft failure. The range for which this occurs is roughly related to machine precision; the routine will fail if $x > 1/machine\ precision$ (see the Users’ Note for your implementation for details).

4. References


5. Parameters

1: X -- DOUBLE PRECISION  
   Input  
   On entry: the argument x of the function. Constraint: \( X > 0 \).  
2: IFAIL -- INTEGER  
   Input/Output  
   On entry: IFAIL must be set to 0, -1 or 1. For users not  
   familiar with this parameter (described in the Essential  
   Introduction) the recommended value is 0.  
   On exit: IFAIL = 0 unless the routine detects an error (see  
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   X is too large. On soft failure the routine returns the
   / 2
   amplitude of the Y oscillation, / -----.  
   \( \frac{1}{(\pi)x} \)

IFAIL= 2
   X <= 0.0, Y is undefined. On soft failure the routine
   returns zero.

IFAIL= 3
   X is too close to zero, there is a danger of overflow. On
   soft failure, the routine returns the value of Y (x) at the
   smallest valid argument.

7. Accuracy

Let (\( \delta \)) be the relative error in the argument and E be the
absolute error in the result. (Since Y (x) oscillates about zero,
absolute error and not relative error is significant, except for
very small x.)

If (\( \delta \)) is somewhat larger than the machine precision (e.g. if
(\( \delta \)) is due to data errors etc), then E and (\( \delta \)) are
approximately related by:
\[ E^0 = |xY(x) - Y(x)| \delta \]

(provided \( E \) is also within machine bounds). Figure 1 displays the behaviour of the amplification factor \( |xY(x) - Y(x)| \).

Figure 1
Please see figure in printed Reference Manual

However, if \( \delta \) is of the same order as machine precision, then rounding errors could make \( E \) slightly larger than the above relation predicts.

For very small \( x \), absolute error becomes large, but the relative error in the result is of the same order as \( \delta \).

For very large \( x \), the above relation ceases to apply. In this region, \( Y(x) \approx \frac{3(\pi)}{2} \sin(x - \frac{3(\pi)}{4}) \). The amplitude \( \frac{3(\pi)}{4} \) can be calculated with reasonable accuracy for all \( x \), but \( \sin(x - \frac{3(\pi)}{4}) \) cannot. If \( x - \frac{3(\pi)}{4} \) is written as \( 2N(\pi) + (\theta) \) where \( N \) is an integer and \( 0 < (\theta) < 2(\pi) \), then \( \sin(x - \frac{3(\pi)}{4}) \) is determined by \( -1 \) (\( \theta \)) only. If \( x > (\delta) \), \( (\theta) \) cannot be determined with any accuracy at all. Thus if \( x \) is greater than, or of the order of, the inverse of the machine precision, it is impossible to calculate the phase of \( Y(x) \) and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation.
Library software and should be available on-line.

S17 -- Approximations of Special Functions
S17AEF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AEF returns the value of the Bessel Function $J_0(x)$, via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S17AEF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Bessel Function of the first kind $J_0(x)$.

Note: $J_0(-x)=J_0(x)$, so the approximation need only consider $x>0$.

The routine is based on three Chebyshev expansions:

For $0<x\leq 8$, 

$$J_0(x) = \sum_{r=0}^{2} a_r T_r(t), \text{ with } t=2(1-x) - 1.$$ 

For $x>8$, 

$$J_0(x) = \frac{1}{\pi x} \left\{ \frac{(\pi)}{4} \left[ P(x) \cos(x-\pi/4) - Q(x) \sin(x-\pi/4) \right] \right\} -.$$ 

where $P(x) = \sum b_r T_r(t),$
\[ 0 = \ldots r r \]
\[ \begin{array}{c}
0 \\
r=0
\end{array} \]
\[ \begin{array}{c}
2 \\
8 \quad (8)
\end{array} \]
\[ and \quad Q(x) = - > c T(t), \quad with \quad t = 2(\ldots -1) \]
\[ \begin{array}{c}
0 \\
x = r r \quad (x)
\end{array} \]
\[ \begin{array}{c}
r=0
\end{array} \]

For \( x \) near zero, \( J(x)^{\approx}1 \). This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision.

For very large \( x \), it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of \( J(x) \); only the amplitude, \( \sqrt{\pi |x|} \), can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision; the routine will fail if \(|x|^{\approx}1/machine \) precision (see the Users’ Note for your implementation).

4. References


5. Parameters

1: \( X -- DOUBLE \) PRECISION Input
   On entry: the argument \( x \) of the function.

2: \( IFAIL -- INTEGER \) Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
7. Accuracy

Let $(\delta)$ be the relative error in the argument and $E$ be the absolute error in the result. (Since $J(x)$ oscillates about zero, absolute error and not relative error is significant.)

If $(\delta)$ is somewhat larger than the machine precision (e.g. if $(\delta)$ is due to data errors etc), then $E$ and $(\delta)$ are approximately related by:

$$E \approx |xJ(x)|(\delta)$$

(provided $E$ is also within machine bounds). Figure 1 displays the behaviour of the amplification factor $|xJ(x)|$.

Figure 1
Please see figure in printed Reference Manual

However, if $(\delta)$ is of the same order as machine precision, then rounding errors could make $E$ slightly larger than the above relation predicts.

For very large $x$, the above relation ceases to apply. In this

$$J(x) \approx \frac{\sqrt{2}}{\sqrt{\pi}x} \cos\left(x - \frac{\pi}{4}\right)$$

region, $J(x) \approx \frac{\sqrt{2}}{\sqrt{\pi}x} \cos\left(x - \frac{\pi}{4}\right)$. The amplitude

$$\frac{\sqrt{2}}{\sqrt{\pi}\sqrt{x}}$$

can be calculated with reasonable accuracy for all $x$ but $\cos(x - \frac{\pi}{4})$ cannot. If $x - \frac{\pi}{4}$ is written as

$$2N\pi + \theta$$

where $N$ is an integer and $0 \leq \theta < 2\pi$, then

$$\cos(x - \frac{\pi}{4}) \approx \cos(2N\pi + \theta) = \cos(\theta)$$

and

$$\text{amp} \approx \frac{\sqrt{2}}{\sqrt{\pi}x} \cos(\theta)$$
then \( \cos\left(x - \frac{\pi}{4}\right) \) is determined by \( \theta \) only. If \( x \sim \delta \),

\( \theta \) cannot be determined with any accuracy at all. Thus if \( x \) is greater than, or of the order of, the inverse of the machine precision, it is impossible to calculate the phase of \( J_0(x) \) and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

S17 -- Approximations of Special Functions

S17AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AFF returns the value of the Bessel Function \( J_1(x) \), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S17AFF (X, IFAIL)}
\]

\[
\text{INTEGER IFAIL}
\]

\[
\text{DOUBLE PRECISION X}
\]

3. Description

This routine evaluates an approximation to the Bessel Function of the first kind \( J_1(x) \).
Note: $J(-x) = -J(x)$, so the approximation need only consider $x \geq 0$.

The routine is based on three Chebyshev expansions:

For $0 < x \leq 8$,

$$
J(x) = \sum_{r=0}^{8} a_r T_r(t), \quad \text{with} \quad t = 2(8-x) - 1.
$$

For $x > 8$,

$$
J(x) = \sum_{r=0}^{2} b_r T_r(t), \quad \text{where} \quad t = 2(8-x) - 1.
$$

For $x$ near zero, $J(x) \approx -1$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision.

For very large $x$, it becomes impossible to provide results with any reasonable accuracy (see Section 7), hence the routine fails. Such arguments contain insufficient information to determine the phase of oscillation of $J(x)$; only the amplitude, $\sqrt{\pi |x|}$, can be determined and this is returned on soft failure. The range for which this occurs is roughly related to the machine precision; the routine will fail if $|x| > 1/$machine precision (see the Users’ Note for your implementation for details).

4. References

5. Parameters

1: X -- DOUBLE PRECISION  
  On entry: the argument x of the function.

2: IFAIL -- INTEGER  
  On entry: IFAIL must be set to 0, -1 or 1. For users not
  familiar with this parameter (described in the Essential
  Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
  X is too large. On soft failure the routine returns the

    / 2
  1 \ (pi)|x|

7. Accuracy

Let (delta) be the relative error in the argument and E be the
absolute error in the result. (Since J (x) oscillates about zero,
absolute error and not relative error is significant.)

If (delta) is somewhat larger than machine precision (e.g. if
(delta) is due to data errors etc), then E and (delta) are
approximately related by:

    E^\{\pm\} = |xJ (x) - J (x)| (delta)
              0  1

(provided E is also within machine bounds). Figure 1 displays the
behaviour of the amplification factor |xJ (x) - J (x)|.
However, if (delta) is of the same order as machine precision, then rounding errors could make E slightly larger than the above relation predicts.

For very large x, the above relation ceases to apply. In this region, $J(x) \approx \frac{\cos(x - \frac{3\pi}{4})}{\sqrt{|x|}}$. The amplitude can be calculated with reasonable accuracy for all x but $\cos(x - \frac{3\pi}{4})$ cannot. If $x - \frac{3\pi}{4}$ is written as $2N\pi + \theta$ where N is an integer and $0 \leq \theta < 2\pi$, then $\cos(x - \frac{3\pi}{4})$ is determined by $\theta$ only. If $x > \frac{\delta}{\sqrt{3\pi}}$, $\theta$ cannot be determined with any accuracy at all. Thus if x is greater than, or of the order of, machine precision, it is impossible to calculate the phase of $J(x)$ and the routine must fail.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17AGF returns a value for the Airy function, Ai(x), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S17AGF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Airy function, Ai(x). It is based on a number of Chebyshev expansions:

For \( x \leq -5 \),

\[
Ai(x) = \frac{a(t) \sin z - b(t) \cos z}{1/4 (-x)^{3/4}}
\]

where \( z = \frac{\pi}{4} \sqrt{-1/3 - x} \), and \( a(t) \) and \( b(t) \) are expansions in the variable \( t = -2 (x)^{-1} \).

For \(-5 \leq x \leq 0 \),

\[
Ai(x) = f(t) - x g(t),
\]

where \( f \) and \( g \) are expansions in \( t = -2 (x)^{-1} \).

For \( 0 < x < 4.5 \),

\[
Ai(x) = e^{-3x/2} y(t),
\]
where \( y \) is an expansion in \( t = \frac{4x}{9} - 1 \).

For \( 4.5 \leq x < 9 \),

\[
-\frac{5x}{2} \\
Ai(x) = e^{u(t)},
\]

where \( u \) is an expansion in \( t = \frac{4x}{9} - 3 \).

For \( x \geq 9 \),

\[
-\frac{z}{e^{v(t)}} \\
Ai(x) = \frac{1}{\sqrt{x^{\frac{2}{3}}}},
\]

where \( z = -\sqrt[3]{x} \) and \( v \) is an expansion in \( t = 2(\frac{z}{3} - 1) \).

For \(|x| < \) the machine precision, the result is set directly to \( Ai(0) \). This both saves time and guards against underflow in intermediate calculations.

For large negative arguments, it becomes impossible to calculate the phase of the oscillatory function with any precision and so the routine must fail. This occurs if \( x < -\left( \frac{3}{2^{\varepsilon}} \right) \), where \( (\varepsilon) \) is the machine precision.

For large positive arguments, where \( Ai \) decays in an essentially exponential manner, there is a danger of underflow so the routine must fail.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION 
   Input
   On entry: the argument \( x \) of the function.
2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   X is too large and positive. On soft failure, the routine
   returns zero.

IFAIL= 2
   X is too large and negative. On soft failure, the routine
   returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence
absolute error is the appropriate measure. In the positive region
the function is essentially exponential-like and here relative
error is appropriate. The absolute error, E, and the relative
error, (epsilon), are related in principle to the relative error
in the argument, (delta), by

   | xAi'(x) |
   E' = |xAi'(x)|(delta), (epsilon)' = | ------- | (delta).
   | Ai(x) |

In practice, approximate equality is the best that can be
expected. When (delta), (epsilon) or E is of the order of the
machine precision, the errors in the result will be somewhat
larger.

For small x, errors are strongly damped by the function and hence
will be bounded by the machine precision.

For moderate negative x, the error behaviour is oscillatory but
the amplitude of the error grows like

   \( \frac{5}{4} \)
   (E) \ |x|
   amplitude (-------)^* -------
   (delta) \ /\(\pi\)
However the phase error will be growing roughly like \(-\sqrt[3]{|x|}\)
and hence all accuracy will be lost for large negative arguments
due to the impossibility of calculating \(\sin\) and \(\cos\) to any
accuracy if \(-\sqrt[3]{|x|} > \frac{2}{3}\) (\(\delta\)).

For large positive arguments, the relative error amplification is considerable:
\[
\frac{\epsilon}{3} \sim \sqrt{x}.
\]

This means a loss of roughly two decimal places accuracy for
arguments in the region of 20. However very large arguments are
not possible due to the danger of setting underflow and so the
errors are limited in practice.

8. Further Comments

None.

9. Example

The example program reads values of the argument \(x\) from a file,
evaluates the function at each value of \(x\) and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
S17AHF returns a value of the Airy function, Bi(x), via the routine name.

2. Specification

DOUBLE PRECISION FUNCTION S17AHF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Airy function Bi(x). It is based on a number of Chebyshev expansions.

For x<-5,

\[ Bi(x) = \frac{a(t)\cos z + b(t)\sin z}{1/4 \left(-\pi\right)^{2/3} (-x)} \]

where \( z = \frac{\pi}{4} + \sqrt{-x} \) and \( a(t) \) and \( b(t) \) are expansions in the variable \( t = -2(\frac{5}{x})^{-1} \).

For -5<=x<=0,

\[ Bi(x) = \sqrt{3}(f(t) + xg(t)) \]

where \( f \) and \( g \) are expansions in \( t = -2(\frac{5}{x})^{-1} \).

For 0<x<4.5,

\[ Bi(x) = e^{\frac{11x}{8}}y(t), \]

where \( y \) is an expansion in \( t = 4x/9 - 1 \).

For 4.5<=x<=9,
\[ 5x/2 \]
\[ Bi(x) = e^{v(t)}, \]

where \( v \) is an expansion in \( t = \frac{4x}{9} - 3 \).

For \( x \geq 9 \),

\[ \frac{z}{\left(\begin{array}{c}
\text{e}^u(t) \\
\frac{1}{4} \\
x
\end{array}\right)} \]

\[ Bi(x) = \frac{z}{\left(\begin{array}{c}
\text{e}^u(t) \\
\frac{1}{4} \\
x
\end{array}\right)} \]

where \( z = -\sqrt[3]{x} \) and \( u \) is an expansion in \( t = 2(\epsilon) - 1 \).

For \( |x| < \) the machine precision, the result is set directly to \( Bi(0) \). This both saves time and avoids possible intermediate underflows.

For large negative arguments, it becomes impossible to calculate the phase of the oscillating function with any accuracy so the

\[ \frac{2/3}{\left(\begin{array}{c}
3 \\
\epsilon
\end{array}\right)} \]

routine must fail. This occurs if \( x < -\left(\frac{\epsilon}{2(\epsilon)}\right) \), where \( \epsilon \) is the machine precision.

For large positive arguments, there is a danger of causing overflow since \( Bi \) grows in an essentially exponential manner, so the routine must fail.

4. References


5. Parameters

1: X -- DOUBLE PRECISION
   On entry: the argument \( x \) of the function.

2: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
X is too large and positive. On soft failure, the routine returns zero.

IFAIL= 2
X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region the function is essentially exponential-like and here relative error is appropriate. The absolute error, E, and the relative error, (\epsilon), are related in principle to the relative error in the argument, (\delta), by

\[ E \approx \frac{|xB_0'(x)|}{|B_0(x)|} \text{ and } \epsilon \approx \frac{|xB_0'(x)|}{|B_0(x)|} \delta. \]

In practice, approximate equality is the best that can be expected. When (\delta), (\epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, errors are strongly damped and hence will be bounded essentially by the machine precision.

For moderate to large negative x, the error behaviour is clearly oscillatory but the amplitude of the error grows like amplitude 5/4

\[ E \approx \frac{|xB_0'(x)|}{|B_0(x)|} \left( \frac{1}{\sqrt{\pi}} \right)^{5/4} \delta. \]

However the phase error will be growing roughly as \[ -\sqrt{3} |x| \text{ and } \]
hence all accuracy will be lost for large negative arguments.

This is due to the impossibility of calculating sin and cos to

\[ \frac{2}{3} \frac{1}{\sqrt{|x|}} \]

any accuracy if \( -\sqrt{|x|} > \frac{\text{epsilon}}{3} \frac{\delta}{\text{delta}} \).

For large positive arguments, the relative error amplification is considerable:

\[ \frac{\text{epsilon}}{3} \frac{\delta}{\text{delta}} \sqrt{|x|} \]

This means a loss of roughly two decimal places accuracy for arguments in the region of 20. However very large arguments are not possible due to the danger of causing overflow and errors are therefore limited in practice.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Ai(x), via the routine name.

2. Specification

\[
\text{DOUBLE PRECISION FUNCTION S17AJF (X, IFAIL)}
\]
\[
\text{INTEGER IFAIL}
\]
\[
\text{DOUBLE PRECISION X}
\]

3. Description

This routine evaluates an approximation to the derivative of the Airy function Ai(x). It is based on a number of Chebyshev expansions.

For \( x < -5 \),

\[
Ai'(x) = \sqrt{-x} [a(t) \cos z + \frac{b(t)}{4} \sin z],
\]

where
\[
z = \frac{(\pi)}{3} \left( \frac{2}{3} \right) + \frac{3}{4} \zeta,
\]

\[
\zeta = \sqrt{-x}
\]

and \( a(t) \) and \( b(t) \) are expansions in variable \( t = \left( \frac{x}{4} \right)^{\frac{1}{3}} \).

For \(-5 \leq x \leq 0\),

\[
Ai'(x) = x f(t) - g(t),
\]

where \( f \) and \( g \) are expansions in \( t = -2(\text{x})^{-1} \).

For \( 0 < x < 4.5 \),

\[
Ai'(x) = \frac{-11x}{8} e^{y(t)},
\]

where \( y(t) \) is an expansion in \( t = 4(\text{x})^{-1} \).

For \( 4.5 \leq x < 9 \),
\[-5x/2\]
\[Ai'(x) = e^{v(t)},\]
\(\text{where } v(t) \text{ is an expansion in } t = 4(-) - 3.\)
\((9)\)

For \(x \geq 9,\)
\[\frac{4}{3} - z\]
\[Ai'(x) = \sqrt{-x} e^{u(t)},\]
\(\text{where } z = -\sqrt{x} \text{ and } u(t) \text{ is an expansion in } t = 2(-) - 1.\)
\((18)\)

For \(|x| < \text{the square of the machine precision},\) the result is set directly to \(Ai'(0).\) This both saves time and avoids possible intermediate underflows.

For large negative arguments, it becomes impossible to calculate a result for the oscillating function with any accuracy and so the routine must fail. This occurs for \(x < \left(\frac{4}{7}\right)^{\sqrt{(\pi)}}\), where \((\text{epsilon})\) is the machine precision.

For large positive arguments, where \(Ai'\) decays in an essentially exponential manner, there is a danger of underflow so the routine must fail.

4. References


5. Parameters

1: \(X -- \text{DOUBLE PRECISION} \quad \text{Input}\)
\(\text{On entry: the argument } x \text{ of the function.}\)

2: \(IFAIL -- \text{INTEGER} \quad \text{Input/Output}\)
\(\text{On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.}\)

\(\text{On exit: IFAIL = 0 unless the routine detects an error (see}\)
6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   X is too large and positive. On soft failure, the routine returns zero.

IFAIL = 2
   X is too large and negative. On soft failure, the routine returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is the appropriate measure. In the positive region the function is essentially exponential in character and here relative error is needed. The absolute error, E, and the relative error, (epsilon), are related in principle to the relative error in the argument, (delta), by

\[ E = \frac{2}{x \text{Ai}(x)} \delta \]
\[ (\epsilon) = \frac{2}{|\text{Ai}'(x)|} \delta. \]

In practice, approximate equality is the best that can be expected. When (delta), (epsilon) or E is of the order of the machine precision, the errors in the result will be somewhat larger.

For small x, positive or negative, errors are strongly attenuated by the function and hence will be roughly bounded by the machine precision.

For moderate to large negative x, the error, like the function, is oscillatory; however the amplitude of the error grows like

\[ \frac{7/4}{|x|} \]

\[ \frac{1}{\sqrt{\pi}} \]

Therefore it becomes impossible to calculate the function with
any accuracy if \(|x| > \frac{7/4}{\pi}\). \((\delta)\)

For large positive \(x\), the relative error amplification is considerable:

\[
\frac{\epsilon}{3} \approx \frac{3}{\pi} \sqrt{x}.
\]

However, very large arguments are not possible due to the danger of underflow. Thus in practice error amplification is limited.

8. Further Comments

None.

9. Example

The example program reads values of the argument \(x\) from a file, evaluates the function at each value of \(x\) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
This routine calculates an approximate value for the derivative of the Airy function $Bi(x)$. It is based on a number of Chebyshev expansions.

For $x<-5$,

$$Bi'(x)=\sqrt{-x}[-a(t)\sin z+\frac{b(t)}{(\pi/2)^{3/2}}\cos z],$$

where $z=\frac{\pi}{4}+(\zeta)$, $(\zeta)=-\sqrt{-x}$ and $a(t)$ and $b(t)$ are expansions in the variable $t=-2(\frac{x}{5})^{-1}$.

For $-5\leq x\leq 0$,

$$Bi'(x)=\sqrt{3}(x f(t)+g(t)),$$

where $f$ and $g$ are expansions in $t=-2(\frac{x}{5})^{-1}$.

For $0<x<4.5$,

$$Bi'(x)=e^{\frac{3x}{2}}y(t),$$

where $y(t)$ is an expansion in $t=4x/9-1$.

For $4.5\leq x<9$,

$$Bi'(x)=e^{\frac{21x}{8}}u(t),$$

where $u(t)$ is an expansion in $t=4x/9-3$.

For $x\geq 9$,

$$Bi'(x)=\sqrt{x}e^{\frac{4z}{8}}v(t),$$
\[
\frac{2}{3} \quad (18)
\]
where \(z = -\sqrt{x}\) and \(v(t)\) is an expansion in \(t = 2(\frac{\pi}{z})^{-1}\).

For \(|x| < \) the square of the machine precision, the result is set directly to \(B_1'(0)\). This saves time and avoids possible underflows in calculation.

For large negative arguments, it becomes impossible to calculate a result for the oscillating function with any accuracy so the routine must fail. This occurs for \(x < \left(\frac{4}{\pi}\right)^{1/7} (\epsilon)\), where \((\epsilon)\) is the machine precision.

For large positive arguments, where \(B_1'\) grows in an essentially exponential manner, there is a danger of overflow so the routine must fail.

4. References


5. Parameters

1: \(X -- DOUBLE PRECISION\) \hspace{1cm} Input
On entry: the argument \(x\) of the function.

2: \(IFAIL -- INTEGER\) \hspace{1cm} Input/Output
On entry: \(IFAIL\) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: \(IFAIL = 0\) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

\(IFAIL= 1\)
\(X\) is too large and positive. On soft failure the routine returns zero.

\(IFAIL= 2\)
\(X\) is too large and negative. On soft failure the routine
returns zero.

7. Accuracy

For negative arguments the function is oscillatory and hence absolute error is appropriate. In the positive region the function has essentially exponential behaviour and hence relative error is needed. The absolute error, \( E \), and the relative error (\( \epsilon \)), are related in principle to the relative error in the argument (\( \delta \)), by

\[
|2| \quad \frac{2}{|x \, \text{Bi}(x)|} (\epsilon) = \frac{| \text{Bi}'(x) |}{| x \, \text{Bi}(x) |} (\delta).
\]

In practice, approximate equality is the best that can be expected. When (\( \delta \)), (\( \epsilon \)) or \( E \) is of the order of the machine precision, the errors in the result will be somewhat larger.

For small \( x \), positive or negative, errors are strongly attenuated by the function and hence will effectively be bounded by the machine precision.

For moderate to large negative \( x \), the error is, like the function, oscillatory. However, the amplitude of the absolute error grows like \( \frac{7/4}{\sqrt{\pi}} \). Therefore it becomes impossible to calculate the function with any accuracy if \( |x| > \frac{7/4}{\sqrt{\pi}} \).

For large positive \( x \), the relative error amplification is considerable: \( \frac{\epsilon}{3 \delta} \). However, very large arguments are not possible due to the danger of overflow. Thus in practice the actual amplification that occurs is limited.

8. Further Comments
9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S17DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17DCF returns a sequence of values for the Bessel functions \( Y(nu)(z) \) for complex \( z \), non-negative \( (nu) \) and \( n=0,1,...,N-1 \), with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DCF (FNU, Z, N, SCALE, CY, NZ, CWRK, IFAIL)
INTEGER N, NZ, IFAIL
DOUBLE PRECISION FNU
COMPLEX(KIND(1.0D0)) Z, CY(N), CWRK(N)
CHARACTER*1 SCALE
```

3. Description

This subroutine evaluates a sequence of values for the Bessel function \( Y(nu)(z) \), where \( z \) is complex, \(-\pi < \arg z \leq \pi\), and \( (nu) \) is the real, non-negative order. The \( N \)-member sequence is generated for orders \( (nu), (nu)+1,...,(nu)+N-1 \). Optionally, the sequence is scaled by the factor \( e^{-|\Im z|} \).

Note: although the routine may not be called with \( (nu) \) less than zero, for negative orders the formula
\[
Y((-nu)) = Y((nu)) \cos((\pi)(nu)) + J((nu)) \sin((\pi)(nu))
\]
may be used.
The routine is derived from the routine CBESY in Amos [2]. It is based on the relation
\[ Y_n(z) = \frac{\text{H}_n^1(z)}{2i} \text{H}_n^2(z) \]
where \( \text{H}_n^1(z) \) and \( \text{H}_n^2(z) \) are the Hankel functions of the first and second kinds respectively (see S17DLF).

When \( N \) is greater than 1, extra values of \( Y_n(z) \) are computed using recurrence relations.

For very large \( |z| \) or \((\nu) + N - 1\), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller \( |z| \) or \((\nu) + N - 1\), the computation is performed but results are accurate to less than half of machine precision. If \( |z| \) is very small, near the machine underflow threshold, or \((\nu) + N - 1\) is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: FNU -- DOUBLE PRECISION
Input
On entry: the order, \( (\nu) \), of the first member of the sequence of functions. Constraint: \( FNU \geq 0.0 \).

2: Z -- COMPLEX(KIND(1.0D0))
Input
On entry: the argument, \( z \), of the functions. Constraint: \( Z /= (0.0, 0.0) \).

3: N -- INTEGER
Input
On entry: the number, \( N \), of members required in the sequence \( Y_n(z), Y_{n+1}(z), \ldots, Y_{n+N-1}(z) \). Constraint: \( N \geq 1 \).
4: SCALE -- CHARACTER*1
   Input
   On entry: the scaling option.
   If SCALE = 'U', the results are returned unscaled.
   If SCALE = 'S', the results are returned scaled by the
   \(-|\text{Im}z|\)
   factor e . Constraint: SCALE = 'U' or 'S'.

5: CY(N) -- COMPLEX(KIND(1.0D)) array
   Output
   On exit: the N required function values: CY(i) contains
   \(Y^{(\nu)+i-1}(z)\), for i=1,2,...,N.

6: NZ -- INTEGER
   Output
   On exit: the number of components of CY that are set to zero
   due to underflow. The positions of such components in the
   array CY are arbitrary.

7: CWRK(N) -- COMPLEX(KIND(1.0D)) array
   Workspace

8: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not
   familiar with this parameter (described in the Essential
   Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see
   Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry FNU < 0.0,
   or \(Z = (0.0, 0.0)\),
   or \(N < 1\),
   or SCALE /= 'U' or 'S'.

IFAIL= 2
   No computation has been performed due to the likelihood of
   overflow, because ABS(Z) is less than a machine-dependent
   threshold value (given in the Users' Note for your
implementation).

IFAIL= 3
No computation has been performed due to the likelihood of overflow, because FNU + N - 1 is too large - how large depends on Z as well as the overflow threshold of the machine.

IFAIL= 4
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S17DCF are accurate to less than half of machine precision. This error exit may occur if either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S17DCF would be lost. This error exit may occur if either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 6
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DCF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DCF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S17DCF, the actual number of correct digits is limited, in general, by p-s, where s=max(1,|log |z||,|log (nu)|) represents the number of digits lost due to the argument reduction. Thus the larger the values of |z| and (nu), the less the precision in the result. If S17DCF is called with N>1, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S17DCF with different base values of (nu) and different N, the computed values may not agree exactly.
Empirical tests with modest values of \((\nu)\) and \(z\) have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S17DCF is approximately proportional to the value of \(N\), plus a constant. In general it is much cheaper to call S17DCF with \(N\) greater than 1, rather than to make \(N\) separate calls to S17DCF.

Paradoxically, for some values of \(z\) and \((\nu)\), it is cheaper to call S17DCF with a larger value of \(N\) than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different \(N\), and the costs in each region may differ greatly.

Note that if the function required is \(Y(x)\) or \(Y(x)\), i.e., \((\nu)\) = 0.0 or 1.0, where \(x\) is real and positive, and only a single unscaled function value is required, then it may be much cheaper to call S17ACF or S17ADF respectively.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order FNU, the second is a complex value for the argument, \(Z\), and the third is a value for the parameter SCALE. The program calls the routine with \(N = 2\) to evaluate the function for orders FNU and FNU + 1, and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
1. Purpose

S17DEF returns a sequence of values for the Bessel functions $J_{n+\nu}(z)$ for complex $z$, non-negative $\nu$ and $n=0,1,\ldots,N-1$, with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DEF (FNU, Z, N, SCALE, CY, NZ, IFAIL)
INTEGER N, NZ, IFAIL
DOUBLE PRECISION FNU
COMPLEX(KIND(1.0D0)) Z, CY(N)
CHARACTER*1 SCALE
```

3. Description

This subroutine evaluates a sequence of values for the Bessel function $J_{\nu}(z)$, where $z$ is complex, $-(\pi) < \arg z \leq (\pi)$, and $\nu$ is the real, non-negative order. The $N$-member sequence is generated for orders $\nu$, $\nu+1$, $\ldots$, $\nu+N-1$. Optionally, the sequence is scaled by the factor $e^{-|\text{Im} z|}$.

Note: although the routine may not be called with $\nu$ less than zero, for negative orders the formula

$$J_{\nu}(z) = J_{\nu}(z) \cos(\pi\nu) - Y_{\nu}(z) \sin(\pi\nu)$$

may be used (for the Bessel function $Y_{\nu}(z)$, see S17DCF).

The routine is derived from the routine CBESJ in Amos [2]. It is based on the relations

$$J_{\nu}(z) = e^{-(\nu)(\pi i/2)} I_{\nu-\nu}(iz), \ \text{Im} z > 0.0$$

and

$$J_{\nu}(z) = e^{-(\nu)(\pi i/2)} I_{\nu-\nu}(-iz), \ \text{Im} z < 0.0.$$

The Bessel function $I_{\nu}(z)$ is computed using a variety of techniques depending on the region under consideration.

When $N$ is greater than 1, extra values of $J_{\nu}(z)$ are computed using recurrence relations.

For very large $|z|$ or $(\nu+N-1)$, argument reduction will cause total loss of accuracy, and so no computation is performed. For
slightly smaller $|z|$ or $((\nu)+N-1)$, the computation is performed but results are accurate to less than half of machine precision. If Im $z$ is large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: FNU -- DOUBLE PRECISION
   On entry: the order, $(\nu)$, of the first member of the sequence of functions. Constraint: FNU $\geq 0.0$.

2: Z -- COMPLEX(KIND(1.0D0))
   On entry: the argument $z$ of the functions.

3: N -- INTEGER
   On entry: the number, N, of members required in the sequence $J_{(\nu)}(z), J_{(\nu)+1}(z), \ldots, J_{(\nu)+N-1}(z)$. Constraint: N $\geq 1$.

4: SCALE -- CHARACTER*1
   On entry: the scaling option.
   If SCALE = 'U', the results are returned unscaled.
   If SCALE = 'S', the results are returned scaled by the factor $e^{-|Imz|}$.
   Constraint: SCALE = 'U' or 'S'.

5: CY(N) -- COMPLEX(KIND(1.0D)) array
   On exit: the N required function values: CY(i) contains $J_{(\nu)+i-1}(z)$, for i=1,2,\ldots,N.

6: NZ -- INTEGER
   On exit: the number of components of CY that are set to zero due to underflow. If NZ > 0, then elements CY(N-NZ+1), CY(N-NZ+2),\ldots,CY(N) are set to zero.

7: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
  On entry FNU < 0.0,
  or N < 1,
  or SCALE /= 'U' or 'S'.

IFAIL= 2
  No computation has been performed due to the likelihood of overflow, because Im Z is larger than a machine-dependent threshold value (given in the Users' Note for your implementation). This error exit can only occur when SCALE = 'U'.

IFAIL= 3
  The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S17DEF are accurate to less than half of machine precision. This error exit may occur if either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 4
  No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S17DEF would be lost. This error exit may occur when either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
  No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DEF would have caused overflow or underflow.
7. Accuracy

All constants in subroutine S17DEF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used \( t \), then clearly the maximum number of correct digits in the results obtained is limited by \( p = \min(t, 18) \). Because of errors in argument reduction when computing elementary functions inside S17DEF, the actual number of correct digits is limited, in general, by \( p - s \), where

\[
10^{s = \max(1, |\log|(|z|), |\log(\nu)|)}
\]

represents the number of digits lost due to the argument reduction. Thus the larger the values of \( |z| \) and \( (\nu) \), the less the precision in the result. If S17DEF is called with \( N > 1 \), then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S17DEF with different base values of \( (\nu) \) and different \( N \), the computed values may not agree exactly. Empirical tests with modest values of \( (\nu) \) and \( z \) have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S17DEF is approximately proportional to the value of \( N \), plus a constant. In general it is much cheaper to call S17DEF with \( N \) greater than 1, rather than to make \( N \) separate calls to S17DEF.

Paradoxically, for some values of \( z \) and \( (\nu) \), it is cheaper to call S17DEF with a larger value of \( N \) than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different \( N \), and the costs in each region may differ greatly.

Note that if the function required is \( J_0(x) \) or \( J_1(x) \), i.e., \( (\nu) = 0 \) or 1, where \( x \) is real and positive, and only a single unscaled function value is required, then it may be much cheaper to call S17AEF or S17AFF respectively.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order \( FNU \), the second is a complex value for the argument, \( Z \), and the third is a value for the parameter \( SCALE \).
The program calls the routine with \( N = 2 \) to evaluate the function for orders \( FNU \) and \( FNU + 1 \), and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%% 
S17 -- Approximations of Special Functions
S17DGF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17DGF returns the value of the Airy function \( Ai(z) \) or its derivative \( Ai'(z) \) for complex \( z \), with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S17DGF (DERIV, Z, SCALE, AI, NZ, IFAIL)
INTEGER NZ, IFAIL
COMPLEX(KIND(1.0D0)) Z, AI
CHARACTER*1 DERIV, SCALE
```

3. Description

This subroutine returns a value for the Airy function \( Ai(z) \) or its derivative \( Ai'(z) \), where \( z \) is complex, \(-\pi < \arg z \leq \pi\).

\[
2z^{2/3}/z^{3/2}
\]

Optionally, the value is scaled by the factor \( e^{-2z^{1/3}} \).

The routine is derived from the routine CAIRY in Amos [2]. It is

\[
\frac{\sqrt[3]{2}}{z^{1/3}} K_{1/3}(w) \quad -z^{2/3} K_{2/3}(w)
\]

based on the relations \( Ai(z) = \ldots \), and \( Ai'(z) = \ldots \),
where \( K \) is the modified Bessel function and \( w = 2z\sqrt{z}/3 \).

For very large \(|z|\), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller \(|z|\), the computation is performed but results are accurate to less than half of machine precision. If \( \text{Re } w \) is too large, and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: DERIV -- CHARACTER*1
   On entry: specifies whether the function or its derivative is required.
   If DERIV = 'F', \( \text{Ai}(z) \) is returned.
   If DERIV = 'D', \( \text{Ai}'(z) \) is returned.
   Constraint: DERIV = 'F' or 'D'.

2: Z -- COMPLEX(KIND(1.0D0))
   On entry: the argument \( z \) of the function.

3: SCALE -- CHARACTER*1
   On entry: the scaling option.
   If SCALE = 'U', the result is returned unscaled.
   If SCALE = 'S', the result is returned scaled by the factor
   \( 2z\sqrt{z}/3 \).
   Constraint: SCALE = 'U' or 'S'.

4: AI -- COMPLEX(KIND(1.0D0))
   On exit: the required function or derivative value.
5: NZ -- INTEGER
   Output
On exit: NZ indicates whether or not AI is set to zero due to underflow. This can only occur when SCALE = 'U'.

If NZ = 0, AI is not set to zero.
If NZ = 1, AI is set to zero.

6: IFAIL -- INTEGER
   Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
On entry DERIV /= 'F' or 'D'.
or
SCALE /= 'U' or 'S'.

IFAIL = 2
No computation has been performed due to the likelihood of overflow, because Re w is too large, where \( w=2Z/\sqrt{3} \) -- how large depends on Z and the overflow threshold of the machine. This error exit can only occur when SCALE = 'U'.

IFAIL = 3
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the result returned by S17DGF is accurate to less than half of machine precision. This error exit may occur if ABS (Z) is greater than a machine-dependent threshold value (given in the Users’ Note for your implementation).

IFAIL = 4
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in the result returned by S17DGF would be lost. This error exit may occur if ABS(Z) is greater than a machine-dependent threshold value (given in the Users’ Note
for your implementation).

IFAIL= 5
No result is returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DGF would have caused overflow or underflow.

7. Accuracy
All constants in subroutine S17DGF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S17DGF, the actual number of correct digits is limited, in general, by p-s, where s=\max(1,\log |z|) represents the number of digits lost due to the argument reduction. Thus the larger the value of |z|, the less the precision in the result.

Empirical tests with modest values of z, checking relations between Airy functions Ai(z), Ai'(z), Bi(z) and Bi'(z), have shown errors limited to the least significant 3-4 digits of precision.

8. Further Comments
Note that if the function is required to operate on a real argument only, then it may be much cheaper to call S17AGF or S17AJF.

9. Example
The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the parameter DERIV, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine and prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
S17 -- Approximations of Special Functions
S17DHF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17DHF returns the value of the Airy function Bi(z) or its derivative Bi'(z) for complex z, with an option for exponential scaling.

2. Specification

SUBROUTINE S17DHF (DERIV, Z, SCALE, BI, IFAIL)
INTEGER IFAIL
COMPLEX(KIND(1.0D0)) Z, BI
CHARACTER*1 DERIV, SCALE

3. Description

This subroutine returns a value for the Airy function Bi(z) or its derivative Bi'(z), where z is complex, -(pi) < argz <= (pi).

\[ |\text{Re } (2z/\sqrt{3})| \]

Optionally, the value is scaled by the factor \( e^{\text{Re } z/3} \).

The routine is derived from the routine CBIRY in Amos [2]. It is based on the relations:

\[ \text{Bi}(z) = \frac{1}{z} \left( I_{-1/3}(w) + I_{1/3}(w) \right), \]

\[ \frac{z}{\sqrt{3}} \]

where \( I_{\text{nu}}(w) \) is the modified Bessel function and \( w = 2z/\sqrt{3} \).

For very large \( |z| \), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller \( |z| \), the computation is performed but results are accurate to less than half of machine precision. If \( \text{Re } z \) is too large, and the unscaled function is required, there is a risk of
overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: DERIV -- CHARACTER*1
   On entry: specifies whether the function or its derivative is required.
   If DERIV = 'F', Bi(z) is returned.
   If DERIV = 'D', Bi'(z) is returned.
   Constraint: DERIV = 'F' or 'D'.

2: Z -- COMPLEX(KIND(1.0D0))
   On entry: the argument z of the function.

3: SCALE -- CHARACTER*1
   On entry: the scaling option.
   If SCALE = 'U', the result is returned unscaled.
   If SCALE = 'S', the result is returned scaled by the factor |Re(2z/z/3)| .
   Constraint: SCALE = 'U' or 'S'.

4: BI -- COMPLEX(KIND(1.0D0))
   On exit: the required function or derivative value.

5: IFAIL -- INTEGER
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry DERIV /= 'F' or 'D'.

or
SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of overflow, because real(Z) is too large - how large depends on the overflow threshold of the machine. This error exit can only occur when SCALE = 'U'.

IFAIL= 3
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the result returned by S17DHF is accurate to less than half of machine precision. This error exit may occur if ABS(Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 4
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in the result returned by S17DHF would be lost. This error exit may occur if ABS(Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No result is returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S17DHF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S17DHF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S17DHF, the actual number of correct digits is limited, in general, by p-s, where s=max(1,|log |z||) represents the number of digits lost due to the argument reduction. Thus the larger the value of |z|, the less the precision in the result.
Empirical tests with modest values of \( z \), checking relations between Airy functions \( \text{Ai}(z), \text{Ai}'(z), \text{Bi}(z) \) and \( \text{Bi}'(z) \), have shown errors limited to the least significant 3-4 digits of precision.

8. Further Comments

Note that if the function is required to operate on a real argument only, then it may be much cheaper to call S17AHF or S17AKF.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the parameter DERIV, the second is a complex value for the argument, \( Z \), and the third is a value for the parameter SCALE. The program calls the routine and prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S17 -- Approximations of Special Functions

S17DLF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S17DLF returns a sequence of values for the Hankel functions

(1) \( H_n^{(1)}(z) \) or \( H_n^{(2)}(z) \) for complex \( z \), non-negative \( (nu) \) and

(2) \( (nu)+n \)

\( n=0,1,...,N-1 \), with an option for exponential scaling.

2. Specification

```
SUBROUTINE S17DLF (M, FNU, Z, N, SCALE, CY, NZ, IFAIL)
INTEGER M, N, NZ, IFAIL
DOUBLE PRECISION FNU, CY(N)
COMPLEX(KIND(1.0D0)) Z
```
3. Description

This subroutine evaluates a sequence of values for the Hankel function $H^{(1)}(z)$ or $H^{(2)}(z)$, where $z$ is complex, $-(\pi) < \arg z \leq (\pi)$, and $(\nu)$ is the real, non-negative order. The $N$-member sequence is generated for orders $(\nu), (\nu)+1,\ldots,(\nu)+N-1$.

Optionally, the sequence is scaled by the factor $e^{-iz}$ if the function is $H^{(1)}(z)$ or by the factor $e^{iz}$ if the function is $H^{(2)}(z)$.

Note: although the routine may not be called with $(\nu)$ less than zero, for negative orders the formulae

$$H^{(1)}(z) = e^{-(\nu)\pi i} H^{(2)}(z), \quad H^{(2)}(z) = e^{(\nu)\pi i} H^{(1)}(z)$$

may be used.

The routine is derived from the routine CBESH in Amos [2]. It is based on the relation

$$H^{(m)}(z) = e^{-(\nu)\pi i} K^{(m)}(ze^{\pi i}),$$

where $p=i$ ---- if $m=1$ and $p=-i$ ---- if $m=2$, and the Bessel function $K^{(2)}(z)$ is computed in the right half-plane only. Continuation of $K^{(2)}(z)$ to the left half-plane is computed in terms of the Bessel function $I^{(2)}(z)$. These functions are evaluated using a variety of different techniques, depending on the region under consideration.

When $N$ is greater than 1, extra values of $H^{(m)}(z)$ are computed using recurrence relations.
For very large \(|z|\) or \(((\nu)+N-1)\), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller \(|z|\) or \(((\nu)+N-1)\), the computation is performed but results are accurate to less than half of machine precision. If \(|z|\) is very small, near the machine underflow threshold, or \(((\nu)+N-1)\) is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: \(M\) -- INTEGER Input
   On entry: the kind of functions required.
   \((1)\)
   If \(M = 1\), the functions are \(H_{\nu}(z)\).
   \((2)\)
   If \(M = 2\), the functions are \(H_{\nu}(z)\).
   Constraint: \(M = 1\) or 2.

2: \(FNU\) -- DOUBLE PRECISION Input
   On entry: the order, \((\nu)\), of the first member of the sequence of functions. Constraint: \(FNU \geq 0.0\).

3: \(Z\) -- COMPLEX(KIND(1.0D0)) Input
   On entry: the argument \(z\) of the functions. Constraint: \(Z \neq (0.0, 0.0)\).

4: \(N\) -- INTEGER Input
   On entry: the number, \(N\), of members required in the sequence \(H_{\nu}, H_{\nu+1}, \ldots, H_{\nu+N-1}\). Constraint: \(N \geq 1\).

5: \(SCALE\) -- CHARACTER*1 Input
   On entry: the scaling option.
   If \(SCALE = 'U'\), the results are returned unscaled.
   If \(SCALE = 'S'\), the results are returned scaled by the
-iz
factor e when M = 1, or by the factor e when M = 2.
Constraint: SCALE = 'U' or 'S'.

6: CY(N) -- COMPLEX(KIND(1.0D)) array Output
On exit: the N required function values: CY(i) contains (M)
H
, for i=1,2,...,N.
(nu)+i-1

7: NZ -- INTEGER Output
On exit: the number of components of CY that are set to zero
due to underflow. If NZ > 0, then if Imz>0.0 and M = 1, or
Imz<0.0 and M = 2, elements CY(1), CY(2),...,CY(NZ) are set
to zero. In the complementary half-planes, NZ simply states
the number of underflows, and not which elements they are.

8: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry M /= 1 and M /= 2,
or FNU < 0.0,
or Z = (0.0, 0.0),
or N < 1,
or SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of
overflow, because ABS(Z) is less than a machine-dependent
threshold value (given in the Users’ Note for your
implementation).
IFAIL= 3
No computation has been performed due to the likelihood of
overflow, because FNU + N - 1 is too large — how large
depends on Z and the overflow threshold of the machine.

IFAIL= 4
The computation has been performed, but the errors due to
argument reduction in elementary functions make it likely
that the results returned by S17DLF are accurate to less
than half of machine precision. This error exit may occur if
either ABS(Z) or FNU + N - 1 is greater than a machine-
dependent threshold value (given in the Users’ Note for
your implementation).

IFAIL= 5
No computation has been performed because the errors due to
argument reduction in elementary functions mean that all
precision in results returned by S17DLF would be lost. This
error exit may occur when either of ABS(Z) or FNU + N - 1 is
greater than a machine-dependent threshold value (given in
the Users’ Note for your implementation).

IFAIL= 6
No results are returned because the algorithm termination
condition has not been met. This may occur because the
parameters supplied to S17DLF would have caused overflow or
underflow.

7. Accuracy

All constants in subroutine S17DLF are given to approximately 18
digits of precision. Calling the number of digits of precision in
the floating-point arithmetic being used t, then clearly the
maximum number of correct digits in the results obtained is
limited by p=min(t,18). Because of errors in argument reduction
when computing elementary functions inside S17DLF, the actual
number of correct digits is limited, in general, by p-s, where
s=\max(1,\log |z|,|\log (nu)|) represents the number of digits
lost due to the argument reduction. Thus the larger the values of
|z| and (nu), the less the precision in the result. If S17DLF is
called with N>1, then computation of function values via
recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are
computed by calls to S17DLF with different base values of (nu)
and different N, the computed values may not agree exactly.
Empirical tests with modest values of (nu) and z have shown that
the discrepancy is limited to the least significant 3-4 digits of
precision.
8. Further Comments

The time taken by the routine for a call of S17DLF is approximately proportional to the value of \( N \), plus a constant. In general it is much cheaper to call S17DLF with \( N \) greater than 1, rather than to make \( N \) separate calls to S17DLF.

Paradoxically, for some values of \( z \) and \( (\nu) \), it is cheaper to call S17DLF with a larger value of \( N \) than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different \( N \), and the costs in each region may differ greatly.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the kind of function, \( M \), the second is a value for the order \( FNU \), the third is a complex value for the argument, \( Z \), and the fourth is a value for the parameter \( SCALE \). The program calls the routine with \( N = 2 \) to evaluate the function for orders \( FNU \) and \( FNU + 1 \), and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
3. Description

This routine evaluates an approximation to the modified Bessel Function of the second kind $K(x)$.

Note: $K(x)$ is undefined for $x \leq 0$ and the routine will fail for such arguments.

The routine is based on five Chebyshev expansions:

For $0 < x \leq 1$,

$K(x) = -\ln x \cdot a\sum r=0^{2} T_r(t) + b\sum r=0 T_r(t)$, where $t = 2x - 1$;

For $1 < x \leq 2$,

$K(x) = e^{-x} \cdot c\sum r=0 T_r(t)$, where $t = 2x - 3$;

For $2 < x \leq 4$,

$K(x) = e^{-x} \cdot d\sum r=0 T_r(t)$, where $t = x - 3$;

For $x > 4$,

$K(x) = e^{-x} \cdot e_{9-x}\sum r=0 T_r(t)$, where $t = \frac{9-x}{1+x}$

For $x$ near zero, $K(x) \approx -(\gamma) - \ln x$, where $(\gamma)$ denotes Euler's constant. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision.
For large $x$, where there is a danger of underflow due to the smallness of $K$, the result is set exactly to zero.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION
   Input
   On entry: the argument $x$ of the function. Constraint: $X > 0$.

2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   $X \leq 0.0$, $K$ is undefined. On soft failure the routine returns zero.

7. Accuracy

Let $(\delta)$ and $(\epsilon)$ be the relative errors in the argument and result respectively.

If $(\delta)$ is somewhat larger than the machine precision (i.e., if $(\delta)$ is due to data errors etc), then $(\epsilon)$ and $(\delta)$ are approximately related by:

$$| \frac{xK(x)}{K(x)} | \approx \frac{1}{| \delta |}.$$ 

Figure 1 shows the behaviour of the error amplification factor $| xK(x) |$. 
Figure 1
Please see figure in printed Reference Manual

However, if (delta) is of the same order as machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x, the amplification factor is approximately | --- |, which implies strong attenuation of the error, but in general (epsilon) can never be less than the machine precision.

For large x, (epsilon)^x = x(delta) and we have strong amplification of the relative error. Eventually K, which is asymptotically 0

-e
\sqrt{x}

given by ---, becomes so small that it cannot be calculated without underflow and hence the routine will return zero. Note that for large x the errors will be dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the appropriate the Users’ Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18ADF returns the value of the modified Bessel Function $K_1(x)$ via the routine name.

2. Specification

```plaintext
DOUBLE PRECISION FUNCTION S18ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the modified Bessel Function of the second kind $K_1(x)$.

Note: $K_1(x)$ is undefined for $x \leq 0$ and the routine will fail for such arguments.

The routine is based on five Chebyshev expansions:

For $0 < x \leq 1$,

$$
K_1(x) = -\frac{\ln x}{x} + \sum_{r=0}^{1} a_r T_r(t),
$$

where $t = 2x - 1$.

For $1 < x \leq 2$,

$$
K_1(x) = e^{-x} \sum_{r=0}^{1} c_r T_r(t),
$$

where $t = 2x - 3$.

For $2 < x \leq 4$,

$$
K_1(x) = e^{-x} \sum_{r=0}^{1} d_r T_r(t),
$$

where $t = x - 3$. 

Where $T_r(t)$ are Chebyshev polynomials of the first kind.
For $x > 4$,
\[
K(x) = \frac{-x}{e^{9-x}} > \frac{e^{-t}}{1 + x}, \quad \text{where } t = \frac{r}{1 + x}, \quad \sqrt{x} r = 0
\]

For $x$ near zero, $K(x) \approx -\frac{1}{x}$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision. For very small $x$ on some machines, it is impossible to calculate $-\frac{1}{x}$ without overflow and the routine must fail.

For large $x$, where there is a danger of underflow due to the smallness of $K$, the result is set exactly to zero.

4. References


5. Parameters

1: X -- DOUBLE PRECISION
   Input
   On entry: the argument $x$ of the function. Constraint: $X > 0$.

2: IFAIL -- INTEGER
   Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   $X \leq 0.0$, $K$ is undefined. On soft failure the routine returns zero.

IFAIL= 2
CHAPTER 15. CHAPTER N

X is too small, there is a danger of overflow. On soft failure the routine returns approximately the largest representable value.

7. Accuracy

Let \( (\Delta) \) and \( (\varepsilon) \) be the relative errors in the argument and result respectively.

If \( (\Delta) \) is somewhat larger than the machine precision (i.e., if \( (\Delta) \) is due to data errors etc), then \( (\varepsilon) \) and \( (\Delta) \) are approximately related by:

\[
\begin{vmatrix}
\frac{|xK(x) - K(x)|}{0 1} \\
|K(x)|
\end{vmatrix}
\approx \frac{|\Delta|}{K(x)}.
\]

Figure 1 shows the behaviour of the error amplification factor

\[
\begin{vmatrix}
\frac{|xK(x) - K(x)|}{0 1} \\
|K(x)|
\end{vmatrix}
\]

However if \( (\Delta) \) is of the same order as the machine precision, then rounding errors could make \( (\varepsilon) \) slightly larger than the above relation predicts.

For small \( x \), \( (\varepsilon) \approx (\Delta) \) and there is no amplification of errors.

For large \( x \), \( (\varepsilon) \approx x(\Delta) \) and we have strong amplification of the relative error. Eventually \( K \), which is asymptotically given by \( \frac{1}{e^{\frac{-x}{x}}} \), becomes so small that it cannot be calculated without underflow and hence the routine will return zero. Note that for large \( x \) the errors will be dominated by those of the Fortran intrinsic function EXP.
8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
\[ I(x) = e^{a \sum_{r=0}^{\infty} \frac{t(t)}{r!}} \quad (4) \]

For \(4 < x \leq 12\),
\[ I(x) = e^{b \sum_{r=0}^{\infty} \frac{t}{r!}} \quad (4) \]

For \(x > 12\),
\[ I(x) = e^{c \sum_{r=0}^{\infty} \frac{t}{r!}} \quad (12) \]

For small \(x\), \(I(x) \approx 1\). This approximation is used when \(x\) is sufficiently small for the result to be correct to machine precision.

For large \(x\), the routine must fail because of the danger of overflow in calculating \(e^x\).

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument \(x\) of the function.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
IFAIL = 1

X is too large. On soft failure the routine returns the approximate value of I (x) at the nearest valid argument.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e., if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

\[
\frac{|xI(x)|}{|I(x)|} \approx \frac{1}{\epsilon_0} (\delta).
\]

Figure 1 shows the behaviour of the error amplification factor

\[
\frac{|xI(x)|}{|I(x)|} \approx \frac{1}{\epsilon_0}.
\]

However if (delta) is of the same order as machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small \( x \) the amplification factor is approximately \( \frac{1}{\epsilon_0} \), which implies strong attenuation of the error, but in general (epsilon) can never be less than the machine precision.

For large \( x \), \( \epsilon_0 \approx x (\delta) \) and we have strong amplification of errors. However the routine must fail for quite moderate values of \( x \), because \( I(x) \) would overflow; hence in practice the loss of accuracy for large \( x \) is not excessive. Note that for large \( x \) the errors will be dominated by those of the Fortran intrinsic function EXP.
8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S18 -- Approximations of Special Functions
S18AFF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18AFF returns a value for the modified Bessel Function $I_1(x)$, via the routine name.

2. Specification

    DOUBLE PRECISION FUNCTION S18AFF (X, IFAIL)
    INTEGER IFAIL
    DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the modified Bessel Function of the first kind $I_1(x)$.

Note: $I_{-x} = -I_1(x)$, so the approximation need only consider $x >= 0$.

The routine is based on three Chebyshev expansions:

For $0 < x <= 4$, 

    --', (x)^2
\[ I(x) = x^{\frac{a}{t}} \text{ for } 4 < x \leq 12, \]
\[ I(x) = e^{\frac{x-8}{t}} \text{ for } x > 12, \]
\[ I(x) \approx x \text{ for small } x, \]
\[ I(x) = 2(\frac{-1}{x}) \text{ for large } x. \]

For small \( x \), \( I(x) \approx x \). This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision.

For large \( x \), the routine must fail because \( I(x) \) cannot be represented without overflow.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION
   \text{ Input}
   \text{ On entry: the argument } x \text{ of the function.}

2: \( IFAIL \) -- INTEGER
   \text{ Input/Output}
   \text{ On entry: } IFAIL \text{ must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.}
   \text{ On exit: } IFAIL = 0 \text{ unless the routine detects an error (see Section 6).}

6. Error Indicators and Warnings

Errors detected by the routine:
IFAIL= 1

X is too large. On soft failure the routine returns the
approximate value of I (x) at the nearest valid argument.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument
and result respectively.

If (delta) is somewhat larger than the machine precision (i.e.,
if (delta) is due to data errors etc), then (epsilon) and (delta)
are approximately related by:

\[
\frac{|xI(x) - I(x)|}{|I(x)|} \approx \frac{(\epsilon)}{(\delta)}
\]

Figure 1 shows the behaviour of the error amplification factor

However if (delta) is of the same order as machine precision,
then rounding errors could make (epsilon) slightly larger than
the above relation predicts.

For small x, (epsilon) ~ (delta) and there is no amplification of
errors.

For large x, (epsilon) ~ x(epsilon) and we have strong amplification
of errors. However the routine must fail for quite moderate
values of x because I (x) would overflow; hence in practice the

loss of accuracy for large x is not excessive. Note that for
large x, the errors will be dominated by those of the Fortran
intrinsic function EXP.

8. Further Comments

For details of the time taken by the routine see the Users’ Note
for your implementation.
9. Example

The example program reads values of the argument x from a file, evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S18 -- Approximations of Special Functions
S18DCF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S18DCF returns a sequence of values for the modified Bessel functions $K_n(z)$ for complex $z$, non-negative $(\nu)$ and $(\nu)+n, n=0,1,...,N-1,$ with an option for exponential scaling.

2. Specification

```fortran
SUBROUTINE S18DCF (FNU, Z, N, SCALE, CY, NZ, IFAIL)
INTEGER N, NZ, IFAIL
DOUBLE PRECISION FNU
COMPLEX(KIND(1.0D0)) Z, CY(N)
CHARACTER*1 SCALE
```

3. Description

This subroutine evaluates a sequence of values for the modified Bessel function $K_n(z)$, where $z$ is complex, $-(\pi) < \arg z \leq (\nu)$, $(\nu)$ is the real, non-negative order. The $N$-member sequence is generated for orders $(\nu), (\nu)+1,...,(\nu)+N-1$. Optionally, the sequence is scaled by the factor $e^{i\pi\nu}$.

The routine is derived from the routine CBESK in Amos [2].

Note: although the routine may not be called with $(\nu)$ less than zero, for negative orders the formula $K_{-(\nu)}(z)=K_{(\nu)}(z)$ may be used.
When \( N \) is greater than 1, extra values of \( K(z) \) are computed using recurrence relations.

For very large \(|z|\) or \(((\nu)+N-1)\), argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller \(|z|\) or \(((\nu)+N-1)\), the computation is performed but results are accurate to less than half of machine precision. If \(|z|\) is very small, near the machine underflow threshold, or \(((\nu)+N-1)\) is too large, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: \( \text{FNU -- DOUBLE PRECISION Input} \)  
   On entry: the order, \((\nu)\), of the first member of the sequence of functions. Constraint: \( \text{FNU} \geq 0.0 \).

2: \( Z -- \text{COMPLEX(KIND(1.0D0)) Input} \)  
   On entry: the argument \( z \) of the functions. Constraint: \( Z \neq (0.0, 0.0) \).

3: \( N -- \text{INTEGER Input} \)  
   On entry: the number, \( N \), of members required in the sequence \( K(\nu)(z) \), \( K((\nu)+1)(z) \), ..., \( K((\nu)+N-1)(z) \). Constraint: \( N \geq 1 \).

4: \( \text{SCALE -- CHARACTER*1 Input} \)  
   On entry: the scaling option. 
   If \( \text{SCALE} = 'U' \), the results are returned unscaled. 
   If \( \text{SCALE} = 'S' \), the results are returned scaled by the \( z \) factor \( e^{z} \). Constraint: \( \text{SCALE} = 'U' \) or \( 'S' \).

5: \( \text{CY(N) -- COMPLEX(KIND(1.0D)) array Output} \) On exit: the \( N \) required function values: \( \text{CY}(i) \) contains
K (z), for i=1,2,...,N.
(nu)+i-1

6: NZ -- INTEGER Output
On exit: the number of components of CY that are set to zero
due to underflow. If NZ > 0 and Rez>=0.0, elements CY(1),CY
(2),...,CY(NZ) are set to zero. If Rez<0.0, NZ simply states
the number of underflows, and not which elements they are.

7: IFAIL -- INTEGER Input/Output
On entry: IFAIL must be set to 0, -1 or 1. For users not
familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see
Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry FNU < 0.0,
or Z = (0.0, 0.0),
or N < 1,
or SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of
overflow, because ABS(Z) is less than a machine-dependent
threshold value (given in the Users’ Note for your
implementation).

IFAIL= 3
No computation has been performed due to the likelihood of
overflow, because FNU + N - 1 is too large - how large
depends on Z and the overflow threshold of the machine.

IFAIL= 4
The computation has been performed, but the errors due to
argument reduction in elementary functions make it likely
that the results returned by S18DCF are accurate to less
than half of machine precision. This error exit may occur if
either $\text{ABS}(z)$ or $\text{FNU} + N - 1$ is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S18DCF would be lost. This error exit may occur when either $\text{ABS}(z)$ or $\text{FNU} + N - 1$ is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 6
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S18DCF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S18DCF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used $t$, then clearly the maximum number of correct digits in the results obtained is limited by $p=\min(t,18)$. Because of errors in argument reduction when computing elementary functions inside S18DCF, the actual number of correct digits is limited, in general, by $p-s$, where $s=\max(1,\log_{10}|z|,\log_{10} (\text{nu}))$ represents the number of digits lost due to the argument reduction. Thus the larger the values of $|z|$ and $(\text{nu})$, the less the precision in the result. If S18DCF is called with $N>1$, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S18DCF with different base values of $(\text{nu})$ and different $N$, the computed values may not agree exactly. Empirical tests with modest values of $(\text{nu})$ and $z$ have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S18DCF is approximately proportional to the value of $N$, plus a constant. In general it is much cheaper to call S18DCF with $N$ greater than 1, rather than to make $N$ separate calls to S18DCF.

Paradoxically, for some values of $z$ and $(\text{nu})$, it is cheaper to
call S18DCF with a larger value of N than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different N, and the costs in each region may differ greatly.

Note that if the function required is \( K_0(x) \) or \( K_1(x) \), i.e.,
\[
\begin{align*}
(nu) &= 0.0 \text{ or } 1.0, \\
\end{align*}
\]
where \( x \) is real and positive, and only a single function value is required, then it may be much cheaper to call S18ACF, S18ADF, S18CCF(*) or S18CDF(*), depending on whether a scaled result is required or not.

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order FNU, the second is a complex value for the argument, Z, and the third is a value for the parameter SCALE. The program calls the routine with \( N = 2 \) to evaluate the function for orders FNU and FNU + 1, and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
3. Description

This subroutine evaluates a sequence of values for the modified Bessel function $I_{\nu}(z)$, where $z$ is complex, $-(\pi) < \text{arg} z \leq \nu$ ($\pi$), and $\nu$ is the real, non-negative order. The $N$-member sequence is generated for orders $\nu$, $\nu+1$, ..., $\nu+N-1$.

Optionally, the sequence is scaled by the factor $e^{-|\text{Re}z|}$.

The routine is derived from the routine CBESI in Amos [2].

Note: although the routine may not be called with $\nu$ less than zero, for negative orders the formula

$$I_{\nu}(z) = I_{-\nu}(z) + \frac{\sin(\pi \nu)}{(\pi \nu)} K_{\nu}(z)$$

may be used (for $-(\nu)$, $\nu$, $\pi$, $\nu$, $\pi$, $\nu$, see S18DCF).

When $N$ is greater than 1, extra values of $I_{\nu}(z)$ are computed using recurrence relations.

For very large $|z|$ or $(\nu+N-1)$, argument reduction will cause total loss of accuracy, and so no computation is performed. For slightly smaller $|z|$ or $(\nu+N-1)$, the computation is performed but results are accurate to less than half of machine precision. If $\text{Re}(z)$ is too large and the unscaled function is required, there is a risk of overflow and so no computation is performed. In all the above cases, a warning is given by the routine.

4. References


5. Parameters

1: FNU -- DOUBLE PRECISION Input
   On entry: the order, $\nu$, of the first member of the sequence of functions. Constraint: FNU $\geq 0.0$. 

```fortran
DOUBLE PRECISION FNU
COMPLEX(KIND(1.0D0)) Z, CY(N)
CHARACTER*1 SCALE
```
2: Z -- COMPLEX(KIND(1.0DO))  
   On entry: the argument z of the functions.

3: N -- INTEGER  
   On entry: the number, N, of members required in the sequence
   \( I(z), I(z+1), \ldots, I(z+N-1) \) of the functions. Constraint: \( N \geq 1 \).

4: SCALE -- CHARACTER*1  
   On entry: the scaling option.
   If SCALE = 'U', the results are returned unscaled.
   If SCALE = 'S', the results are returned scaled by the factor \( e^{-|Re(z)|} \).
   Constraint: SCALE = 'U' or 'S'.

5: CY(N) -- COMPLEX(KIND(1.0D)) array  
   On exit: the N required function values: CY(i) contains
   \( I(z+i-1) \) for \( i=1,2,\ldots,N \).

6: NZ -- INTEGER  
   On exit: the number of components of CY that are set to zero due to underflow.
   If NZ > 0, then elements CY(N-NZ+1),CY(N-NZ+2),...,CY(N) are set to zero.

7: IFAIL -- INTEGER  
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1  
On entry FNU < 0.0,

or N < 1,
or SCALE /= 'U' or 'S'.

IFAIL= 2
No computation has been performed due to the likelihood of overflow, because real(Z) is greater than a machine-dependent threshold value (given in the Users' Note for your implementation). This error exit can only occur when SCALE = 'U'.

IFAIL= 3
The computation has been performed, but the errors due to argument reduction in elementary functions make it likely that the results returned by S18DEF are accurate to less than half of machine precision. This error exit may occur when either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 4
No computation has been performed because the errors due to argument reduction in elementary functions mean that all precision in results returned by S18DEF would be lost. This error exit may occur when either ABS(Z) or FNU + N - 1 is greater than a machine-dependent threshold value (given in the Users' Note for your implementation).

IFAIL= 5
No results are returned because the algorithm termination condition has not been met. This may occur because the parameters supplied to S18DEF would have caused overflow or underflow.

7. Accuracy

All constants in subroutine S18DEF are given to approximately 18 digits of precision. Calling the number of digits of precision in the floating-point arithmetic being used t, then clearly the maximum number of correct digits in the results obtained is limited by p=min(t,18). Because of errors in argument reduction when computing elementary functions inside S18DEF, the actual number of correct digits is limited, in general, by p-s, where s=max(1,|log |z||,|log (nu)|) represents the number of digits lost due to the argument reduction. Thus the larger the values of |z| and (nu), the less the precision in the result. If S18DEF is called with N>1, then computation of function values via recurrence may lead to some further small loss of accuracy.

If function values which should nominally be identical are computed by calls to S18DEF with different base values of (nu)
and different $N$, the computed values may not agree exactly. Empirical tests with modest values of $(\nu)$ and $z$ have shown that the discrepancy is limited to the least significant 3-4 digits of precision.

8. Further Comments

The time taken by the routine for a call of S18DEF is approximately proportional to the value of $N$, plus a constant. In general it is much cheaper to call S18DEF with $N$ greater than 1, rather than to make $N$ separate calls to S18DEF.

Paradoxically, for some values of $z$ and $(\nu)$, it is cheaper to call S18DEF with a larger value of $N$ than is required, and then discard the extra function values returned. However, it is not possible to state the precise circumstances in which this is likely to occur. It is due to the fact that the base value used to start recurrence may be calculated in different regions for different $N$, and the costs in each region may differ greatly.

Note that if the function required is $I_0(x)$ or $I_1(x)$, i.e.,

$$(\nu)=0.0 \text{ or } 1.0, \text{ where } x \text{ is real and positive, and only a single function value is required, then it may be much cheaper to call S18AEF, S18AFF, S18CEF(*) or S18CFF(*), depending on whether a scaled result is required or not.}$$

9. Example

The example program prints a caption and then proceeds to read sets of data from the input data stream. The first datum is a value for the order $FNU$, the second is a complex value for the argument, $Z$, and the third is a value for the parameter $SCALE$. The program calls the routine with $N = 2$ to evaluate the function for orders $FNU$ and $FNU + 1$, and it prints the results. The process is repeated until the end of the input data stream is encountered.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
CHAPTER 15. CHAPTER N

not included in the Foundation Library.

1. Purpose

S19AAF returns a value for the Kelvin function \( \text{ber} \ x \) via the
routine name.

2. Specification

```plaintext
DOUBLE PRECISION FUNCTION S19AAF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Kelvin function
\( \text{ber} x \).

Note: \( \text{ber} (-x) = \text{ber} x \), so the approximation need only consider
\( x > 0.0 \).

The routine is based on several Chebyshev expansions:

For \( 0 \leq x \leq 5 \),

\[
\begin{align*}
\text{ber} x &= \sum_{r=0}^{(5)} \left( \frac{x}{r} \right)^{4r} T_r(t) \\
&= \sum_{r=0}^{(5)} \left( \frac{x}{r} \right)^{4r} T_r \left( 2 \left( \frac{x}{r} \right) - 1 \right)
\end{align*}
\]

For \( x > 5 \),

\[
\begin{align*}
\text{ber} x &= \frac{x^{\sqrt{2}}}{\sqrt{\pi} x^{\frac{1}{2}}} \left[ (1 + a(t)) \cos(\alpha) + b(t) \sin(\alpha) \right] \\
&\quad - \frac{x^{-\sqrt{2}}}{\sqrt{\pi} x^{\frac{1}{2}}} \left[ (1 + c(t)) \sin(\beta) + d(t) \cos(\beta) \right]
\end{align*}
\]

where \( (\alpha) = \frac{\pi}{2} \) and \( (\beta) = \frac{\pi}{2} + \frac{\pi}{2} \).
and a(t), b(t), c(t), and d(t) are expansions in the variable 

\[
t = \frac{1}{x}.
\]

When \( x \) is sufficiently close to zero, the result is set directly to \( \text{ber} = 0 = 1.0 \).

For large \( x \), there is a danger of the result being totally inaccurate, as the error amplification factor grows in an essentially exponential manner; therefore the routine must fail.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION Input
   On entry: the argument \( x \) of the function.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL = 1
   On entry ABS(X) is too large for an accurate result to be returned. On soft failure, the routine returns zero.

7. Accuracy

Since the function is oscillatory, the absolute error rather than the relative error is important. Let \( E \) be the absolute error in the result and \( (\delta) \) be the relative error in the argument. If \( (\delta) \) is somewhat larger than the machine precision, then we have:

\[
\left| x \right| \ll \varepsilon
\]
\[ E^{-\frac{x}{\sqrt{2}}} = \frac{\delta}{\sqrt{2}} \]

(provided \( E \) is within machine bounds).

For small \( x \) the error amplification is insignificant and thus the absolute error is effectively bounded by the machine precision.

For medium and large \( x \), the error behaviour is oscillatory and

\[ \frac{1}{\sqrt{2}} \]

its amplitude grows like \( \frac{x}{\sqrt{2}} \). Therefore it is not possible to calculate the function with any accuracy when

\[ \frac{x}{\sqrt{2}} > \frac{\delta}{2\pi} \]

the minimum value of \( x \) for which the function overflows.

8. Further Comments

For details of the time taken by the routine see the Users' Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

S19 -- Approximations of Special Functions
S19ABF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose
S19ABF returns a value for the Kelvin function $\text{bei} \, x$ via the routine name.

2. Specification

```plaintext
DOUBLE PRECISION FUNCTION S19ABF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Kelvin function $\text{bei} \, x$.

Note: $\text{bei}(-x)=\text{beix}$, so the approximation need only consider $x\geq 0.0$.

The routine is based on several Chebyshev expansions:

For $0\leq x \leq 5$,

\[
\text{bei} \, x = \sum_{r=0}^{4} a_r T_r(t), \quad t = 2^{(-r)} - 1;
\]

For $x > 5$,

\[
\text{bei} \, x = e^{x/\sqrt{2}} \left[ (1 - a(t)) \sin(\alpha) - b(t) \cos(\alpha) \right]
\]

\[
\times \left[ (1 - c(t)) \cos(\beta) - d(t) \sin(\beta) \right]
\]

\[
\sqrt{2\pi x}
\]

where $\alpha = \frac{\pi}{8}$, $\beta = \frac{\pi}{8}$.

and \( a(t), b(t), c(t), \) and \( d(t) \) are expansions in the variable
\[
t = \frac{-1}{x}.
\]

When \( x \) is sufficiently close to zero, the result is computed as
\[
\frac{2}{x} \sin(x) \approx -i.
\]
If this result would underflow, the result returned is
\[
\frac{4}{x} \sin(x) = 0.0.
\]

For large \( x \), there is a danger of the result being totally inaccurate, as the error amplification factor grows in an essentially exponential manner; therefore the routine must fail.

4. References


5. Parameters

1: \( X \) -- DOUBLE PRECISION
   Input
   On entry: the argument \( x \) of the function.

2: \( IFAIL \) -- INTEGER
   Input/Output
   On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

\( IFAIL = 1 \)
On entry \( \text{ABS}(X) \) is too large for an accurate result to be returned. On soft failure, the routine returns zero.

7. Accuracy

Since the function is oscillatory, the absolute error rather than the relative error is important. Let \( E \) be the absolute error in the function, and \( \Delta \) be the relative error in the argument. If \( \Delta \) is somewhat larger than the machine precision, then we have:
\[
\begin{array}{|c|c|}
\hline
x & E = -(-\text{ber} x + \text{bei} x)(\delta) \\
\hline
1 & 1 \\
\sqrt{2} & 1 \\
\hline
\end{array}
\]

(provided \( E \) is within machine bounds).

For small \( x \) the error amplification is insignificant and thus the absolute error is effectively bounded by the machine precision.

For medium and large \( x \), the error behaviour is oscillatory and

\[
\frac{x}{\sqrt{\pi}} \approx e^{-\frac{x}{\sqrt{2}}}.
\]

its amplitude grows like \( e^{-\frac{x}{\sqrt{2}}} \). Therefore it is

impossible to calculate the functions with any accuracy when

\[
\frac{x}{\sqrt{\pi}} > \frac{\delta}{\sqrt{\pi}}. \]

Note that this value of \( x \) is much smaller than the minimum value of \( x \) for which the function overflows.

8. Further Comments

For details of the time taken by the routine see the Users’ Note for your implementation.

9. Example

The example program reads values of the argument \( x \) from a file, evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

S19 -- Approximations of Special Functions

S19ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

S19ACF returns a value for the Kelvin function \( \ker x \), via the routine name.

2. Specification

```plaintext
DOUBLE PRECISION FUNCTION S19ACF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Kelvin function \( \ker x \).

Note: for \( x<0 \) the function is undefined and at \( x=0 \) it is infinite so we need only consider \( x>0 \).

The routine is based on several Chebyshev expansions:

For \( 0<x\leq1 \),

\[
\begin{align*}
\ker x &= -f(t) \log x + \frac{(\pi)^2}{16} x g(t) + y(t) \\
&\quad \text{where } f(t), g(t) \text{ and } y(t) \text{ are expansions in the variable } t=2x -1;
\end{align*}
\]

For \( 1<x\leq3 \),

\[
\begin{align*}
\ker x &= \exp(-\frac{11}{16}x)q(t) \\
&\quad \text{where } q(t) \text{ is an expansion in the variable } t=x-2;
\end{align*}
\]

For \( x>3 \),

\[
\begin{align*}
\ker x &= \frac{-x}{\sqrt{2x}} \left(\frac{1}{x} \right) \\
&\quad \text{where } e^{(x^2/4-1/16)} - [((1+(-c(t))\cos(beta))-d(t)\sin(beta)] \\
&\quad \text{with } \beta = \pi \text{ and } c(t), d(t) \text{ are expansions in the variable } t=x-2;
\end{align*}
\]
where \((\beta) = \ldots \), and \(c(t)\) and \(d(t)\) are expansions in the variable \(t = -1\).

When \(x\) is sufficiently close to zero, the result is computed as

\[
\text{ker } x = -(\gamma) - \log(\sqrt{-x}) + \log(\sqrt{2})
\]

and when \(x\) is even closer to zero, simply as

\[
\text{ker } x = -(\gamma) - \log(-)\sqrt{2}
\]

For large \(x\), \(\text{ker } x\) is asymptotically given by

\[
\sqrt{-x}\sqrt{\pi - \sqrt{2}}
\]

this becomes so small that it cannot be computed without underflow and the routine fails.

4. References


5. Parameters

1: \(X\) -- DOUBLE PRECISION Input
   On entry: the argument \(x\) of the function. Constraint: \(X > 0\).

2: \(IFAIL\) -- INTEGER Input/Output
   On entry: \(IFAIL\) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

   On exit: \(IFAIL = 0\) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:
IFAIL= 1
On entry X is too large, the result underflows. On soft failure, the routine returns zero.

IFAIL= 2
On entry X ≤ 0, the function is undefined. On soft failure the routine returns zero.

7. Accuracy

Let E be the absolute error in the result, (epsilon) be the relative error in the result and (delta) be the relative error in the argument. If (delta) is somewhat larger than the machine precision, then we have:

\[ \left| \frac{x}{\sqrt{2}} \right| E = \left| \frac{-\ker x + \kei x}{1 + 1} \right| (\delta), \]

\[ \left| \frac{\ker x + \kei x}{x + 1} \right| \]

\[ \left( \epsilon \right) = \left| \frac{-\ker x}{1 + 1} \right| (\delta). \]

For very small x, the relative error amplification factor is approximately given by

\[ \left| \frac{1}{\log x} \right| \]

which implies a strong attenuation of relative error. However, (epsilon) in general cannot be less than the machine precision.

For small x, errors are damped by the function and hence are limited by the machine precision.

For medium and large x, the error behaviour, like the function itself, is oscillatory, and hence only the absolute accuracy for the function can be maintained. For this range of x, the

\[ \frac{(\pi)x}{\sqrt{2}} \]

amplitude of the absolute error decays like

\[ \frac{-x}{\sqrt{2}} \]

which implies a strong attenuation of error. Eventually, ker x,

\[ \frac{-(\pi)}{\sqrt{2}} \]

which asymptotically behaves like

\[ \frac{-x}{\sqrt{2}} \]

becomes so
small that it cannot be calculated without causing underflow, and
the routine returns zero. Note that for large x the errors are
dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

Underflow may occur for a few values of x close to the zeros of
ker x, below the limit which causes a failure with IFAIL = 1.

9. Example

The example program reads values of the argument x from a file,
evaluates the function at each value of x and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.

1. Purpose

S19ADF returns a value for the Kelvin function keix via the
routine name.

2. Specification

DOUBLE PRECISION FUNCTION S19ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description

This routine evaluates an approximation to the Kelvin function
keix.

Note: for x<0 the function is undefined, so we need only consider
x>=0.

The routine is based on several Chebyshev expansions:
For 0<=x<=1,
\[
keix = \frac{2}{4} \pi x \left[ f(t) + \frac{1}{4} [-g(t) \log x + v(t)] \right]
\]
where \( f(t), g(t), \) and \( v(t) \) are expansions in the variable \( t = 2x - 1 \);

For 1<x<=3,
\[
keix = \exp(-\frac{x}{4}) u(t)
\]
where \( u(t) \) is an expansion in the variable \( t = x - 2 \);

For x>3,
\[
keix = \sqrt{\frac{\pi}{2x}} \left( 1 + \frac{x}{\sqrt{2}} \right) c(t) \sin(\beta) + \frac{x}{\sqrt{2}} d(t) \cos(\beta)
\]
where \( \beta = \frac{\pi}{4} \), and \( c(t) \) and \( d(t) \) are expansions in the variable \( t = \frac{1}{x} - 1 \);

For x<0, the function is undefined, and hence the routine fails and returns zero.

When \( x \) is sufficiently close to zero, the result is computed as
\[
keix = \frac{2}{4} \pi x \left( 1 - \gamma - \log \frac{\pi}{2} \right)
\]
and when \( x \) is even closer to zero simply as
\[
keix = \frac{\pi}{4}.
\]
For large $x$, $\text{keix}$ is asymptotically given by $\frac{-x}{\sqrt{2}}$ and this becomes so small that it cannot be computed without underflow and the routine fails.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
   On entry: the argument $x$ of the function. Constraint: $X \geq 0$.

2: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

IFAIL= 1
   On entry $X$ is too large, the result underflows. On soft failure, the routine returns zero.

IFAIL= 2
   On entry $X < 0$, the function is undefined. On soft failure the routine returns zero.

7. Accuracy

Let $E$ be the absolute error in the result, and $(\delta)$ be the relative error in the argument. If $(\delta)$ is somewhat larger than the machine representation error, then we have:

$$\frac{|x|}{E} = (-\text{ker} x + \text{kei} x) |(\delta)|.$$
For small $x$, errors are attenuated by the function and hence are limited by the machine precision.

For medium and large $x$, the error behaviour, like the function itself, is oscillatory and hence only absolute accuracy of the function can be maintained. For this range of $x$, the amplitude of

$$\sqrt{\frac{\pi}{2}} x^{-\frac{1}{2}}$$

the absolute error decays like

$$\sqrt{\frac{\pi}{2}} e^{-x\sqrt{2}}$$

which implies a strong attenuation of error. Eventually, $ke^{ix}$, which is

$$\sqrt{\frac{\pi}{2}} e^{-x\sqrt{2}}$$

asymptotically given by

$$\sqrt{\frac{\pi}{2}} e^{-x\sqrt{2}}$$

becomes so small that it cannot be calculated without causing underflow and therefore the routine returns zero. Note that for large $x$, the errors are dominated by those of the Fortran intrinsic function EXP.

8. Further Comments

Underflow may occur for a few values of $x$ close to the zeros of $ke^{ix}$, below the limit which causes a failure with $IFAIL = 1$.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% S20 -- Approximations of Special Functions S20ACF
S20ACF -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users’ Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.
1. Purpose

S20ACF returns a value for the Fresnel Integral \( S(x) \), via the routine name.

2. Specification

```fortran
DOUBLE PRECISION FUNCTION S20ACF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X
```

3. Description

This routine evaluates an approximation to the Fresnel Integral

\[
S(x) = \int_0^x \frac{\sin(t)}{\sqrt{\pi}} dt.
\]

Note: \( S(x) = -S(-x) \), so the approximation need only consider \( x \geq 0 \).

The routine is based on three Chebyshev expansions:

For \( 0 < x \leq 3 \),

\[
S(x) = \sum_{r=0}^{3} a_r T_4(t), \quad t = 2(x - 1) - 1
\]

For \( x > 3 \),

\[
S(x) = \frac{f(x)}{x^2} \cos\left(\frac{\pi}{2} x\right) - \frac{g(x)}{x^3} \sin\left(\frac{\pi}{2} x\right)
\]

where \( f(x) = \sum_{r=0}^{3} b_r T_4(t) \),

\[
T_4(t) = \frac{1}{\sqrt{2}} \left(2t^4 - 3t^2 + 1\right)
\]

and \( g(x) = \sum_{r=0}^{3} c_r T_4(t) \) with \( t = 2(x - 1) - 1 \).
CHAPTER 15. CHAPTER N

For small $x$, $S(x) \approx \frac{1}{x}$. This approximation is used when $x$ is sufficiently small for the result to be correct to machine precision. For very small $x$, this approximation would underflow; the result is then set exactly to zero.

$$\frac{1}{\pi}$$

For large $x$, $f(x) \approx \frac{1}{2}$ and $g(x) \approx \frac{1}{2}$. Therefore for $$\frac{1}{\pi}$$

moderately large $x$, when $\frac{1}{2}$ is negligible compared with $\frac{-1}{\pi x}$, the second term in the approximation for $x > 3$ may be dropped. For very large $x$, when $\frac{1}{\pi x}$ becomes negligible, $S(x) \approx \frac{1}{2}$. However, there will be considerable difficulties in calculating $\cos\left(\frac{\pi}{2}x\right)$ accurately before this final limiting value can be used. Since $\cos\left(\frac{\pi}{2}x\right)$ is periodic, its value is essentially determined by the fractional part of $x$. If $x = N + (\theta)$ where $N$ is an integer and $0 \leq \theta < 1$, then $\cos\left(\frac{\pi}{2}x\right)$ depends on $(\theta)$ and on $N$ modulo 4. By exploiting this fact, it is possible to retain significance in the calculation of $\cos\left(\frac{\pi}{2}x\right)$ either all the way to the very large $x$ limit, or at least until the integer part of $\frac{x}{2}$ is equal to the maximum integer allowed on the machine.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION

Input

On entry: the argument $x$ of the function.
2: IFAIL -- INTEGER
     Input/Output
     On entry: IFAIL must be set to 0, -1 or 1. For users not
     familiar with this parameter (described in the Essential
     Introduction) the recommended value is 0.
     On exit: IFAIL = 0 unless the routine detects an error (see
     Section 6).

6. Error Indicators and Warnings

There are no failure exits from this routine. The parameter IFAIL
has been included for consistency with other routines in this
chapter.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument
and result respectively.

If (delta) is somewhat larger than the machine precision (i.e.,
if (delta) is due to data errors etc), then (epsilon) and (delta)
are approximately related by:

\[ (\text{epsilon}) \approx \frac{(\pi) \ 2}{S(x)} \cdot (\text{delta}) \]

Figure 1 shows the behaviour of the error amplification factor

\[ \frac{(\pi) \ 2}{S(x)} \]

However if (delta) is of the same order as the machine precision,
then rounding errors could make (epsilon) slightly larger than
the above relation predicts.

For small x, (epsilon) \approx 3(delta) and hence there is only moderate
amplification of relative error. Of course for very small x where
the correct result would underflow and exact zero is returned,
relative error-control is lost.
For moderately large values of $x$,
\[
| (\pi/2) | = 2x \sin\left(\frac{\pi}{2}x\right) |(\text{delta})| \\
| (2) |
\]
and the result will be subject to increasingly large amplification of errors. However the above relation breaks down for large values of $x$ (i.e., when $x$ is of the order of the machine precision in this region the relative error in the result is essentially bounded by $\frac{x}{(\pi)x}$).

Hence the effects of error amplification are limited and at worst the relative error loss should not exceed half the possible number of significant figures.

8. Further Comments
None.

9. Example

The example program reads values of the argument $x$ from a file, evaluates the function at each value of $x$ and prints the results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
DOUBLE PRECISION FUNCTION S20ADF (X, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X

3. Description
This routine evaluates an approximation to the Fresnel Integral
\[
C(x) = \int_0^x \frac{\cos\left(\frac{t}{\sqrt{2}}\right) dt}{\left(\frac{\pi}{2}\right)}
\]

Note: \( C(x) = -C(-x) \), so the approximation need only consider \( x \geq 0.0 \)

The routine is based on three Chebyshev expansions:

For \( 0 < x \leq 3 \),
\[
C(x) = x^4 \sum_{r=0}^{(3)} a_r T_r(t), \quad t = 2(\sqrt{2} - 1) - 1
\]

For \( x > 3 \),
\[
C(x) = \sum_{r=0}^{(3)} \left( \frac{1}{\pi^2} \right)^{1/2} \frac{1}{\sqrt{x}} f(x) + \frac{1}{\pi^2} \frac{1}{\sqrt{x}} g(x)
\]
where
\[
f(x) = \sum_{r=0}^{(3)} b_r T_r(t), \quad t = 2(\sqrt{2} - 1) - 1
\]
and
\[
g(x) = \sum_{r=0}^{(3)} c_r T_r(t), \quad t = 2(\sqrt{2} - 1) - 1
\]

For small \( x \), \( C(x) \approx x \). This approximation is used when \( x \) is sufficiently small for the result to be correct to machine precision.

For large \( x \), \( f(x) \approx \frac{1}{\pi^2} \) and \( g(x) \approx \frac{1}{\pi^2} \frac{1}{\sqrt{2}} \). Therefore for
\[
\left(\frac{\pi}{2}\right)
\]
moderately large $x$, when \( \frac{1}{2^3} \) is negligible compared with \( \frac{1}{2} \), the second term in the approximation for $x > 3$ may be dropped. For very large $x$, when \( \frac{1}{\pi x} \) becomes negligible, $C(x) \approx -\frac{1}{2}$. However there will be considerable difficulties in calculating \( \sin\left(\frac{\pi}{2}x\right) \) accurately before this final limiting value can be used. Since $\sin\left(\frac{\pi}{2}x\right)$ is periodic, its value is essentially determined by the fractional part of $x$. If $x = N + \theta$, where $N$ is an integer and $0 \leq \theta < 1$, then $\sin\left(\frac{\pi}{2}x\right)$ depends on $(\frac{\pi}{2})$ $(\theta)$ and on $N$ modulo 4. By exploiting this fact, it is possible to retain some significance in the calculation of \( \sin\left(\frac{\pi}{2}x\right) \) either all the way to the very large $x$ limit, or at least until the integer part of $\frac{x}{2}$ is equal to the maximum integer allowed on the machine.

4. References


5. Parameters

1: $X$ -- DOUBLE PRECISION
   On entry: the argument $x$ of the function.

2: $IFAIL$ -- INTEGER
   On entry: $IFAIL$ must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: $IFAIL = 0$ unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
There are no failure exits from this routine. The parameter IFAIL has been included for consistency with other routines in this chapter.

7. Accuracy

Let (delta) and (epsilon) be the relative errors in the argument and result respectively.

If (delta) is somewhat larger than the machine precision (i.e. if (delta) is due to data errors etc), then (epsilon) and (delta) are approximately related by:

\[
| \frac{(\pi/2) \cos(\pi/2 - x)}{C(x)} | (\text{epsilon}) \approx \frac{\text{(delta)}}{C(x)}.
\]

Figure 1 shows the behaviour of the error amplification factor

\[
| \frac{(\pi/2) \cos(\pi/2 - x)}{C(x)} | (\text{epsilon}) = |2x \cos(\pi/2 - x)| (\text{delta}).
\]

However if (delta) is of the same order as the machine precision, then rounding errors could make (epsilon) slightly larger than the above relation predicts.

For small x, (epsilon) \approx (delta) and there is no amplification of relative error.

For moderately large values of x,

\[
| \frac{(\pi/2) \cos(\pi/2 - x)}{C(x)} | (\text{epsilon}) \approx |2x \cos(\pi/2 - x)| (\text{delta}).
\]

and the result will be subject to increasingly large amplification of errors. However the above relation breaks down for large values of x (i.e., when \( \frac{x}{2} \) is of the order of the machine precision).
machine precision); in this region the relative error in the
result is essentially bounded by \( \frac{\pi x}{2} \).

Hence the effects of error amplification are limited and at worst
the relative error loss should not exceed half the possible
number of significant figures.

8. Further Comments
None.

9. Example
The example program reads values of the argument \( x \) from a file,
evaluates the function at each value of \( x \) and prints the results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation
Library software and should be available on-line.
\[
\frac{1}{dt} \quad R(x, y) = -\frac{1}{C} \quad \frac{1}{2} \quad \sqrt{\int_{t+x(t+y)}^{y}}
\]

where \( x \geq 0 \) and \( y \neq 0 \).

This function, which is related to the logarithm or inverse hyperbolic functions for \( y \leq x \) and to inverse circular functions if \( x \leq y \), arises as a degenerate form of the elliptic integral of the first kind. If \( y < 0 \), the result computed is the Cauchy principal value of the integral.

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the system:

\[
\begin{align*}
x &= x, \\
y &= y, \\
(\mu) &= (x + 2y) / 3, \\
S &= (y - x) / 3(\mu). \\
\end{align*}
\]

\[
(\lambda) = y + 2 \sqrt{x y} \\
x &= (x + (\lambda)) / 4, \\
y &= (y + (\lambda)) / 4. \\
\]

The quantity \(|S|\) for \( n=0, 1, 2, 3, \ldots \) decreases with increasing \( n \), eventually \(|S| \sim 1/4 \). For small enough \( S \) the required function value can be approximated by the first few terms of the Taylor series about the mean. That is

\[
\begin{pmatrix}
2 & 3 & 4 & 5 \\
3S & S & 3S & 9S
\end{pmatrix}
\]
\[ R(x,y) = \frac{1}{10} + \frac{1}{7} + \frac{1}{8} + \frac{1}{22} \]

The truncation error involved in using this approximation is bounded by \(16|S|/(1-2|S|)\) and the recursive process is stopped when \(S\) is small enough for this truncation error to be negligible compared to the machine precision.

Within the domain of definition, the function value is itself representable for all representable values of its arguments. However, for values of the arguments near the extremes the above algorithm must be modified so as to avoid causing underflows or overflows in intermediate steps. In extreme regions arguments are pre-scaled away from the extremes and compensating scaling of the result is done before returning to the calling program.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input

2: Y -- DOUBLE PRECISION Input

On entry: the arguments x and y of the function, respectively. Constraint: \(X \geq 0.0\) and \(Y \neq 0.0\).

3: IFAIL -- INTEGER Input/Output

On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.

On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings
Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
On entry X < 0.0; the function is undefined.

IFAIL= 2
On entry Y = 0.0; the function is undefined.

On soft failure the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine precision. However round-off errors in internal arithmetic will result in slight loss of accuracy. This loss should never be excessive as the algorithm does not involve any significant amplification of round-off error. It is reasonable to assume that the result is accurate to within a small multiple of the machine precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the relationship of this function to the classical definitions of the elliptic integrals.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.
S21BBF returns a value of the symmetrised elliptic integral of the first kind, via the routine name.

2. Specification

DOUBLE PRECISION FUNCTION S21BBF (X, Y, Z, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X, Y, Z

3. Description

This routine calculates an approximation to the integral

\[
\int_{1}^{\infty} \frac{dt}{F^2 / \sqrt{(t+x)(t+y)(t+z)}}
\]

where \( x, y, z \geq 0 \) and at most one is zero.

The basic algorithm, which is due to Carlson \[2\] and \[3\], is to reduce the arguments recursively towards their mean by the rule:

\[
x = \min(x, y, z), \quad z = \max(x, y, z),
\]

\[
y = \text{remaining third intermediate value argument}.
\]

(This ordering, which is possible because of the symmetry of the function, is done for technical reasons related to the avoidance of overflow and underflow.)

\[
(mu) = \frac{x + y + 3z}{3}
\]

\[
X = \frac{(1-x)/(mu)}
\]

\[
Y = \frac{(1-y)/(mu)}
\]

\[
Z = \frac{(1-z)/(mu)}
\]

\[
(lambda) = \sqrt{xy + yz + zx}
\]
\[
x = \frac{(x + \lambda)}{4}
\]
\[
y = \frac{(y + \lambda)}{4}
\]
\[
z = \frac{(z + \lambda)}{4}
\]

\[
(\epsilon) = \max(|X|,|Y|,|Z|)
\]

and the function may be approximated adequately by a 5th order power series:

\[
R(x,y,z) = \frac{1}{(1- \frac{E}{10} - \frac{E}{24} - \frac{E}{44} - \frac{E}{14})}
\]

where \( E = X Y + Y Z + Z X \), \( E = X Y Z \).

The truncation error involved in using this approximation is bounded by \( \frac{\epsilon}{4(1-(\epsilon))} \) and the recursive process is stopped when this truncation error is negligible compared with the machine precision.

Within the domain of definition, the function value is itself representable for all representable values of its arguments. However, for values of the arguments near the extremes the above algorithm must be modified so as to avoid causing underflows or overflows in intermediate steps. In extreme regions arguments are pre-scaled away from the extremes and compensating scaling of the result is done before returning to the calling program.

4. References


CHAPTER 15. CHAPTER N


5. Parameters

1: X -- DOUBLE PRECISION Input
2: Y -- DOUBLE PRECISION Input
3: Z -- DOUBLE PRECISION Input
   On entry: the arguments x, y and z of the function.
   Constraint: X, Y, Z >= 0.0 and only one of X, Y and Z may be zero.
4: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL= 1
   On entry one or more of X, Y and Z is negative; the function is undefined.

IFAIL= 2
   On entry two or more of X, Y and Z are zero; the function is undefined.

   On soft failure, the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine precision. However round-off errors in internal arithmetic will result in slight loss of accuracy. This loss should never be excessive as the algorithm does not involve any significant amplification of round-off error. It is reasonable to assume that the result is accurate to within a small multiple of the machine precision.

8. Further Comments
Users should consult the Chapter Introduction which shows the relationship of this function to the classical definitions of the elliptic integrals.

If two arguments are equal, the function reduces to the elementary integral \( R \), computed by \texttt{S21BAF}.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

---

**S21 -- Approximations of Special Functions**

\texttt{S21BCF} -- NAG Foundation Library Routine Document

Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

\texttt{S21BCF} returns a value of the symmetrised elliptic integral of the second kind, via the routine name.

2. Specification

\begin{verbatim}
DOUBLE PRECISION FUNCTION S21BCF (X, Y, Z, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X, Y, Z
\end{verbatim}

3. Description

This routine calculates an approximate value for the integral

\[
\int_{D}^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}
\]

\( D \) is

3 / 2 / 3
\[
R(x,y,z) = - \int_{D}^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}
\]

\[
\int_{D}^{\infty} \frac{dt}{\sqrt{(t+x)(t+y)(t+z)}}
\]
where \( x, y \geq 0 \), at most one of \( x \) and \( y \) is zero, and \( z > 0 \).

The basic algorithm, which is due to Carlson [2] and [3], is to reduce the arguments recursively towards their mean by the rule:

\[
\begin{align*}
   x &= x^0 \\
   y &= y^0 \\
   z &= z^0 \\
   (\mu) &= (x + y + 3z) / 5 \\
   X &= (1-x)/(\mu) \\
   Y &= (1-y)/(\mu) \\
   Z &= (1-z)/(\mu) \\
   (\lambda) &= (x/y + y/z + z/x) \\
   x &= (x + (\lambda)) / 4 \\
   y &= (y + (\lambda)) / 4 \\
   z &= (z + (\lambda)) / 4 \\

\end{align*}
\]

For \( n \) sufficiently large,

\[
\begin{align*}
   (\epsilon) &= \max(|X|, |Y|, |Z|) / n \\
   (\epsilon) &= \max(|X|, |Y|, |Z|) / n \\

\end{align*}
\]

and the function may be approximated adequately by a 5th order

\[
\begin{align*}
   (-1)^n \\
   \lambda = \max(|X|, |Y|, |Z|)^4 \\

\end{align*}
\]
power series \( R(x,y,z)=\frac{3}{\sqrt{D}} \)

\[ \sum_{m=0}^{\infty} \frac{(z+(\lambda))^m}{z^m} + \frac{4}{7n} \left[ \frac{3}{n} \right. \left. + \frac{1}{3n} + \frac{1}{3n} + \frac{1}{3n} + \frac{1}{n} + \frac{1}{n} + \frac{1}{n} + \frac{1}{n} \right] \]

\[ \cdot \frac{1}{1-(\epsilon)} \logo{\mu n} \]

where

\( (m) = \binom{m}{n} \)

\( S = \frac{x+y+3z}{2m} \)

The truncation error in this expansion is bounded by

\[ \frac{6 \epsilon}{\sqrt{n}} \]

and the recursive process is terminated when

\[ \frac{3}{1-(\epsilon)} \]

this quantity is negligible compared with the machine precision.

The routine may fail either because it has been called with arguments outside the domain of definition, or with arguments so extreme that there is an unavoidable danger of setting underflow or overflow.

\[-\frac{3}{2}\]

Note: \( R(x,x,x) = x \), so there exists a region of extreme arguments for which the function value is not representable.

4. References


5. Parameters

1: X -- DOUBLE PRECISION Input
2: Y -- DOUBLE PRECISION Input
3: Z -- DOUBLE PRECISION Input
   On entry: the arguments x, y and z of the function.
   Constraint: X, Y >= 0.0, Z > 0.0 and only one of X and Y may be zero.
4: IFAIL -- INTEGER Input/Output
   On entry: IFAIL must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential Introduction) the recommended value is 0.
   On exit: IFAIL = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry IFAIL = 0 or -1, explanatory error messages are output on the current error message unit (as defined by X04AAF).

IFAIL = 1
   On entry either X or Y is negative, or both X and Y are zero; the function is undefined.

IFAIL = 2
   On entry Z <= 0.0; the function is undefined.

IFAIL = 3
   On entry either Z is too close to zero or both X and Y are too close to zero: there is a danger of setting overflow.

IFAIL = 4
   On entry at least one of X, Y and Z is too large: there is a danger of setting underflow.

   On soft failure the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will result in slight loss of accuracy. This loss should never be excessive as the algorithm does not involve any significant amplification of round-off error. It is reasonable to assume that the result is accurate to within a small multiple of the machine precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the relationship of this function to the classical definitions of the elliptic integrals.

9. Example

This example program simply generates a small set of non-extreme arguments which are used with the routine to produce the table of low accuracy results.

The example program is not reproduced here. The source code for all example programs is distributed with the NAG Foundation Library software and should be available on-line.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
S21 -- Approximations of Special Functions
S21 -- NAG Foundation Library Routine Document
Note: Before using this routine, please read the Users' Note for your implementation to check implementation-dependent details. The symbol (*) after a NAG routine name denotes a routine that is not included in the Foundation Library.

1. Purpose

S21BDF returns a value of the symmetrised elliptic integral of the third kind, via the routine name.

2. Specification

```plaintext
DOUBLE PRECISION FUNCTION S21BDF (X, Y, Z, R, IFAIL)
INTEGER IFAIL
DOUBLE PRECISION X, Y, Z, R
```

3. Description

This routine calculates an approximation to the integral

\[
\int_{0}^{\infty} \frac{3}{t} \, dt
\]
\[
R(x,y,z,(\rho)) = - \int_0^2 \frac{\sqrt{t+\rho}}{(t+x)(t+y)(t+z)} dt
\]

where \( x, y, z \geq 0 \), \( \rho \neq 0 \) and at most one of \( x, y \) and \( z \) is zero.

If \( p < 0 \), the result computed is the Cauchy principal value of the integral.

The basic algorithm, which is due to Carlson \[2\] and \[3\], is to reduce the arguments recursively towards their mean by the rule:

\[
x = x_0
\]

\[
y = y_0
\]

\[
z = z_0
\]

\[
(\rho) = (\rho)_0
\]

\[
(\mu) = \frac{(x+y+z+2(\rho))}{5}
\]

\[
X = \frac{1-x}{(\mu)}
\]

\[
Y = \frac{1-y}{(\mu)}
\]

\[
Z = \frac{1-z}{(\mu)}
\]

\[
P = \frac{1-(\rho)}{(\mu)}
\]

\[
(\lambda) = \frac{x+y+z+2(\rho)}{4}
\]

\[
x = \frac{(x+\lambda)}{4}
\]

\[
y = \frac{(y+\lambda)}{4}
\]
\[
\begin{align*}
\frac{z}{n+1} &= \frac{z + (\lambda)}{4} \\
\frac{\rho}{n+1} &= \frac{(\rho) + (\lambda)}{4} \\
\frac{\alpha}{n} &= \left[ \frac{\rho}{n} \right] \frac{1}{n} + \frac{\rho}{n} + \frac{\rho}{n} + \frac{\rho}{n} + \frac{\rho}{n} \\
\frac{\beta}{n} &= \frac{(\rho) + (\lambda) + (\lambda)}{n} \\
\end{align*}
\]

For \( n \) sufficiently large,

\[
\epsilon = \max(\frac{|X|}{n}, |Y|, |Z|, |P|)^{\frac{1}{4}}
\]

and the function may be approximated by a 5th order power series

\[
R(x, y, z, \rho) = \sum_{m=0}^{n-1} \frac{a_m}{4^m} (\alpha) (\beta)
\]

where \( a_m = \frac{(X + Y + Z + 2P)}{2^m} \).

The truncation error in this expansion is bounded by

\[
\epsilon \leq \frac{3}{6} \epsilon (1 - \epsilon) \quad \text{and the recursion process is terminated when this quantity is negligible compared with the machine precision. The routine may fail either because it has been called with arguments outside the domain of definition or}
\]
with arguments so extreme that there is an unavoidable danger of
setting underflow or overflow.

\[ \begin{align*}
3 \\
-2
\end{align*} \]

Note: \( R(x,x,x,x) = x \), so there exists a region of extreme values
for which the function value is not representable.

4. References

Functions. Dover Publications.


5. Parameters

1: \( X \) -- DOUBLE PRECISION Input
2: \( Y \) -- DOUBLE PRECISION Input
3: \( Z \) -- DOUBLE PRECISION Input
4: \( R \) -- DOUBLE PRECISION Input
On entry: the arguments \( x, y, z \) and (rho) of the function.
Constraint: \( X, Y, Z \geq 0.0, R \neq 0.0 \) and at most one of \( X, Y \) and \( Z \) may be zero.

5: \( IFAIL \) -- INTEGER Input/Output
On entry: \( IFAIL \) must be set to 0, -1 or 1. For users not familiar with this parameter (described in the Essential
Introduction) the recommended value is 0.
On exit: \( IFAIL = 0 \) unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

Errors detected by the routine:

If on entry \( IFAIL = 0 \) or -1, explanatory error messages are
output on the current error message unit (as defined by X04AAF).
IFAIL = 1
On entry at least one of X, Y and Z is negative, or at least
two of them are zero; the function is undefined.

IFAIL = 2
On entry R = 0.0; the function is undefined.

IFAIL = 3
On entry either R is too close to zero, or any two of X, Y
and Z are too close to zero; there is a danger of setting
overflow.

IFAIL = 4
On entry at least one of X, Y, Z and R is too large; there
is a danger of setting underflow.

IFAIL = 5
An error has occurred in a call to S21BAF. Any such
occurrence should be reported to NAG.

On soft failure, the routine returns zero.

7. Accuracy

In principle the routine is capable of producing full machine
precision. However round-off errors in internal arithmetic will
result in slight loss of accuracy. This loss should never be
excessive as the algorithm does not involve any significant
amplification of round-off error. It is reasonable to assume that
the result is accurate to within a small multiple of the machine
precision.

8. Further Comments

Users should consult the Chapter Introduction which shows the
relationship of this function to the classical definitions of the
elliptic integrals.

If the argument R is equal to any of the other arguments, the
function reduces to the integral R , computed by S21BCF.

9. Example

This example program simply generates a small set of non-extreme
arguments which are used with the routine to produce the table of
low accuracy results.

The example program is not reproduced here. The source code for
all example programs is distributed with the NAG Foundation Library software and should be available on-line.

NagSpecialFunctionsPackage (NAGS)

Exports:
s01eaf s13aaf s13acf s13adf s14aaf
s14abf s14baf s15adf s15aef s17acf
s17adf s17aef s17aff s17agf s17ahf
s17ajf s17akf s17def s17def s17dgf
s17dhf s17dlf s18acf s18adf s18aef
s18aff s18dcf s18def s19aaf s19abf
s19acf s19adf s20acf s20adf s21baf
s21bbf s21bcf s21bdf

— package NAGS NagSpecialFunctionsPackage —

)abbrev package NAGS NagSpecialFunctionsPackage
++ Author: Godfrey Nolan and Mike Dewar
++ Date Created: Jan 1994
++ Date Last Updated: Thu May 12 17:45:44 1994
++ Description:
++ This package uses the NAG Library to compute some commonly occurring physical and mathematical functions.

NagSpecialFunctionsPackage(): Exports == Implementation where
S ==> Symbol
FOP ==> FortranOutputStackPackage

Exports == with
  s01eaf : (Complex DoubleFloat,Integer) -> Result
++ s01eaf(x,ifail)
++ S01EAF evaluates the exponential function exp(z), for complex z.
++ See \downlink{Manual Page}{manpageXXs01eaf}.
s13aaf : (DoubleFloat,Integer) -> Result
++ s13aaf(x,ifail)
++ returns the value of the exponential integral
++ E(x), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs13aaf}.
s13acf : (DoubleFloat,Integer) -> Result
++ s13acf(x,ifail)
++ returns the value of the cosine integral
++ See \downlink{Manual Page}{manpageXXs13acf}.
s13adf : (DoubleFloat,Integer) -> Result
++ s13adf(x,ifail)
++ returns the value of the sine integral
++ See \downlink{Manual Page}{manpageXXs13adf}.
s14aaf : (DoubleFloat,Integer) -> Result
++ s14aaf(x,ifail) returns the value of the Gamma function Gamma(x),
++ via the routine name.
++ See \downlink{Manual Page}{manpageXXs14aaf}.
s14abf : (DoubleFloat,Integer) -> Result
++ s14abf(x,ifail) returns a value for the log, \ln(Gamma(x)), via
++ the routine name.
++ See \downlink{Manual Page}{manpageXXs14abf}.
s14abf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s14abf(a,x,tol,ifail)
++ computes values for the incomplete gamma functions P(a,x)
++ and Q(a,x).
++ See \downlink{Manual Page}{manpageXXs14abf}.
s15adf : (DoubleFloat,Integer) -> Result
++ s15adf(x,ifail)
++ returns the value of the complementary error function,
++ erfc(x), via the routine name.
++ See \downlink{Manual Page}{manpageXXs15adf}.
s15aef : (DoubleFloat,Integer) -> Result
++ s15aef(x,ifail)
++ returns the value of the error function erf(x), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs15aef}.
s17acf : (DoubleFloat,Integer) -> Result
++ s17acf(x,ifail)
++ returns the value of the Bessel Function
++ Y(x), via the routine name.
++ 0
++ See \downlink{Manual Page}{manpageXXs17acf}.
s17adf : (DoubleFloat,Integer) -> Result
++ s17adf(x,ifail)
++ returns the value of the Bessel Function
++ Y(x), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs17adf}.
s17aef : (DoubleFloat,Integer) -> Result
++ s17aef(x,ifail)
++ returns the value of the Bessel Function
++ J \(x\), via the routine name.
++ 0
++ See \downlink{Manual Page}{manpageXXs17aef}.
s17aff : (DoubleFloat,Integer) -> Result
++ s17aff(x,ifail)
++ returns the value of the Bessel Function
++ J \(x\), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs17aff}.
s17agf : (DoubleFloat,Integer) -> Result
++ s17agf(x,ifail)
++ returns a value for the Airy function, Ai\(x\), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs17agf}.
s17ahf : (DoubleFloat,Integer) -> Result
++ s17ahf(x,ifail)
++ returns a value of the Airy function, Bi\(x\), via the
++ routine name.
++ See \downlink{Manual Page}{manpageXXs17ahf}.
s17ajf : (DoubleFloat,Integer) -> Result
++ s17ajf(x,ifail)
++ returns a value of the derivative of the Airy function
++ Ai\(x\), via the routine name.
++ See \downlink{Manual Page}{manpageXXs17ajf}.
s17akf : (DoubleFloat,Integer) -> Result
++ s17akf(x,ifail)
++ returns a value for the derivative of the Airy function
++ Bi\(x\), via the routine name.
++ See \downlink{Manual Page}{manpageXXs17akf}.
s17dcf : (DoubleFloat,Complex DoubleFloat,Integer,String,_
    Integer) -> Result
++ s17dcf(fnu,z,n,scale,ifail)
++ returns a sequence of values for the Bessel functions
++ Y \(z\) for complex \(z\), non-negative \(nu\) and \(n=0,1,...,N-1\),
++ \(nu+n\)
++ with an option for exponential scaling.
++ See \downlink{Manual Page}{manpageXXs17dcf}.
s17def : (DoubleFloat,Complex DoubleFloat,Integer,String,_
    Integer) -> Result
++ s17def(fnu,z,n,scale,ifail)
++ returns a sequence of values for the Bessel functions
++ J \(z\) for complex \(z\), non-negative \(nu\) and \(n=0,1,...,N-1\),
++ \(nu+n\)
++ with an option for exponential scaling.
++ See \downlink{Manual Page}{manpageXXs17def}.
s17dgf : (String,Complex DoubleFloat,String,Integer) -> Result
++ s17dgf(deriv,z,scale,ifail)
++ returns the value of the Airy function Ai(z) or its
++ derivative Ai'(z) for complex z, with an option for exponential
++ scaling.
++ See \downlink{Manual Page}{manpageXXs17dgf}.
s17dhf : (String,Complex DoubleFloat,String,Integer) -> Result
++ s17dhf(deriv,z,scale,ifail)
++ returns the value of the Airy function Bi(z) or its
++ derivative Bi'(z) for complex z, with an option for exponential
++ scaling.
++ See \downlink{Manual Page}{manpageXXs17dhf}.
s17dlf : (Integer,DoubleFloat,Complex DoubleFloat,Integer,_
String,Integer) -> Result
++ s17dlf(m,fnu,z,n,scale,ifail)
++ returns a sequence of values for the Hankel functions
++ (1) (2)
++ H (z) or H (z) for complex z, non-negative (nu) and
++ (nu)+n (nu)+n
++ n=0,1,...,N-1, with an option for exponential scaling.
++ See \downlink{Manual Page}{manpageXXs17dlf}.
s18acf : (DoubleFloat,Integer) -> Result
++ s18acf(x,ifail)
++ returns the value of the modified Bessel Function
++ K (x), via the routine name.
++ 0
++ See \downlink{Manual Page}{manpageXXs18acf}.
s18adf : (DoubleFloat,Integer) -> Result
++ s18adf(x,ifail)
++ returns the value of the modified Bessel Function
++ K (x), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs18adf}.
s18aef : (DoubleFloat,Integer) -> Result
++ s18aef(x,ifail)
++ returns the value of the modified Bessel Function
++ I (x), via the routine name.
++ 0
++ See \downlink{Manual Page}{manpageXXs18aef}.
s18aff : (DoubleFloat,Integer) -> Result
++ s18aff(x,ifail)
++ returns a value for the modified Bessel Function
++ I (x), via the routine name.
++ 1
++ See \downlink{Manual Page}{manpageXXs18aff}.
s18dcf : (DoubleFloat,Complex DoubleFloat,Integer,String,_
Integer) -> Result
++ s18dcf(fnu,z,n,scale,ifail)
++ returns a sequence of values for the modified Bessel functions
++ K (z) for complex z, non-negative (nu) and
++ (nu)+n
++ n=0,1,...,N-1, with an option for exponential scaling.
++ See `downlink{Manual Page}{manpageXXs18dcf}.

s18def : (DoubleFloat,Complex DoubleFloat,Integer,String,_
  Integer) -> Result
++ s18def(fnu,z,n,scale,ifail)
++ returns a sequence of values for the modified Bessel functions
++ I (z) for complex z, non-negative (nu) and
++ (nu)+n
++ n=0,1,...,N-1, with an option for exponential scaling.
++ See `downlink{Manual Page}{manpageXXs18def}.
s19aaf : (DoubleFloat,Integer) -> Result
++ s19aaf(x,ifail)
++ returns a value for the Kelvin function ber(x) via the
++ routine name.

++ See `downlink{Manual Page}{manpageXXs19aaf}.
s19abf : (DoubleFloat,Integer) -> Result
++ s19abf(x,ifail)
++ returns a value for the Kelvin function bei(x) via the
++ routine name.

++ See `downlink{Manual Page}{manpageXXs19abf}.
s19acf : (DoubleFloat,Integer) -> Result
++ s19acf(x,ifail)
++ returns a value for the Kelvin function ker(x), via the
++ routine name.

++ See `downlink{Manual Page}{manpageXXs19acf}.
s19adf : (DoubleFloat,Integer) -> Result
++ s19adf(x,ifail)
++ returns a value for the Kelvin function kei(x) via the
++ routine name.

++ See `downlink{Manual Page}{manpageXXs19adf}.
s20acf : (DoubleFloat,Integer) -> Result
++ s20acf(x,ifail)
++ returns a value for the Fresnel Integral S(x), via the
++ routine name.

++ See `downlink{Manual Page}{manpageXXs20acf}.
s20adf : (DoubleFloat,Integer) -> Result
++ s20adf(x,ifail)
++ returns a value for the Fresnel Integral C(x), via the
++ routine name.

++ See `downlink{Manual Page}{manpageXXs20adf}.
s21baf : (DoubleFloat,DoubleFloat,Integer) -> Result
++ s21baf(x,y,ifail)
++ returns a value of an elementary integral, which occurs as
++ a degenerate case of an elliptic integral of the first kind, via
++ the routine name.

++ See `downlink{Manual Page}{manpageXXs21baf}.
s21bbf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s21bbf(x,y,z,ifail)
++ returns a value of the symmetrised elliptic integral of
++ the first kind, via the routine name.
++ See \downlink{Manual Page}{manpageXXs21bbf}.
s21bcf : (DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s21bcf(x,y,z,ifail)
++ returns a value of the symmetrised elliptic integral of
++ the second kind, via the routine name.
++ See \downlink{Manual Page}{manpageXXs21bcf}.
s21bdf : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat,Integer) -> Result
++ s21bdf(x,y,z,r,ifail)
++ returns a value of the symmetrised elliptic integral of
++ the third kind, via the routine name.
++ See \downlink{Manual Page}{manpageXXs21bdf}.

Implementation ==> add

import Lisp
import DoubleFloat
import Any
import Record
import Integer
import Matrix DoubleFloat
import Boolean
import NAGLinkSupportPackage
import AnyFunctions1(Complex DoubleFloat)
import AnyFunctions1(Integer)
import AnyFunctions1(DoubleFloat)
import AnyFunctions1(String)

s01eaf(zArg:Complex DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"s01eaf",_,
["z":S,"ifail"::S]$Lisp_,
[]$Lisp_,
["integer"::S,"ifail"::S]$Lisp_,
,["double complex"::S,"s01eafResult"::S,"z"::S]$Lisp_,
]$Lisp_,
["s01eafResult"::S,"ifail"::S]$Lisp_,
[[[zArg::Any,ifailArg::Any ]]_)
@List Any]$Lisp)$Lisp)
pretend List (Record(key:Symbol,entry:Any))$Result

s13aaf(xArg:DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_,
"s13aaf",_,
["x":S,"ifail"::S]$Lisp_,
[]$Lisp_,
["double"::S,"s13aafResult"::S,"x"::S]$Lisp_,
,["integer"::S,"ifail"::S]$Lisp_,
]$Lisp_,
["s13aafResult"::S,"ifail"::S]$Lisp_,
CHAPTER 15. CHAPTER N

s13acf(xArg:DoubleFloat, ifailArg:Integer): Result ==

s13adf(xArg:DoubleFloat, ifailArg:Integer): Result ==

s14aaf(xArg:DoubleFloat, ifailArg:Integer): Result ==

s14abf(xArg:DoubleFloat, ifailArg:Integer): Result ==
s14baf (a: DoubleFloat, x: DoubleFloat, tol: DoubleFloat, ifail: Integer): Result ==
    [(invokeNagman(NIL$Lisp, 
        "s14baf", 
            "ifail"::S]$Lisp_, 
        ["p"::S,"q"::S]$Lisp_, 
            "q"::S]$Lisp_, 
        ,["integer"::S,"ifail"::S]$Lisp_ 
    ]$Lisp_, 
    ["p"::S,"q"::S,"ifail"::S]$Lisp_, 
    [[a::Any,x::Any,tol::Any,ifail::Any ]_ 
    @$List Any$Lisp)$Lisp_)_ 
    pretend List (Record(key:Symbol,entry:Any))$Result

s15adf (x: DoubleFloat, ifail: Integer): Result ==
    [(invokeNagman(NIL$Lisp, 
        "s15adf", 
        ["x"::S,"ifail"::S]$Lisp_, 
        []$Lisp_, 
        ["double"::S,"s15adfResult"::S,"x"::S]$Lisp_, 
        ,["integer"::S,"ifail"::S]$Lisp_ ])$Lisp_, 
    ["s15adfResult"::S,"ifail"::S]$Lisp_, 
    [[x::Any,ifail::Any ]_ 
    @$List Any$Lisp)$Lisp_)_ 
    pretend List (Record(key:Symbol,entry:Any))$Result

s15aef (x: DoubleFloat, ifail: Integer): Result ==
    [(invokeNagman(NIL$Lisp, 
        "s15aef", 
        ["x"::S,"ifail"::S]$Lisp_, 
        []$Lisp_, 
        ["double"::S,"s15aefResult"::S,"x"::S]$Lisp_, 
        ,["integer"::S,"ifail"::S]$Lisp_ ])$Lisp_, 
    ["s15aefResult"::S,"ifail"::S]$Lisp_, 
    [[x::Any,ifail::Any ]_ 
    @$List Any$Lisp)$Lisp_)_ 
    pretend List (Record(key:Symbol,entry:Any))$Result

s17acf (x: DoubleFloat, ifail: Integer): Result ==
    [(invokeNagman(NIL$Lisp, 
        ["s17acf", 
        ["x"::S,"ifail"::S]$Lisp_, 
        []$Lisp_, 
        ["double"::S,"s17acfResult"::S,"x"::S]$Lisp_, 
        ,["integer"::S,"ifail"::S]$Lisp_ ])$Lisp_, 
    ["s17acfResult"::S,"ifail"::S]$Lisp_, 
    [[x::Any,ifail::Any ]_ 
    @$List Any$Lisp)$Lisp_)_ 
    pretend List (Record(key:Symbol,entry:Any))$Result
"s17acf", 
[["x"::S,"ifail"::S]$Lisp, 
[]$Lisp, 
[["double"::S,"s17acfResult"::S,"x"::S]$Lisp_, 
[["integer"::S,"ifail"::S]$Lisp_ 
]$Lisp_, 
["s17acfResult"::S,"ifail"::S]$Lisp_, 
[[xArg::Any,ifailArg::Any ]]_ 
@List Any]@List Any]$Lisp)_) 
pretend List (Record(key:Symbol,entry:Any))]$Result 

s17adfl(xArg::DoubleFloat,ifailArg::Integer): Result == 
[(invokeNagman(NIL$Lisp, 
"s17adfl", 
["x"::S,"ifail"::S]$Lisp, 
[]$Lisp, 
[["double"::S,"s17adflResult"::S,"x"::S]$Lisp_, 
[["integer"::S,"ifail"::S]$Lisp_ 
]$Lisp_, 
["s17adflResult"::S,"ifail"::S]$Lisp_, 
[[xArg::Any,ifailArg::Any ]]_ 
@List Any]@List Any)$Lisp)_) 
pretend List (Record(key:Symbol,entry:Any))]$Result 

s17aef(xArg::DoubleFloat,ifailArg::Integer): Result == 
[(invokeNagman(NIL$Lisp, 
"s17aef", 
["x"::S,"ifail"::S]$Lisp, 
[]$Lisp, 
[["double"::S,"s17aefResult"::S,"x"::S]$Lisp_, 
[["integer"::S,"ifail"::S]$Lisp_ 
]$Lisp_, 
["s17aefResult"::S,"ifail"::S]$Lisp_, 
[[xArg::Any,ifailArg::Any ]]_ 
@List Any]@List Any)$Lisp)_) 
pretend List (Record(key:Symbol,entry:Any))]$Result 

s17affl(xArg::DoubleFloat,ifailArg::Integer): Result == 
[(invokeNagman(NIL$Lisp, 
"s17aff", 
["x"::S,"ifail"::S]$Lisp, 
[]$Lisp, 
[["double"::S,"s17affResult"::S,"x"::S]$Lisp_, 
[["integer"::S,"ifail"::S]$Lisp_ 
]$Lisp_, 
["s17affResult"::S,"ifail"::S]$Lisp_, 
[[xArg::Any,ifailArg::Any ]]_ 
@List Any]@List Any)$Lisp)_) 
pretend List (Record(key:Symbol,entry:Any))]$Result
s17agf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, \\
    "s17agf", \\
    ["x":S,"ifail":S]$Lisp, \\
    []$Lisp, \\
    ["double":S,"s17agfResult":S,"x":S]$Lisp_, \\
    ["integer":S,"ifail":S]$Lisp_ \\
    ]$Lisp_, \\
    ["s17agfResult":S,"ifail":S]$Lisp_, \\
    [([xArg::Any,ifailArg::Any ])_ \\
     @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s17ahf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, \\
    "s17ahf", \\
    ["x":S,"ifail":S]$Lisp, \\
    []$Lisp, \\
    ["double":S,"s17ahfResult":S,"x":S]$Lisp_, \\
    ["integer":S,"ifail":S]$Lisp_ \\
    ]$Lisp_, \\
    ["s17ahfResult":S,"ifail":S]$Lisp_, \\
    [([xArg::Any,ifailArg::Any ])_ \\
     @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s17ajf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, \\
    "s17ajf", \\
    ["x":S,"ifail":S]$Lisp, \\
    []$Lisp, \\
    ["double":S,"s17ajfResult":S,"x":S]$Lisp_, \\
    ["integer":S,"ifail":S]$Lisp_ \\
    ]$Lisp_, \\
    ["s17ajfResult":S,"ifail":S]$Lisp_, \\
    [([xArg::Any,ifailArg::Any ])_ \\
     @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s17akf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, \\
    "s17akf", \\
    ["x":S,"ifail":S]$Lisp, \\
    []$Lisp, \\
    ["double":S,"s17akfResult":S,"x":S]$Lisp_, \\
    ["integer":S,"ifail":S]$Lisp_ \\
    ]$Lisp_, \\
    ["s17akfResult":S,"ifail":S]$Lisp_, \\
    [([xArg::Any,ifailArg::Any ])_ \\
     @List Any]$Lisp)$Lisp)
[(invokeNagman(NIL$Lisp, 
  "s17dcf", 
  "ifail":S,"cy":S,"cwrk":S]$Lisp,_, 
  ["cy":S,"nz":S,"cwrk":S]$Lisp,_, 
  ["double":S,"fnu":S]$Lisp_, 
  ["character":S,"scale":S]$Lisp_, 
  ["double complex":S,"z":S,"cy":S,"n":S]$Lisp,_, 
  ["cwrk":S,"n":S]$Lisp]$Lisp$Lisp_, 
["cy":S,"nz":S,"ifail":S]$Lisp_, 
[[[fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ]]_]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

[(invokeNagman(NIL$Lisp, 
  "s17def", 
  "ifail":S,"cy":S]$Lisp,_, 
  ["cy":S,"nz":S]$Lisp,_, 
  ["double":S,"fnu":S]$Lisp_, 
  ["character":S,"scale":S]$Lisp_, 
["cy":S,"nz":S,"ifail":S]$Lisp_, 
[[[fnuArg::Any,zArg::Any,nArg::Any, scaleArg::Any,ifailArg::Any ]]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

s17dgf(derivArg:String,zArg:Complex DoubleFloat,scaleArg:String, ifailArg:Integer): Result == 
[(invokeNagman(NIL$Lisp, 
  "s17dgf", 
  "ifail":S]$Lisp,_, 
  ["ai":S,"nz":S]$Lisp,_, 
  ["integer":S,"nz":S,"ifail":S]$Lisp_, 
  ["character":S,"deriv":S,"scale":S]$Lisp_, 
  ["double complex":S,"z":S,"ai":S]$Lisp]$Lisp_, 
[a1":S,"nz":S]$Lisp,_, 
[[[fnuArg::Any,zArg::Any,nArg::Any, scaleArg::Any,ifailArg::Any ]]_ 
@List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result
s17dhf(derivArg::String,zArg::Complex DoubleFloat,scaleArg::String,ifailArg::Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s17dhf",_
]$Lisp,_
["bi":S]$Lisp,_
["integer":S,"ifail":S]$Lisp_
,[],"character":S,"deriv":S,"scale":S]$Lisp_
,[],"double complex":S,"z":S,"bi":S]$Lisp_
]$Lisp,_
["bi":S,"ifail":S]$Lisp,_
[(derivArg::Any,zArg::Any,scaleArg::Any,ifailArg::Any ])
@List Any$Lisp$Lisp)_,
pretend List (Record(key:Symbol,entry:Any))$Result

s17dlf(mArg:Integer,fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer,scaleArg::String,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s17dlf",_
["m":S,"fnu":S,"z":S,"n":S,"scale":S_
["cy":S,"nz":S]$Lisp,_
["integer":S,"m":S,"n":S,"nz":S,"ifail":S_
]$Lisp,_
,[],"character":S,"scale":S]$Lisp_
]$Lisp,_
["cy":S,"nz":S,"ifail":S]$Lisp,_
[(mArg::Any,fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,_
ifailArg::Any ])
@List Any$Lisp$Lisp)_,
pretend List (Record(key:Symbol,entry:Any))$Result

s18acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
[(invokeNagman(NIL$Lisp,_
"s18acf",_
["x":S,"ifail":S]$Lisp,_
[])$Lisp,_
["double":S,"s18acfResult":S,"x":S]$Lisp_
,[],"integer":S,"ifail":S]$Lisp_
]$Lisp,_
["s18acfResult":S,"ifail":S]$Lisp,_
[(xArg::Any,ifailArg::Any ])
@List Any$Lisp$Lisp)_,
pretend List (Record(key:Symbol,entry:Any))$Result

s18adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s18adf", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s18adfResult"::S,"x"::S]$Lisp_,
      ["integer"::S,"ifail"::S]$Lisp_,
      ]$Lisp_, _
    ["s18adfResult"::S,"ifail"::S]$Lisp_, _
    [(xArg::Any,ifailArg::Any )]$Lisp]]$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s18aef(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s18aef", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s18aefResult"::S,"x"::S]$Lisp_,
      ["integer"::S,"ifail"::S]$Lisp_,
      ]$Lisp_, _
    ["s18aefResult"::S,"ifail"::S]$Lisp_, _
    [(xArg::Any,ifailArg::Any )]$Lisp]]$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s18aff(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s18aff", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s18affResult"::S,"x"::S]$Lisp_,
      ["integer"::S,"ifail"::S]$Lisp_,
      ]$Lisp_, _
    ["s18affResult"::S,"ifail"::S]$Lisp_, _
    [(xArg::Any,ifailArg::Any )]$Lisp]]$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s18dcf(fnuArg:DoubleFloat,zArg:Complex DoubleFloat,nArg:Integer, _
  scaleArg:String,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s18dcf", _
      "ifail"::S,"cy"::S]$Lisp_,
      ["cy"::S,"nz"::S]$Lisp_, _
      [["double"::S,"fnu"::S]$Lisp_,
s18def (fnuArg: DoubleFloat, zArg: Complex DoubleFloat, nArg: Integer, _
   scaleArg: String, ifailArg: Integer): Result ==
   [(invokeNagman(NIL$Lisp, _
      "s18def", _
      "ifail"::S,"cy"::S]$Lisp, _
      ["cy"::S,"nz"::S]$Lisp, _
      [["double"::S,"fnu"::S]$Lisp_ _
       ,["character"::S,"scale"::S]$Lisp_ _
       ,["double complex"::S,"z"::S,["cy"::S,"n"::S]$Lisp]$Lisp_ _
       ]$Lisp, _
      ["cy"::S,"nz"::S,"ifail"::S]$Lisp, _
      [[fnuArg::Any,zArg::Any,nArg::Any,scaleArg::Any,ifailArg::Any ])_ _
     @List Any]$Lisp)$Lisp) _
   pretend List (Record(key:Symbol,entry:Any))$Result

s19aaf (xArg: DoubleFloat, ifailArg: Integer): Result ==
   [(invokeNagman(NIL$Lisp, _
      "s19aaf", _
      ["x"::S,"ifail"::S]$Lisp, _
      []$Lisp, _
      [["double"::S,"s19aafResult"::S,"x"::S]$Lisp_ _
       ,["integer"::S,"ifail"::S]$Lisp_ _
       [$Lisp_ _
       ["s19aafResult"::S,"ifail"::S]$Lisp, _
       [[xArg::Any,ifailArg::Any ])_ _
     @List Any]$Lisp)$Lisp) _
   pretend List (Record(key:Symbol,entry:Any))$Result

s19abf (xArg: DoubleFloat, ifailArg: Integer): Result ==
   [(invokeNagman(NIL$Lisp, _
      "s19abf", _
      ["x"::S,"ifail"::S]$Lisp, _
      []$Lisp, _
      [["double"::S,"s19abfResult"::S,"x"::S]$Lisp_ _
       ,["integer"::S,"ifail"::S]$Lisp_ _
       [$Lisp_ _
       ["s19abfResult"::S,"ifail"::S]$Lisp, _
       [[xArg::Any,ifailArg::Any ])_ _
     @List Any]$Lisp)$Lisp) _
   pretend List (Record(key:Symbol,entry:Any))$Result
CHAPTER 15. CHAPTER N

s19acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s19acf", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s19acfResult"::S,"x"::S]$Lisp_, _
      ["integer"::S,"ifail"::S]$Lisp_ _
      ]$Lisp_, _
    ["s19acfResult"::S,"ifail"::S]$Lisp_, _
    [[xArg::Any,ifailArg::Any ]])_ 
  @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s19adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s19adf", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s19adfResult"::S,"x"::S]$Lisp_, _
      ["integer"::S,"ifail"::S]$Lisp_ _
      ]$Lisp_, _
    ["s19adfResult"::S,"ifail"::S]$Lisp_, _
    [[xArg::Any,ifailArg::Any ]])_ 
  @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s20acf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s20acf", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s20acfResult"::S,"x"::S]$Lisp_, _
      ["integer"::S,"ifail"::S]$Lisp_ _
      ]$Lisp_, _
    ["s20acfResult"::S,"ifail"::S]$Lisp_, _
    [[xArg::Any,ifailArg::Any ]])_ 
  @List Any]$Lisp)$Lisp)
  pretend List (Record(key:Symbol,entry:Any))$Result

s20adf(xArg:DoubleFloat,ifailArg:Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s20adf", _
    ["x"::S,"ifail"::S]$Lisp, _
    []$Lisp, _
    [["double"::S,"s20adfResult"::S,"x"::S]$Lisp_, _
      ["integer"::S,"ifail"::S]$Lisp_ _
      ]$Lisp_, _
    ["s20adfResult"::S,"ifail"::S]$Lisp_, _
    [[xArg::Any,ifailArg::Any ]])_
s21baf(xArg: DoubleFloat, yArg: DoubleFloat, ifailArg: Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s21baf", _
    ["x":S,"y":S,"ifail":S]$Lisp, _
    []$Lisp, _
    ["double":S,"s21bafResult":S,"x":S,"y":S_ 
      ]$Lisp, _
    ["integer":S,"ifail":S]$Lisp_ 
      ]$Lisp, _
    ["s21bafResult":S,"ifail":S]$Lisp, _
    [[xArg::Any,yArg::Any,ifailArg::Any ]_ 
      @List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

s21bbf(xArg: DoubleFloat, yArg: DoubleFloat, zArg: DoubleFloat, _
  ifailArg: Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s21bbf", _
    ["x":S,"y":S,"z":S,"ifail":S]$Lisp, _
    []$Lisp, _
    ["double":S,"s21bbfResult":S,"x":S,"y":S_ 
     ,"z":S]$Lisp_ 
    ,["integer":S,"ifail":S]$Lisp_ 
      ]$Lisp, _
    ["s21bbfResult":S,"ifail":S]$Lisp, _
    [[xArg::Any,yArg::Any,zArg::Any,ifailArg::Any ]_ 
      @List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

s21bcf(xArg: DoubleFloat, yArg: DoubleFloat, zArg: DoubleFloat, _
  ifailArg: Integer): Result ==
  [(invokeNagman(NIL$Lisp, _
    "s21bcf", _
    ["x":S,"y":S,"z":S,"ifail":S]$Lisp, _
    []$Lisp, _
    ["double":S,"s21bcfResult":S,"x":S,"y":S_ 
     ,"z":S]$Lisp_ 
    ,["integer":S,"ifail":S]$Lisp_ 
      ]$Lisp, _
    ["s21bcfResult":S,"ifail":S]$Lisp, _
    [[xArg::Any,yArg::Any,zArg::Any,ifailArg::Any ]_ 
      @List Any]$Lisp)$Lisp) 
pretend List (Record(key:Symbol,entry:Any))$Result

s21bdf(xArg: DoubleFloat, yArg: DoubleFloat, zArg: DoubleFloat, _
  rArg: DoubleFloat, ifailArg: Integer): Result ==
  [(invokeNagman(NIL$Lisp, _

"s21bdf",
[]$_Lisp_ _,
[["integer":S,"ifail":S]$_Lisp_ _
[]$_Lisp_ _,
["s21bdfResult":S,"ifail":S]$_Lisp_ _
[[[xArg::Any,yArg::Any,zArg::Any,rArg::Any,ifailArg::Any ])_ _
0List Any]$_Lisp)$_Lisp) _
pretend List (Record(key:Symbol,entry:Any))$_Result

|---

--- NAGS.dotabb ---
"NAGS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NAGS"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"NAGS" -> "ALIST"

|---

package NSUP2 NewSparseUnivariatePolynomialFunctions2

--- NewSparseUnivariatePolynomialFunctions2.input ---

)set break resume
)sys rm -f NewSparseUnivariatePolynomialFunctions2.output
)spool NewSparseUnivariatePolynomialFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NewSparseUnivariatePolynomialFunctions2
--R
--R NewSparseUnivariatePolynomialFunctions2(R: Ring,S: Ring) is a package constructor
--R Abbreviation for NewSparseUnivariatePolynomialFunctions2 is NSUP2
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NSUP2
--R
--R-------------------------- Operations --------------------------
--R map : ((R -> S),NewSparseUnivariatePolynomial(R)) -> NewSparseUnivariatePolynomial(S)
This package lifts a mapping from coefficient rings \( R \) to \( S \) to a mapping from sparse univariate polynomial over \( R \) to a sparse univariate polynomial over \( S \). Note that the mapping is assumed to send zero to zero, since it will only be applied to the non-zero coefficients of the polynomial.

See Also:
\- )show NewSparseUnivariatePolynomialFunctions2

---

NewSparseUnivariatePolynomialFunctions2 (NSUP2)

Exports:
map

---

)abbrev package NSUP2 NewSparseUnivariatePolynomialFunctions2
++ Description:
++ This package lifts a mapping from coefficient rings R to S to
++ a mapping from sparse univariate polynomial over R to
++ a sparse univariate polynomial over S.
++ Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

NewSparseUnivariatePolynomialFunctions2(R:Ring, S:Ring): with
  map:(R->S,NewSparseUnivariatePolynomial R) -> NewSparseUnivariatePolynomial S
++ \aat{map(func, poly)} creates a new polynomial by applying func to
++ every non-zero coefficient of the polynomial poly.
== add
  map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
  NewSparseUnivariatePolynomial R, S, NewSparseUnivariatePolynomial S)

— NSUP2.dotabb —

"NSUP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NSUP2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"NSUP2" -> "LMODULE"
"NSUP2" -> "SGROUP"

package NEWTON NewtonInterpolation

— NewtonInterpolation.input —

)set break resume
)sys rm -f NewtonInterpolation.output
)spool NewtonInterpolation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NewtonInterpolation
--R
--R NewtonInterpolation(F: IntegralDomain) is a package constructor
--R Abbreviation for NewtonInterpolation is NEWTON
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NEWTON
--R
--R----------------------------------- Operations -----------------------------------
--R newton : List(F) -> SparseUnivariatePolynomial(F)
--R

)spool
)lisp (bye)

NewtonInterpolation (NEWTON)

Exports:
newton

NewtonInterpolation examples

This package exports Newton interpolation for the special case where the result is known to be in the original integral domain. The packages defined in this file provide fast fraction free rational interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

See Also:
 o )show NewtonInterpolation
package NEWTON NewtonInterpolation

)abbrev package NEWTON NewtonInterpolation
++ Description:
++ This package exports Newton interpolation for the special case where the
++ result is known to be in the original integral domain
++ The packages defined in this file provide fast fraction free rational
++ interpolation algorithms. (see FAMR2, FFFG, FFFGF, NEWTON)

NewtonInterpolation F:Exports == Implementation where
F: IntegralDomain
Exports == with

newton: List F -> SparseUnivariatePolynomial F
++ \spad{newton}(l) returns the interpolating polynomial for the values
++ l, where the x-coordinates are assumed to be [1,2,3,...,n] and the
++ coefficients of the interpolating polynomial are known to be in the
++ domain F. I.e., it is a very streamlined version for a special case of
++ interpolation.

Implementation == add

 differences(yl: List F): List F ==
 [y2-y1 for y1 in yl for y2 in rest yl]
z: SparseUnivariatePolynomial(F) := monomial(1,1)
-- we assume x=[1,2,3,...,n]
newtonAux(k: F, fact: F, yl: List F): SparseUnivariatePolynomial(F) ==
  if empty? rest yl
    then (((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
  else (((yl.1) exquo fact)::F::SparseUnivariatePolynomial(F)
            + (z-k::SparseUnivariatePolynomial(F)) _
                * newtonAux(k+1$F, fact*k, differences yl)

newton yl == newtonAux(1$F, 1$F, yl)
package NPOLYGON NewtonPolygon

--- NewtonPolygon.input ---

)set break resume
)sys rm -f NewtonPolygon.output
)spool NewtonPolygon.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NewtonPolygon

--R NewtonPolygon(K: Ring,PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategory(dim,NonNegativeInteger)) is a package constructor

--R Abbreviation for NewtonPolygon is NPOLYGON

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NPOLYGON

--R

--- NewtonPolygon.help ---

====================================================================
NewtonPolygon examples
====================================================================

The following is part of the PAFF package

See Also:
  o )show NewtonPolygon
NewtonPolygon (NPOLYGON)

Exports:
  multiplicity  negAndPosEdge  newtonPolygon  slope

--- package NPOLYGON NewtonPolygon ---

)abbrev package NPOLYGON NewtonPolygon
++ Author: Gaetan Hache
++ Date Created: 17 Nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
NewtonPolygon(K,PolyRing,E,dim):Exports == Implementation where
  K:Ring
  dim:NonNegativeInteger
  PolyRing: FiniteAbelianMonoidRing(K,E)
  E: DirectProductCategory(dim,NonNegativeInteger)

PackPoly >>= PackageForPoly(K,PolyRing,E,dim)

recSlope >>= Record( height:Integer, base:Integer, quotient:Integer, _
  reste:Integer, _
  type:Union("left","center","right","vertical","horizontal") )

Exports >>= with

  newtonPolygon: (PolyRing,Integer,Integer,Union("left","center", "right",_
  "vertical","horizontal")) -> List List PolyRing

  multiplicity: List List PolyRing -> NonNegativeInteger
PACKAGE NPOLYGON NEUTONPOLYGON

negAndPosEdge: (PolyRing, List List PolyRing) -> List List PolyRing
slope: (PolyRing, PolyRing) -> recSlope
slope: List PolyRing -> recSlope

Implementation ==> add

tslope(p1,p2)==
  -- calcule la pente de p1 a p2 et change le signe.
  e1:=degree p1
e2:=degree p2
  hgt:= ( e1.2 pretend Integer) - ( e2.2 pretend Integer)
bh:= ( e2.1 pretend Integer) - ( e1.1 pretend Integer )
zero? bh => [hgt, bh, 0$Integer, 0$Integer, "vertical"]$recSlope
zero? hgt => [hgt, bh, 0$Integer, 0$Integer, "horizontal"]$recSlope
hgt = bh => [hgt, bh, 1$Integer, 0$Integer, "center"]$recSlope
hgt > bh =>
  eucl:=divide(hgt,bh)
  [hgt, bh, eucl.quotient, eucl.remainder, "left"]$recSlope
  eucl:=divide(bh, hgt)
  [hgt, bh, eucl.quotient, eucl.remainder, "right"]$recSlope

oneToPos: List List PolyRing -> List List PolyRing

oneToPos(lpol)==
  fedge:= first lpol
  sl:= slope fedge
  one? ( #(lpol) ) =>
    if sl.height > sl.base then [ fedge, empty() ]
    else [ empty(), fedge ]
    ~( sl.base < sl.height ) => [ empty(), fedge ]
  restPANE:= oneToPos rest lpol
  fedge2 := first restPANE
  sl2:= slope fedge2
  ~( sl2.base < sl2.height ) => [ fedge, fedge2 ]
  restPANE

oneToNeg: List List PolyRing -> List List PolyRing

oneToNeg(lpol)==
  fedge:= first lpol
  sl:= slope fedge
  one? ( #(lpol) ) =>
    if sl.height < sl.base then [ empty(), fedge ]
    else [ fedge, empty() ]
    ~( sl.height < sl.base ) => [ empty(), fedge ]
  restPANE:= oneToNeg rest lpol
  fedge2 := first restPANE
  sl2:= slope fedge2
  ~( sl2.height < sl2.base ) => [ fedge, fedge2 ]
restPANE

negAndPosEdge(pol, lpol) ==
  -- cette fonction retourne deux liste de polynomes:
  -- la premiere est liée à
  -- la transformation \( x = x y^l \) (i.e \( v(x) \geq v(y) \)).
  -- la deuxièmeme est liée à la transformation
  -- \( y = x^l y \) (i.e. \( v(x) \leq v(y) \)).
  -- si le degré en \( Y \) est inférieur à celui en \( X \) on
  -- prévilege la transformation
  -- \( y = x^l y \).
  degree( pol , 2 )$PackPoly < degree( pol, 1 )$PackPoly => oneToPos lpol
  oneToNeg lpol

localNewtonPolygon: List PolyRing -> List PolyRing

slEq: (recSlope, recSlope) -> Boolean

regroup: List PolyRing -> List List PolyRing

multiplicity( lpol ) ==
  nl:=#(lpol)
  flpol:= first lpol
  one? nl=> totalDegree( last flpol)$PackPoly
  s:=slope flpol
  s.height < s.base => totalDegree( first flpol )$PackPoly
  multiplicity( rest lpol )

slEq(s1,s2) ==
  s1.height * s2.base = s2.height * s1.base

regroup(lpol) ==
  -- Note : les elements de lpol sont sur la frontière d'un poly.
  -- de Newton et il sont deja trie's.
  nl:=#(lpol)
  one? nl => [lpol]
  2 = nl => [lpol]
  f:=first lpol
  r:= regroup rest lpol
  -- Note : les listes de "r" contiennent au moins 2 elements !!
  fg:=first r
  s1:=slope(f, first fg)
  s2:=slope(fg.1,fg.2)
  slEq(s1,s2) => cons( cons(f, fg) , rest r)
  cons( [f, first fg], r)

-- =====================================================================
-- sortMono : trie les monomes par ordre croissant
-- =====================================================================
sortMono: (PolyRing, PolyRing) -> Boolean
sortMono(p1,p2) ==
a := degree p1
b := degree p2
a.1 < b.1 => true -- p1 est à gauche de p2
a.1 = b.1 and a.2 > b.2 => true -- p1 est au dessus de p2
false

-- newtonPolygon : retourne tous les monomes sur la frontière du polygone de Newton, regroupés selon leur pente.
-- ==============================================================

properSlope: ( List PolyRing, Integer, Integer, _ Union("left","center","right","vertical","horizontal")) -> Boolean
properSlope(lpol,hgt,bs,tp)==
s:=slope lpol
tp case "left" and s.height = hgt and s.base = bs => true
tp case "right" and s.height = bs and s.base = hgt => true
false

newtonPolygon(pol,hgt,bs,tp)==
ans:=regroup localNewtonPolygon
    sort( sortMono(#1,#2) , monomials(pol)$PackPoly)
zero?(bs) => ans
[ 1 for l in ans | properSlope(l,hgt,bs,tp)]

comp2pol: (PolyRing,PolyRing) -> List PolyRing
comp2pol(p1,p2)==
rs:= slope(p1,p2)
zero? rs.base => -- p1 et p2 sont alignes verticalement !!
    zero? rs.height => [p1 + p2] -- les monomes sont identiques !
    rs.height < 0 => [p1] -- p2 est au dessus de p1,
                     -- il faut retourner p1 !!
    rs.height > 0 => [p2] -- sinon p1 est au dessus de p2 .
    rs.base > 0 => [p1] -- p1 est a gauche de p2
    rs.base < 0 => [p2,p1] -- p2 est plus haut que p1
    rs.height < 0 => [p2,p1] -- p2 est plus haut que p1
    rs.height > 0 => [p1] -- p1 est a la meme hauteur que p2

slope(lpol) ==
"one?(#1pol) => slope( first lpol, second lpol)
f:= first lpol
( degree(f,2)$PackPoly < degree(f,1)$PackPoly ) => _
  [ 0$Integer, 1$Integer,0,0, "right"]$recSlope
convex_?(p1,p2,p3)==
  s1:=slope(p1,p2)
  s2:=slope(p2,p3)
  s1.type case "horizontal" => true
  s2.type case "vertical" => true
  s1.type case "vertical" => false -- car ici il faut c2 vertical
  s2.type case "horizontal" => false
  (s1.height * s2.base) < (s2.height * s1.base)

 localNewtonPolygon(lpol)==
  -- lpol doit etre trie' par sortMono
  empty? lpol => empty()
  nl:= #(lpol)
  one? nl => lpol
  lt:=first lpol
  polgRest:= localNewtonPolygon rest lpol
  consBondary( lt , polgRest )

package NCODIV NonCommutativeOperatorDivision
This package provides a division and related operations for MonogenicLinearOperator over a Field. Since the multiplication is in general non-commutative, these operations all have left- and right-hand versions. This package provides the operations based on left-division.

\[ [q,r] = \text{leftDivide}(a,b) \text{ means } a=b*q+r \]

See Also:
  o )show NonCommutativeOperatorDivision
NonCommutativeOperatorDivision (NCODIV)

Exports:
leftGcd leftLcm leftQuotient leftRemainder leftDivide leftExactQuotient

— package NCODIV NonCommutativeOperatorDivision —

)abbrev package NCODIV NonCommutativeOperatorDivision
++ Author: Jean Della Dora, Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This package provides a division and related operations for
++ \spad{MonogenicLinearOperator}s over a \spad{Field}.
++ Since the multiplication is in general non-commutative,
++ these operations all have left- and right-hand versions.
++ This package provides the operations based on left-division.
++ \tab{5}\[q,r\] = leftDivide(a,b) means \(a=b*q+r\)

NonCommutativeOperatorDivision(P, F): PDcat == PDdef where
P: MonogenicLinearOperator(F)
F: Field

PDcat == with

leftDivide: (P, P) -> Record(quotient: P, remainder: P)
++ leftDivide(a,b) returns the pair \spad{\{q,r\}} such that
++ \spad{a = b*q + r} and the degree of \spad{r} is
++ less than the degree of \spad{b}.
++ This process is called ‘‘left division’’.

leftQuotient: (P, P) -> P
++ leftQuotient(a,b) computes the pair \spad{\{q,r\}} such that
++ \spad{a = b*q + r} and the degree of \spad{r} is
++ less than the degree of \spad{b}.
++ The value \spad{q} is returned.

leftRemainder: (P, P) -> P
++ leftRemainder(a,b) computes the pair \spad{\{q,r\}} such that
++ \spad{a = b*q + r} and the degree of \spad{r} is
++ less than the degree of \spad{b}.
++ The value \spad{r} is returned.
leftExactQuotient:(P, P) \to \text{Union}(P, \text{"failed"})
++ leftExactQuotient(a,b) computes the value \spad{q}, if it exists,
++ such that \spad{a = b*q}.

leftGcd: (P, P) \to P
++ leftGcd(a,b) computes the value \spad{g} of highest degree
++ such that
++ \spad{a = aa*g}
++ \spad{b = bb*g}
++ for some values \spad{aa} and \spad{bb}.
++ The value \spad{g} is computed using left-division.

leftLcm: (P, P) \to P
++ leftLcm(a,b) computes the value \spad{m} of lowest degree
++ such that \spad{m = a*aa = b*bb} for some values
++ \spad{aa} and \spad{bb}. The value \spad{m} is
++ computed using left-division.

PDdef == add
leftDivide(a, b) ==
q: P := 0
r: P := a
iv:F := inv leadingCoefficient b
while degree r >= degree b and r ^= 0 repeat
  h := monomial(iv*leadingCoefficient r, 
               (degree r - degree b)::NonNegativeInteger)$P
  r := r - b*h
  q := q + h
[q,r]

-- leftQuotient(a,b) is the quotient from left division, etc.
leftQuotient(a,b) == leftDivide(a,b).quotient
leftRemainder(a,b) == leftDivide(a,b).remainder
leftExactQuotient(a,b) ==
  qr := leftDivide(a,b)
  if qr.remainder = 0 then qr.quotient else "failed"
-- l = leftGcd(a,b) means a = aa*l b = bb*l. Uses leftDivide.
leftGcd(a,b) ==
  a = 0 \Rightarrow b
  b = 0 \Rightarrow a
  while degree b > 0 repeat (a,b) := (b, leftRemainder(a,b))
  if b=0 then a else b
-- l = leftLcm(a,b) means l = a*aa l = b*bb Uses leftDivide.
leftLcm(a,b) ==
  a = 0 \Rightarrow b
  b = 0 \Rightarrow a
  b0 := b
  u := monomial(1,0)$P
  v := 0
while leadingCoefficient b ^= 0 repeat
  qr := leftDivide(a,b)
  (a, b) := (b, qr.remainder)
  (u, v) := (u*qr.quotient+v, u)
  b0*u

— NCODIV.dotabb —

"NCODIV" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCODIV"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"NCODIV" -> "FIELD"

——

package NONE1 NoneFunctions1

—— NoneFunctions1.input ——

)set break resume
)sys rm -f NoneFunctions1.output
)spool NoneFunctions1.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show NoneFunctions1
--R
--R NoneFunctions1(S: Type) is a package constructor
--R Abbreviation for NoneFunctions1 is NONE1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NONE1
--R
--R--------------------------------------------- Operations -----------------------------------
--R coerce : S -> None
--R
--E 1

)spool
)lisp (bye)

——
NoneFunctions1 implements functions on None. It particular it includes a particulary dangerous coercion from any other type to \spadtype{None}.

See Also:
o \)show NoneFunctions1

---

NoneFunctions1 (NONE1)

---

Exports:
coerce

---

\)abbrev package NONE1 NoneFunctions1
++ Description:
++ \spadtype{NoneFunctions1} implements functions on \spadtype{None}.
++ It particular it includes a particulary dangerous coercion from
++ any other type to \spadtype{None}.

NoneFunctions1(S:Type): Exports == Implementation where
Exports ==>
  coerce: S -> None
  ++ coerce(x) changes \spad{x} into an object of type
\texttt{\textbackslash spadtype\{None\}}.

\par
\texttt{Implementation} \texttt{=} \texttt{add}
\texttt{coerce(s:S):None == s pretend None}

\begin{verbatim}
package NODE1 NonLinearFirstOrderODESolver

NonLinearFirstOrderODESolver.input

set break resume
sys rm -f NonLinearFirstOrderODESolver.output
spool NonLinearFirstOrderODESolver.output
set message test on
set message auto off
clear all

--S 1 of 1
show NonLinearFirstOrderODESolver
--R
--R NonLinearFirstOrderODESolver(R: Join(OrderedSet,EuclideanDomain,RetractableTo(Integer),Linear)
--R Abbreviation for NonLinearFirstOrderODESolver is NODE1
--R This constructor is not exposed in this frame.
--R Issue \texttt{)edit bookvol10.4.pamphlet} to see algebra source code for NODE1
--R
--R------------------------------------------ Operations ------------------------------------------
--R solve : (F,F,BasicOperator,Symbol) \rightarrow \text{Union}(F,\text{"failed"})
--R
--E 1

spool
lisp (bye)
\end{verbatim}
NonLinearFirstOrderODESolver provides a function for finding closed form first integrals of nonlinear ordinary differential equations of order 1.

See Also:
- )show NonLinearFirstOrderODESolver

---

NonLinearFirstOrderODESolver (NODE1)

Exports:
solve

---

)abbrev package NODE1 NonLinearFirstOrderODESolver
++ Author: Manuel Bronstein
++ Date Created: 2 September 1991
++ Date Last Updated: 14 October 1994
++ Description:
++ NonLinearFirstOrderODESolver provides a function for finding closed form first integrals of nonlinear ordinary differential equations of order 1.

NonLinearFirstOrderODESolver(R, F):Exports == Implementation where
R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer, LinearlyExplicitRingOver Integer, CharacteristicZero)
F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory, PrimitiveFunctionCategory)

N ==> NonNegativeInteger
Q ==> Fraction Integer
UQ ==> Union(Q, "failed")
OP ==> BasicOperator
SY ==> Symbol
K ==> Kernel F
U ==> Union(F, "failed")
P ==> SparseMultivariatePolynomial(R, K)
REC ==> Record(coef:Q, logand:F)
SOL ==> Record(particular: F, basis: List F)
BER ==> Record(coef1:F, coefn:F, exponent:N)

Exports ==> with
solve: (F, F, OP, SY) -> U
++ solve(M(x,y), N(x,y), y, x) returns \spad{F(x,y)} such that
++ \spad{F(x,y) = c} for a constant \spad{c} is a first integral
++ of the equation \spad{M(x,y) dx + N(x,y) dy = 0}, or
++ "failed" if no first-integral can be found.

Implementation ==> add
import ODEIntegration(R, F)
import ElementaryFunctionODESolver(R, F) -- recursive dependency!

checkBernoulli : (F, F, K) -> Union(BER, "failed")
solveBernoulli : (BER, OP, SY, F) -> Union(F, "failed")
checkRiccati : (F, F, K) -> Union(List F, "failed")
solveRiccati : (List F, OP, SY, F) -> Union(F, "failed")
partSolRiccati : (List F, OP, SY, F) -> Union(F, "failed")
integratingFactor: (F, F, SY, SY) -> U

unk := new()$SY
kunk:K := kernel unk

solve(m, n, y, x) ==
-- first replace the operator y(x) by a new symbol z in m(x,y) and n(x,y)
    lk:List(K) := [retract(yx := y(x::F))@K]
    lv:List(F) := [kunk::F]
    mm := eval(m, lk, lv)
    nn := eval(n, lk, lv)
-- put over a common denominator (to balance m and n)
    d := lcm(denom mm, denom nn)::F
    mm := d * mm
    nn := d * nn
-- look for an integrating factor mu
    (u := integratingFactor(mm, nn, unk, x)) case F =>
mu := u::F
mm := mm * mu
nn := nn * mu
eval(int(mm,x) + int(nn-int(differentiate(mm,unk),x), unk),[kunk],[yx])

-- check for Bernoulli equation
(w := checkBernoulli(m, n, k1 := first lk)) case BER =>
solveBernoulli(w::BER, y, x, yx)

-- check for Riccati equation
(v := checkRiccati(m, n, k1)) case List(F) =>
solveRiccati(v::List(F), y, x, yx)
"failed"

-- look for an integrating factor
integratingFactor(m, n, y, x) ==

-- check first for exactness
zero?(d := differentiate(m, y) - differentiate(n, x)) => 1

-- look for an integrating factor involving x only
not member?(y, variables(f := d / n)) => expint(f, x)

-- look for an integrating factor involving y only
not member?(x, variables(f := - d / m)) => expint(f, y)

-- room for more techniques later on (e.g. Prelle-Singer etc...)
"failed"

-- check whether the equation is of the form
-- dy/dx + p(x)y + q(x)y^N = 0 with N > 1
-- i.e. whether m/n is of the form p(x) y + q(x) y^N
-- returns [p, q, N] if the equation is in that form
checkBernoulli(m, n, ky) ==
r := denom(f := m / n)::F
(not freeOf?(r, y := ky::F))
or (d := degree(p := univariate(numer f, ky))) < 2
or degree(pp := reductum p) ^= 1 or reductum(pp) ^= 0
or (not freeOf?(a := (leadingCoefficient(pp)::F), y))
or (not freeOf?(b := (leadingCoefficient(p)::F), y)) => "failed"
[a / r, b / r, d]

-- solves the equation dy/dx + rec.coef1 y + rec.coefn y^rec.exponent = 0
-- the change of variable v = y^{1-n} transforms the above equation to
-- dv/dx + (1 - n) p v + (1 - n) q = 0
solveBernoulli(rec, y, x, yx) ==
n1 := 1 - rec.exponent::Integer
deq := differentiate(yx, x) + n1 * rec.coef1 * yx + n1 * rec.coefn
sol := solve(deq, y, x)::SOL
-- can always solve for order 1
-- if v = vp + c v0 is the general solution of the linear equation, then
-- the general first integral for the Bernoulli equation is
-- (y^{1-n} - vp) / v0 = c for any constant c
-- (yx**n1 - sol.particular) / first(sol.basis)

-- check whether the equation is of the form
-- dy/dx + q0(x) y + q1(x)y + q2(x)y^2 = 0
-- i.e. whether $m/n$ is a quadratic polynomial in $y$.
-- returns the list $[q_0, q_1, q_2]$ if the equation is in that form
checkRiccati($m, n, ky$) ==
  q := denom($f := m / n$)::$F$
  (not freeOf?(q, y := ky::$F$)) or degree($p := \text{univariate}(\text{numer } f, ky)$) > 2
  or (not freeOf?($a_0 := (\text{coefficient}(p, 0)::F), y$))
  or (not freeOf?($a_1 := (\text{coefficient}(p, 1)::F), y$))
  or (not freeOf?($a_2 := (\text{coefficient}(p, 2)::F), y$)) => "failed"
  $[a_0 / q, a_1 / q, a_2 / q]$

-- solves the equation $dy/dx + l.1 + l.2 y + l.3 y^2 = 0$
solveRiccati($1, y, x, yx$) ==
-- get first a particular solution
  ($u := \text{partSolRiccati}(1, y, x, yx)$ case "failed" => "failed"
-- once a particular solution $y_p$ is known, the general solution is of the
-- form $y = y_p + 1/v$ where $v$ satisfies the linear 1st order equation
-- $v' - (1.2 + 2 * 1.3 y_p) v = 1.3$
  $\text{deq := differentiate}(yx, x) - (1.2 + 2 * 1.3 * u::F) * yx - 1.3$
  $\text{gsol := solve}(\text{deq}, y, x)::\text{SOL}$
-- can always solve for order 1
-- if $v = v_p + c v_0$ is the general solution of the above equation, then
-- the general first integral for the Riccati equation is
-- $(1/(y - y_p) - v_0) / v_0 = c$ for any constant $c$
  $(\text{inv}(yx - u::F) - \text{gsol.particular}) / \text{first(gsol.basis)}$

-- looks for a particular solution of $dy/dx + 1.1 + 1.2 y + 1.3 y^2 = 0$
partSolRiccati($1, y, x, yx$) ==
-- we first do the change of variable $y = z / 1.3$, which transforms
-- the equation into $dz/dx + 1.1 1.3 + (1.2 - 1.3'/1.3) z + z^2 = 0$
  $q_0 := 1.1 * (13 := 1.3)$
  $q_1 := 1.2 - \text{differentiate}(13, x) / 13$
-- the equation $dz/dx + q_0 + q_1 z + z^2 = 0$ is transformed by the change
-- of variable $z = w'/w$ into the linear equation $w'' + q_1 w' + q_0 w = 0$
  $\text{lineq := differentiate}(yx, x, 2) + q_1 * \text{differentiate}(yx, x) + q_0 * yx$
-- should be made faster by requesting a particular nonzero solution only
  (not($\text{gsol := solve}(\text{lineq}, y, x)$ case $\text{SOL}$))
  or empty?($\text{basis := (gsol::SOL).basis}$) => "failed"
  $\text{differentiate(first bas, x)} / (13 * \text{first bas})$
package NLINSOL NonLinearSolvePackage

--- NonLinearSolvePackage.input ---

)set break resume
)sys rm -f NonLinearSolvePackage.output
)spool NonLinearSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NonLinearSolvePackage
--R
--R NonLinearSolvePackage(R: IntegralDomain) is a package constructor
--R Abbreviation for NonLinearSolvePackage is NLINSOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NLINSOL
--R
--R------------------------------- Operations --------------------------------
--R
--R solve : (List(Polynomial(R)),List(Symbol)) -> List(List(Equation(Fraction(Polynomial(R)))))
--R solve : List(Polynomial(R)) -> List(List(Equation(Fraction(Polynomial(R)))))
--R solveInField : (List(Polynomial(R)),List(Symbol)) -> List(List(Equation(Fraction(Polynomial(R)))))
--R solveInField : List(Polynomial(R)) -> List(List(Equation(Fraction(Polynomial(R)))))
--R
--E 1

)spool
)lisp (bye)

---

--- NonLinearSolvePackage.help ---

NonLinearSolvePackage examples

NonLinearSolvePackage is an interface to SystemSolvePackage that attempts to retract the coefficients of the equations before solving. The solutions are given in the algebraic closure of R whenever possible.

See Also:
  o )show NonLinearSolvePackage
NonLinearSolvePackage (NLINSOL)

Exports:
solve  solveInField

— package NLINSOL NonLinearSolvePackage —

)abbrev package NLINSOL NonLinearSolvePackage
++ Author: Manuel Bronstein
++ Date Created: 31 October 1991
++ Date Last Updated: 26 June 1992
++ Description:
++ NonLinearSolvePackage is an interface to \spadtype{SystemSolvePackage}
++ that attempts to retract the coefficients of the equations before
++ solving. The solutions are given in the algebraic closure of \emph{R} whenever
++ possible.

NonLinearSolvePackage(R: IntegralDomain):Exports == Implementation where
Z ==> Integer
Q ==> Fraction Z
SY ==> Symbol
P ==> Polynomial R
F ==> Fraction P
EQ ==> Equation F
SSP ==> SystemSolvePackage
SOL ==> RetractSolvePackage

Exports => with
solveInField: (List P, List SY) -> List List EQ
  ++ solveInField(lp,lv) finds the solutions of the list lp of
  ++ rational functions with respect to the list of symbols lv.
solveInField: List P -> List List EQ
  ++ solveInField(lp) finds the solution of the list lp of rational
  ++ functions with respect to all the symbols appearing in lp.
solve: (List P, List SY) -> List List EQ
++ solve(lp,lv) finds the solutions in the algebraic closure of R
++ of the list lp of
++ rational functions with respect to the list of symbols lv.
solve: List P -> List List EQ
++ solve(lp) finds the solution in the algebraic closure of R
++ of the list lp of rational
++ functions with respect to all the symbols appearing in lp.

Implementation ==> add
solveInField l == solveInField(l, "setUnion"/[variables p for p in l])

if R has AlgebraicallyClosedField then
import RationalFunction(R)

expandSol: List EQ -> List List EQ
RIFCan : F -> Union(R, "failed")
addRoot : (EQ, List List EQ) -> List List EQ
allRoots : List P -> List List EQ
evalSol : (List EQ, List EQ) -> List EQ

solve l == solve(l, "setUnion"/[variables p for p in l])
solve(lp, lv) == concat([expandSol sol for sol in solveInField(lp, lv)])
addRoot(eq, l) == [concat(eq, sol) for sol in l]
evalSol(ls, l) == [equation(lhs eq, eval(rhs eq, l)) for eq in ls]

-- converts \{p_1(a_1),\ldots,p_n(a_n)\} to
-- \{[a_1=v_1,\ldots,a_n=v_n]\} where vi ranges over all the zeros of pi
allRoots l ==
empty? l => [empty()$List(EQ)]
z := allRoots rest l
s := mainVariable(p := first l)::SY::P::F
concat [addRoot(equation(s, a::P::F), z) for a in zerosOf univariate p]

expandSol l ==
lassign := lsubs := empty()$List(EQ)
luniv := empty()$List(P)
for eq in l repeat
  if retractIfCan(lhs eq)@Union(SY, "failed") case SY then
    if RIfCan(rhs eq) case R then lassign := concat(eq, lassign)
    else lsubs := concat(eq, lsubs)
  else
    if ((u := retractIfCan(lhs eq)@Union(P, "failed")) case P) and
one?(# variables(u::P)) and ((r := RIfCan rhs eq) case R) then
      ((# variables(u::P)) = 1) and ((r := RIfCan rhs eq) case R) then
        luniv := concat(u::P - r::R::P, luniv)
    else return [l]
  empty? luniv => [l]
[concat(z, concat(evalSol(lsubs,z), lassign)) for z in allRoots luniv]
RIfCan f ==
  ((n := retractIfCan(numer f)@Union(R,"failed")) case R) and
  ((d := retractIfCan(denom f)@Union(R,"failed")) case R) => n::R / d::R
  "failed"
else
  solve l == solveInField l
solve(lp, lv) == solveInField(lp, lv)

-- 'else if' is doubtful with this compiler so all 3 conditions are explicit
if (not(R is Q)) and (R has RetractableTo Q) then
  solveInField(lp, lv) == solveRetract(lp, lv)$SOL(Q, R)
if (not(R is Z)) and (not(R has RetractableTo Q)) and
  (R has RetractableTo Z) then
  solveInField(lp, lv) == solveRetract(lp, lv)$SOL(Z, R)
if (not(R is Z)) and (not(R has RetractableTo Q)) and
  (not(R has RetractableTo Z)) then
  solveInField(lp, lv) == solve([p::F for p in lp]$List(F), lv)$SSP(R)

| NLINSOL.dotabb |

"NLINSOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NLINSOL"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"NLINSOL" -> "ACF"

package NORMPK NormalizationPackage

| NormalizationPackage.input |

)set break resume
)sys rm -f NormalizationPackage.output
)spool NormalizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NormalizationPackage
--R
NormalizationPackage(R: GcdDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategory(R,E,V),TS: RegularTriangularSetCategory(R,E,V,P)) is a package constructor

Abbreviation for NormalizationPackage is NORMPK

This constructor is exposed in this frame.

Issue )edit bookvol10.4.pamphlet to see algebra source code for NORMPK

---

------------------------------- Operations --------------------------------

normalizedAssociate: (P,TS) -> P
normInvertible?: (P,TS) -> List(Record(val: Boolean,tower: TS))
normalize: (P,TS) -> List(Record(val: P,tower: TS))
outputArgs: (String,String,P,TS) -> Void
recip: (P,TS) -> Record(num: P,den: P)

---

)spool
)lisp (bye)

— NormalizationPackage.help —

NormalizationPackage examples

A package for computing normalized associates of univariate polynomials with coefficients in a tower of simple extensions of a field.

See Also:
o )show NormalizationPackage
NormalizationPackage (NORMPK)

Exports:
normalizedAssociate normInvertible? normalize outputArgs recip

— package NORMPK NormalizationPackage —

)abbrev package NORMPK NormalizationPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ References:
++ [1] D. LAZARD "A new method for solving algebraic systems of
++ algebraic towers of simple extensions" In proceedings of AAECC11
++ d'extensions simples et resolution des systemes d'equations
++ Description:
++ A package for computing normalized associates of univariate polynomials
++ with coefficients in a tower of simple extensions of a field.

NormalizationPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
K ==> Fraction R
LP ==> List P
PWT ==> Record(val : P, tower : TS)

BWT ==> Record(val : Boolean, tower : TS)

LpWT ==> Record(val : (List P), tower : TS)

Split ==> List TS

-- KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)

-- EntryGcd ==> List PWT

-- HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)

-- KeyInvSet ==> Record(arg1: P, arg3: TS)

-- EntryInvSet ==> List TS

-- HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)

polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)

regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports == with

recip: (P, TS) -> Record(num:P, den:P)
++ \texttt{recip(p,ts)} returns the inverse of \texttt{p} w.r.t \texttt{ts}
++ assuming that \texttt{p} is invertible w.r.t \texttt{ts}.

normalizedAssociate: (P, TS) -> P
++ \texttt{normalizedAssociate(p,ts)} returns a normalized polynomial
++ \texttt{p} w.r.t \texttt{ts} such that \texttt{p} and \texttt{n} are
++ associates w.r.t \texttt{ts} and assuming that \texttt{p} is invertible
++ w.r.t \texttt{ts}.

normalize: (P, TS) -> List PWT
++ \texttt{normalize(p,ts)} normalizes \texttt{p} w.r.t \texttt{ts}.

outputArgs: (S, S, P, TS) -> Void
++ \texttt{outputArgs(s1,s2,p,ts)}
++ is an internal subroutine, exported only for development.

normInvertible?: (P, TS) -> List BWT
++ \texttt{normInvertible?(p,ts)}
++ is an internal subroutine, exported only for development.

Implementation == add

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P) then


stoseInvertible?_sqfreg(p,ts)$regsetgcdpack

else


stoseInvertible?_reg(p,ts)$regsetgcdpack

if (R has RetractableTo(Integer)) and (V has ConvertibleTo(Symbol)) then

if not empty? s1 then output(s1, p::OutputForm)$OutputPackage
if not empty? s1 then _
   output(s1,(convert(p)@String)::OutputForm)$OutputPackage
output(" ")$OutputPackage
if not empty? s2 then output(s2, ts::OutputForm)$OutputPackage
empty? s2 => void()
output(s2,("[")::OutputForm)$OutputPackage
lp: List P := members(ts)
for q in lp repeat
   output((convert(q)@String)::OutputForm)$OutputPackage
output(" ")$OutputPackage
else
if not empty? s1 then output(s1, p::OutputForm)$OutputPackage
output(" ")$OutputPackage
if not empty? s2 then output(s2, ts::OutputForm)$OutputPackage
output(" ")$OutputPackage
   -- ASSUME p is invertible w.r.t. ts
   -- ASSUME mvar(p) is algebraic w.r.t. ts
   v := mvar(p)
   ts_v := select(ts,v)::P
   if mdeg(p) < mdeg(ts_v)
      then
         hesrg: Record (gcd : P, coef2 : P) := _
            halfExtendedSubResultantGcd2(ts_v,p)$P
         d: P := hesrg.gcd; n: P := hesrg.coef2
      else
         hesrg: Record (gcd : P, coef1 : P) := _
            halfExtendedSubResultantGcd1(p,ts_v)$P
         d: P := hesrg.gcd; n: P := hesrg.coef1
   g := gcd(n,d)
   (n, d) := ((n exquo g)::P, (d exquo g)::P)
   remn, remd: Record(rnum:R,polnum:P,den:R)
   remn := remainder(n,ts); remd := remainder(d,ts)
   cn := remn.rnum; pn := remn.polnum; dn := remn.den
   cd := remd.rnum; pd := remd.polnum; dp := remd.den
   k: K := (cn / cd) * (dp / dn)
   pn := removeZero(pn,ts)
   pd := removeZero(pd,ts)
   [numer(k) * pn, denom(k) * pd]$Record(num:P, den:P)
normalizedAssociate(p:P,ts:TS): P ==
   -- ASSUME p is invertible or zero w.r.t. ts
   empty? ts => p
   zero?(p) => p
ground?(p) => 1
zero? initiallyReduce(init(p),ts) =>
  error "in normalizedAssociate$NORMPK: bad #1"
vp := mvar(p)
ip: P := p
mp: P := 1
tp: P := 0
while not ground?(ip) repeat
  v := mvar(ip)
  if algebraic?(v,ts)
    then
      if v = vp
      then
        ts_v := select(ts,v)::P
        ip := lastSubResultant(ip,ts_v)$P
        ip := remainder(ip,ts).polnum
        -- ip := primitivePart stronglyReduce(ip,ts)
        ip := primitivePart initiallyReduce(ip,ts)
      else
        qr := recip(ip,ts)
        ip := qr.den
        tp := qr.num * tp
        zero? ip =>
          outputArgs("p = ", " ts = ",p,ts)
          error _
          "in normalizedAssociate$NORMPK: should never happen !"
        else
          tp := tail(ip) * mp + tp
          mp := mainMonomial(ip) * mp
          ip := init(ip)
r := ip * mp + tp
r := remainder(r,ts).polnum
  -- primitivePart stronglyReduce(r,ts)
  primitivePart initiallyReduce(r,ts)

normalize(p: P, ts: TS): List PWT ==
  zero? p => [[p,ts]$PWT]
ground? p => [[1,ts]$PWT]
zero? initiallyReduce(init(p),ts) =>
  error "in normalize$NORMPK: init(#1) reduces to 0 w.r.t. #2"
  --output("Entering normalize")$OutputPackage
  --outputArgs("p = ", " ts = ",p,ts)
  --output("Calling normInvertible?")$OutputPackage
lbwt: List BWT := normInvertible?(p,ts)
  --output("Result is: ")$OutputPackage
  --output(lbwt::OutputForm)$OutputPackage
lpwt: List PWT := []
for bwt in lbwt repeat
  us := bwt.tower
  q := remainder(p,us).polnum
q := removeZero(q,us)
bwt.val =>$output("Calling normalizedAssociate")$OutputPackage
--outputArgs("q = ", " us = ",q,us)
lwp := cons([normalizedAssociate(q,us)@P,us]$PWT, lwp)
--output("Leaving normalizedAssociate")$OutputPackage
zero? q =>$output([0$P,us]$PWT, lwp)
lwp := concat(normalize(q,us)@(List PWT),lwp)
lwp

——— NORMPK.dotabb ———

"NORMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"NORMPK" -> "SFRTCAT"

package NORMMA NormInMonogenicAlgebra

— NormInMonogenicAlgebra.input —

)set break resume
)sys rm -f NormInMonogenicAlgebra.output
)spool NormInMonogenicAlgebra.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NormInMonogenicAlgebra
--R
--R NormInMonogenicAlgebra(R: GcdDomain,PoR: UnivariatePolynomialCategory(R),E: MonogenicAlgebra)
--R Abbreviation for NormInMonogenicAlgebra is NORMMA
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NORMMA
--R
--R----------------------------------- Operations ---------------------------------
--R norm : PolE -> PoR
--R
--E 1
This package implements the norm of a polynomial with coefficients in a monogenic algebra (using resultants)

See Also:
  o )show NormInMonogenicAlgebra

Exports:
  norm

This package implements the norm of a polynomial with coefficients in a monogenic algebra (using resultants)
++ in a monogenic algebra (using resultants)

NormInMonogenicAlgebra(R, PolR, E, PolE): Exports == Implementation where
R: GcdDomain
PolR: UnivariatePolynomialCategory R
E: MonogenicAlgebra(R, PolR)
PolE: UnivariatePolynomialCategory E

SUP ==> SparseUnivariatePolynomial

Exports ==> with
norm: PolE -> PolR
++ norm q returns the norm of q,
++ i.e. the product of all the conjugates of q.

Implementation ==> add
import UnivariatePolynomialCategoryFunctions2(R, PolR, PolR, SUP PolR)

Po1R2SUP: PolR -> SUP PolR
Po1R2SUP q == map(x +-> x::PolR, q)
defpol := Po1R2SUP(definingPolynomial()$E)
norm q ==
p:SUP PolR := 0
while q ~= 0 repeat
  p := p + monomial(1,degree q)$PolR * Po1R2SUP lift leadingCoefficient q
  q := reductum q
  primitivePart resultant(p, defpol)

package NORMRETR NormRetractPackage
---

**NormRetractPackage**

---

This package has no description

See Also:

- `)show NormRetractPackage`

---
NormRetractPackage (NORMRETR)

Exports:

Frobenius normFactors retractIfCan

— package NORMRETR NormRetractPackage —

)abbrev package NORMRETR NormRetractPackage
++ Description:
++ This package has no description

NormRetractPackage(F, ExtF, SUEEx, ExtP, n):C == T where
F : FiniteFieldCategory
ExtF : FiniteAlgebraicExtensionField(F)
SUEEx : UnivariatePolynomialCategory ExtF
ExtP : UnivariatePolynomialCategory SUEEx
n : PositiveInteger
SUP ==> SparseUnivariatePolynomial
R ==> SUP F
P ==> SUP R

C ==> with
  normFactors : ExtP -> List ExtP
  ++ normFactors(x) \ undocumented
  retractIfCan : ExtP -> Union(P, "failed")
  ++ retractIfCan(x) \ undocumented
  Frobenius : ExtP -> ExtP
  ++ Frobenius(x) \ undocumented

T ==> add

  normFactors(p:ExtP):List ExtP ==
    facs : List ExtP := [p]
    for i in 1..n-1 repeat
      member?((p := Frobenius p), facs) => return facs
      facs := cons(p, facs)
    facs

  Frobenius
Frobenius(ff:ExtP):ExtP ==
  fft:=ExtP:=0
  while ff ^= 0 repeat
    fft:=fft + monomial(map(Frobenius, leadingCoefficient ff),
                       degree ff)
    ff:=reductum ff
  fft

retractIfCan(ff:ExtP):Union(P, "failed") ==
  fft:=P:=0
  while ff ^= 0 repeat
    lc := SUEx := leadingCoefficient ff
    plc := SUP F := 0
    while lc ^= 0 repeat
      lclc := ExtF := leadingCoefficient lc
      (retlc := retractIfCan lclc) case "failed" => return "failed"
      plc := plc + monomial(retlc::F, degree lc)
      lc := reductum lc
    fft:=fft+monomial(plc, degree ff)
  fft:=reductum ff
  fft

— NORMRETR.dotabb —

"NORMRETR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NORMRETR"]
"FAXF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FAXF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"NORMRETR" -> "FAXF"
"NORMRETR" -> "PFECAT"

— NPCoef.input —

)set break resume
)sys rm -f NPCoef.output
)spool NPCoef.output
)set message test on
)set message auto off
Package for the determination of the coefficients in the lifting process. Used by MultivariateLifting. This package will work for every euclidean domain $R$ which has property $F$, i.e. there exists a factor operation in $R[x]$.

See Also:

- show NPCoef
NPCoef (NPCOEF)

Exports:
  listexp  npcoef

   — package NPCOEF NPCoef —

)abbrev package NPCOEF NPCoef
++ Author : P.Gianni, revised May 1990
++ Description:
++ Package for the determination of the coefficients in the lifting
++ process. Used by \spadtype{MultivariateLifting}.
++ This package will work for every euclidean domain \( R \) which has property
++ \( F \), i.e. there exists a factor operation in \( \mathbb{R}[x] \).

NPCoef(BP,E,OV,R,P) : C == T where

  OV : OrderedSet
  E : OrderedAbelianMonoidSup
  R : EuclideanDomain  -- with property "F"
  BP : UnivariatePolynomialCategory R
  P : PolynomialCategory(R,E,OV)

  Z ==> Integer
  NNI ==> NonNegativeInteger
  USP ==> SparseUnivariatePolynomial(P)
  Term ==> Record(expt:NNI,pcoef:P)
  Detc ==> Record(valexp:NNI,valcoef:P,posit:NNI)
  VTerm ==> List(Term)
  DetCoef ==> Record(deter:List(USP),dterm:List(VTerm),
                   nfacts:List(BP),nlead:List(P))
  TermC ==> Record(coefu:P,detfacts:List(VTerm))
  TCoef ==> List(TermC)

C == with
  npcoef : (USP,List(BP),List(P)) -> DetCoef
  ++ npcoef \undocumented
\begin{verbatim}
listexp : BP -> List(NNI) ++ listexp \undocumented
T == add

---- Local Functions ----
check : (TermC, Vector P) -> Union(Detc, "failed")
buildvect : (List(VTerm), NNI) -> Vector(List(VTerm))
buildtable : (Vector(P), List(List NNI), List P) -> TCoef
modify : (TCoef, Detc) -> TCoef
constructp : VTerm -> USP
npcoef(u:USP, factlist:List(BP), leadlist:List(P)) :DetCoef ==
detcoef:List(VTerm):=empty(); detufact:List(USP):=empty()
lexp:List(List(NNI)):=\{listexp(v) for v in factlist\}
ulist :Vector(P):=vector [coefficient(u,i) for i in 0..degree u]
tablecoef:=buildtable(ulist, lexp, leadlist)
detcoef:=[[ep.first, lcu]$Term for ep in lexp for lcu in leadlist]
ldtcf:=detcoef
lexp:=[ep.rest for ep in lexp]
ndet:NNI:=#factlist
changed:Boolean:=true
ltochange:List(NNI):=empty()
ltochange:List(NNI):=empty()
while changed and ndet\neq 1 repeat
  changed :=false
  dt:=#tablecoef
  for i in 1..dt while ^changed repeat
    (cf:=check(tablecoef.i, ulist)) case "failed" => "next i"
    ltochange:=cons(i, ltochange)
celf:Detc:=cf::Detc
    tablecoef:=modify(tablecoef, celf)
    vpos:=celf.posit
    vexp:=celf.valexp
    nterm:=[vexp, celf.valcoef]$Term
    detcoef.vpos:=cons(nterm, detcoef.vpos)
    lexp.vpos:=delete(lexp.vpos, position(vexp, lexp.vpos))
    if lexp.vpos=[] then
      ltochange:=cons(vpos, ltochange)
      ndet:=(ndet-1):NNI
      detufact:=cons(constructp(detcoef.vpos), detufact)
      changed:=true
    for i in ltochange repeat tablecoef:=delete(tablecoef, i)
  ltochange:=[[]
  if ndet=1 then
    uu:=u exquo */[pol for pol in detufact]
    if uu case "failed" then return
      [empty(), ldtcf, factlist, leadlist]$DetCoef
    else detufact:=cons(uu::USP, detufact)
    end
    ltochange:=[(n1:NNI, n2:NNI):Boolean => n1\n2, ltochange]
\end{verbatim}
for i in ltodel repeat
    detcoef:=delete(detcoef,i)
    factlist:=delete(factlist,i)
    leadlist:=delete(leadlist,i)
[detufact,detcoef,factlist,leadlist]$DetCoef

check(tterm:TermC,ulist:Vector(P)) : Union(Detc,"failed") ==
  cfu:P:=1$P;doit:NNI:=0;poselt:NNI:=0;pp:Union(P,"failed")
  termlist:List(VTerm):=tterm.detfacts
  vterm:VTerm:=empty()
  #termlist=1 =>
  vterm:=termlist.first
  for elterm in vterm while doit<2 repeat
    (cu1:=elterm.pcoef)^=0 => cfu:=cu1*cfu
    doit:=doit+1
    poselt:=position(elterm,vterm):NNI
    doit=2 or (pp:=tterm.coefu exquo cfu) case "failed" => "failed"
    [vterm.poselt.expt,pp::P,poselt]$Detc
"failed"

buildvect(lvterm:List(VTerm),n:NNI) : Vector(List(VTerm)) ==
  vtable:Vector(List(VTerm)):=new(n,empty())
  (#lvterm)=1 =>
  for term in lvterm.first repeat vtable.(term.expt+1):=[term]
  vtable

  vtable:=buildvect(lvterm.rest,n)
  ntable:Vector(List(VTerm)):=new(n,empty())
  for term in lvterm.first repeat
    nexp:=term.expt
    for i in 1..n while (nexp+i)<(n+1) repeat
      ntable.(nexp+i):=append(
        [cons(term,lvterm) for lvterm in vtable.i],
        ntable.(nexp+i))

  ntable

buildtable(vu:Vector(P),lvect:List(List(NNI)),leadlist:List(P)):TCoef==
  nfact:NNI:=#leadlist
  table:TCoef:=empty()
  degu:=(#vu-1)::NNI
  prelim:List(VTerm):=[[e,0$P]$Term for e in lv] for lv in lvect]
  for i in 1..nfact repeat prelim.i.first.pcoef:=leadlist.i
  partialv:Vector(List(VTerm)):=new(nfact,empty())
  partialv:=buildvect(prelim,degu)
  for i in 1..degu repeat
    empty? partialv.i => "next i"
    table:=cons([vu.i,partialv.i]$TermC, table)
  table
modify(tablecoef:TCoef,cfter:Detc) : TCoef ==
cfexp:=cfter.valexp;cfcoef:=cfter.valcoef;cfpos:=cfter.posit
lterase:List(NNI):=empty()
for cterm in tablecoef | "empty?(ctdet:=cterm.detfacts) repeat
  (+/[term.expt for term in ctdet.first])<cfexp => "next term"
  for celt in ctdet repeat
    if celt.cfpos.expt=cfexp then
      celt.cfpos.pcoef:=cfcoef
    if (and/[cc.pcoef ^=0 for cc in celt]) then
      k:=position(celt,ctdet):NNI
      lterase:=cons(k,lterase)
      cterm.coefu:=(cterm.coefu - */[cc.pcoef for cc in celt])
    if not empty? lterase then
      lterase:=sort((n1:NNI,n2:NNI):Boolean +-> n1>n2,lterase)
      for i in lterase repeat ctdet:=delete(ctdet,i)
      cterm.detfacts:=ctdet
      lterase:=empty()
    tablecoef

listexp(up:BP) :List(NNI) ==
degree up=0 => [0]
[degree up,:listexp(reductum up)]

constructp(lterm:VTerm):USP ==
  +/[monomial(term.pcoef,term.expt) for term in lterm]

— NPCOEF.dotabb —

"NPCOEF" [color="#FF4488",href="bookvol110.4.pdf#nameddest=NPCOEF"]
"PFECAT" [color="#4488FF",href="bookvol110.2.pdf#nameddest=PFECAT"]
"NPCOEF" -> "PFECAT"

— NumberFieldIntegralBasis.input —

)set break resume
)sys rm -f NumberFieldIntegralBasis.output
)spool NumberFieldIntegralBasis.output
)set message test on
--S 1 of 1
)show NumberFieldIntegralBasis
--R
--R NumberFieldIntegralBasis(UP: UnivariatePolynomialCategory(Integer),F: FramedAlgebra(Integer,UP)) is a package constructor
--R NumberFieldIntegralBasis is NFINTBAS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NFINTBAS
--R
--R----------------------------------- Operations --------------------------------
--R discriminant : () -> Integer
--R integralBasis : () -> Record(basis: Matrix(Integer),basisDen: Integer,basisInv: Matrix(Integer))
--R localIntegralBasis : Integer -> Record(basis: Matrix(Integer),basisDen: Integer,basisInv: Matrix(Integer))
--E 1

)spool
)lisp (bye)

---

NumberFieldIntegralBasis.help
---

NumberFieldIntegralBasis examples
---

In this package F is a framed algebra over the integers (typically F = Z[a] for some algebraic integer a). The package provides functions to compute the integral closure of Z in the quotient field of F.

See Also:
 o )show NumberFieldIntegralBasis

---
NumberFieldIntegralBasis (NFINTBAS)

Exports:
  discriminant  integralBasis  localIntegralBasis

| package NFINTBAS NumberFieldIntegralBasis |

abbrev package NFINTBAS NumberFieldIntegralBasis
++ Author: Victor Miller, Clifton Williamson
++ Date Created: 9 April 1990
++ Date Last Updated: 20 September 1994
++ Description:
  ++ In this package F is a framed algebra over the integers (typically
  ++ \spad{F = Z[a]} for some algebraic integer a). The package provides
  ++ functions to compute the integral closure of Z in the quotient
  ++ field of F.

NumberFieldIntegralBasis(UP,F): Exports == Implementation where
  UP : UnivariatePolynomialCategory Integer
  F : FramedAlgebra(Integer,UP)

  FR ==> Factored Integer
  I  ==> Integer
  Mat ==> Matrix I
  NNI ==> NonNegativeInteger
  Ans ==> Record(basis: Mat, basisDen: I, basisInv:Mat,discr: I)

Exports == with
  discriminant: () -> Integer
    ++ \spad{discriminant()} returns the discriminant of the integral
    ++ closure of Z in the quotient field of the framed algebra F.
  integralBasis : () -> Record(basis: Mat, basisDen: I, basisInv:Mat)
    ++ \spad{integralBasis()} returns a record
    ++ containing information regarding the integral closure of Z in the
    ++ quotient field of F, where F is a framed algebra with Z-module
    ++ basis \spad{\{w1,w2,...,wn\}}.


++ If \( \text{basis} \) is the matrix \( (a_{ij}, i = 1..n, j = 1..n) \), then
++ the \( i \)th element of the integral basis is
++ \( v_i = (1/\text{basisDen}) \times \sum a_{ij} w_j, j = 1..n \), i.e. the
++ \( i \)th row of \( \text{basis} \) contains the coordinates of the
++ \( i \)th basis vector. Similarly, the \( i \)th row of the
++ matrix \( \text{basisInv} \) contains the coordinates of \( w_i \) with
++ respect to the basis \( v_1,\ldots,v_n \): if \( \text{basisInv} \) is the
++ matrix \( (b_{ij}, i = 1..n, j = 1..n) \), then
++ \( w_i = \sum b_{ij} v_j, j = 1..n \).

\[
\text{localIntegralBasis} : \, \mathbb{I} \rightarrow \text{Record} (\text{basis} : \text{Mat}, \text{basisDen} : \mathbb{I}, \text{basisInv} : \text{Mat})
\]
++ \( \text{integralBasis}(p) \) returns a record
++ containing information regarding
++ the local integral closure of \( \mathbb{Z} \) at the prime \( p \) in the quotient
++ field of \( F \), where \( F \) is a framed algebra with \( \mathbb{Z} \)-module basis
++ \( v_1,\ldots,v_n \).
++ If \( \text{basis} \) is the matrix \( (a_{ij}, i = 1..n, j = 1..n) \), then
++ the \( i \)th element of the integral basis is
++ \( v_i = (1/\text{basisDen}) \times \sum a_{ij} w_j, j = 1..n \), i.e. the
++ \( i \)th row of \( \text{basis} \) contains the coordinates of the
++ \( i \)th basis vector. Similarly, the \( i \)th row of the
++ matrix \( \text{basisInv} \) contains the coordinates of \( w_i \) with
++ respect to the basis \( v_1,\ldots,v_n \): if \( \text{basisInv} \) is the
++ matrix \( (b_{ij}, i = 1..n, j = 1..n) \), then
++ \( w_i = \sum b_{ij} v_j, j = 1..n \).

Implementation ==> add
import IntegralBasisTools(I, UP, F)
import ModularHermitianRowReduction(I)
import TriangularMatrixOperations(I, Vector I, Vector I, Matrix I)

def frobMatrix(rbb,rb,rbden,p) ==
  n := rank()$F; b := basis()$F
  v : Vector F := new(n,0)
  for i in minIndex(v).maxIndex(v)
    for ii in minRowIndex(vb)..maxRowIndex(rb) repeat
      a : F := 0
      for j in minIndex(b)...maxIndex(b)
        for jj in minColIndex(rb)..maxColIndex(rb) repeat
          a := a + qelt(rb,ii,jj) * qelt(b,j)
        qsetelt_!(v,i,a**p)
  mat := transpose coordinates v
  ((transpose(rb) * mat) exquo (rbden ** p)) :: Mat

def wildPrimes(factoredDisc,n) ==
-- returns a list of the primes <= n which divide factoredDisc to a
-- power greater than 1
ans : List I := empty()
for f in factors(factoredDisc) repeat
  if f.exponent > 1 and f.factor <= n then ans := concat(f.factor,ans)
ans

(-- returns the product of the primes > n which divide factoredDisc
-- to a power greater than 1)
ans : I := 1
for f in factors(factoredDisc) repeat
  if f.exponent > 1 and f.factor > n then ans := f.factor * ans
ans

integralBasis() ==
  traceMat := traceMatrix()$F; n := rank()$F
  disc := determinant traceMat -- discriminant of current order
  disc0 := disc -- this is disc(F)
  factoredDisc := factor(disc0)$IntegerFactorizationPackage(Integer)
  wilds := wildPrimes(factoredDisc, n)
  sing := tameProduct(factoredDisc, n)
  runningRb := scalarMatrix(n, 1); runningRbinv := scalarMatrix(n, 1)
  runningRbden : I := 1
  for p in wilds repeat
    lb := iWildLocalIntegralBasis(matrixOut, disc, p)
    rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
    disc := lb.discr
    -- update 'running integral basis' if newly computed
    -- local integral basis is non-trivial
    if sizeLess?(1, rbden) then
      mat := vertConCat(rbden * runningRb, runningRbden * rb)
      runningRb := runningRbden * rbden
      runningRbinv := UpTriBddDenomInv(runningRb, runningRbden)
      lb := iTameLocalIntegralBasis(traceMat, disc, sing)
      rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
      disc := lb.discr
      -- update 'running integral basis' if newly computed
      -- local integral basis is non-trivial
      if sizeLess?(1, rbden) then
        mat := vertConCat(rbden * runningRb, runningRbden * rb)
runningRbden := runningRbden * rbden
runningRb := squareTop rowEchelon(mat, runningRbden)
runningRbinv := UpTriBddDenomInv(runningRb, runningRbden)
[runningRb, runningRbden, runningRbinv]

localIntegralBasis p ==
  traceMat := traceMatrix()$F; n := rank()$F
  disc := determinant traceMat  -- discriminant of current order
  (disc exquo (p*p)) case "failed" =>
    [scalarMatrix(n, 1), 1, scalarMatrix(n, 1)]
  lb :=
    p > rank()$F =>
    iTameLocalIntegralBasis(traceMat, disc, p)
    iWildLocalIntegralBasis(scalarMatrix(n,0), disc, p)
  [lb.basis, lb.basisDen, lb.basisInv]

iTameLocalIntegralBasis(traceMat, disc, sing) ==
  n := rank()$F; disc0 := disc
  rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
  -- rb = basis matrix of current order
  -- rbinv = inverse basis matrix of current order
  -- these are wrt the original basis for F
  rbden : I := 1; index : I := 1; oldIndex : I := 1
  -- rbden = denominator for current basis matrix
  -- id = basis matrix of the ideal (p-radical) wrt current basis
  tfm := traceMat
  repeat
    -- compute the p-radical = p-trace-radical
    idinv := transpose squareTop rowEchelon(tfm, sing)
    -- [u1,...,un] are the coordinates of an element of the p-radical
    -- iff [u1,...,un] * idinv is in p * Z^n
    id := rowEchelon LowTriBddDenomInv(idinv, sing)
    -- id = basis matrix of the p-radical
    idinv := UpTriBddDenomInv(id, sing)
    -- id * idinv = sing * identity
    -- no need to check for inseparability in this case
    rbinv := idealiser(id * rb, rbinv * idinv, sing * rbden)
    index := diagonalProduct rbinv
    rb := rowEchelon LowTriBddDenomInv(rbinv, sing * rbden)
    g := matrixGcd(rb, sing, n)
    if sizeLess?(1,g) then rb := (rb exquo g) :: Mat
    rbden := rbden * (sing quo g)
    rbinv := UpTriBddDenomInv(rb, rbden)
    disc := disc0 quo (index * index)
    indexChange := index quo oldIndex; oldIndex := index
    -- one? indexChange => return [rb, rbden, rbinv, disc]
    (indexChange = 1) => return [rb, rbden, rbinv, disc]
  tfm := ((rb * traceMat * transpose rb) exquo (rbden * rbden)) :: Mat

iWildLocalIntegralBasis(matrixOut, disc, p) ==
n := rank()$F; disc0 := disc
rb := scalarMatrix(n, 1); rbinv := scalarMatrix(n, 1)
-- rb = basis matrix of current order
-- rbinv = inverse basis matrix of current order
-- these are wrt the original basis for F
rbden : I := 1; index : I := 1; oldIndex : I := 1
-- rbden = denominator for current basis matrix
-- id = basis matrix of the ideal (p-radical) wrt current basis
p2 := p * p; lp := leastPower(p::NNI,n)
repeat
  tfm := frobMatrix(rb,rbinv,rbden,p::NNI) ** lp
  -- compute Rp = p-radical
  idinv := transpose squareTop rowEchelon(tfm, p)
  -- [u1,...,un] are the coordinates of an element of Rp
  -- iff [u1,...,un] * idinv is in p * Z^n
  id := rowEchelon LowTriBddDenomInv(idinv,p)
  -- id = basis matrix of the p-radical
  idinv := UpTriBddDenomInv(id,p)
  -- id * idinv = p * identity
  -- no need to check for inseparability in this case
  rbinv := idealiser(id * rb, rbinv * idinv, p * rbden)
  index := diagonalProduct rbinv
  rb := rowEchelon LowTriBddDenomInv(rbinv, p * rbden)
  if divideIfCan!(rb,matrixOut,p,n) = 1
    then rb := matrixOut
    else rbden := p * rbden
    rbinv := UpTriBddDenomInv(rb, rbden)
    indexChange := index quo oldIndex; oldIndex := index
    disc := disc quo (indexChange * indexChange)
    -- one? indexChange or gcd(p2,disc) ^= p2 =>
    -- (indexChange = 1) or gcd(p2,disc) ^= p2 =>
    return [rb, rbden, rbinv, disc]
  else
    return [rb, rbden, rbinv, disc]
end repeat

discriminant() ==
  disc := determinant traceMatrix()$F
  intBas := integralBasis()
  rb := intBas.basis; rbden := intBas.basisDen
  index := ((rbden ** rank()$F) exquo (determinant rb)) :: Integer
  (disc exquo (index * index)) :: Integer

— NFINTBAS.dotabb —
NumberFormats examples
====================================================================
NumberFormats provides function to format and read arabic and roman numbers, to convert numbers to strings and to read floating-point numbers.

See Also:
NumberFormats (NUMFMT)

Exports:
FormatArabic FormatRoman ScanArabic ScanFloatIgnoreSpaces
ScanFloatIgnoreSpacesIfCan ScanRoman

— package NUMFMT NumberFormats —

)abbrev package NUMFMT NumberFormats
++ SMW March 88
++ Description:
++ NumberFormats provides function to format and read arabic and
++ roman numbers, to convert numbers to strings and to read
++ floating-point numbers.

NumberFormats(): NFexports == NFimplementation where
PI ==> PositiveInteger
I ==> Integer
C ==> Character
F ==> Float
S ==> String
V ==> PrimitiveArray

NFexports => with
FormatArabic: PI -> S
++ FormatArabic(n) forms an Arabic numeral
++ string from an integer n.
ScanArabic: S -> PI
++ ScanArabic(s) forms an integer from an Arabic numeral string s.
FormatRoman: PI -> S
++ FormatRoman(n) forms a Roman numeral string from an integer n.
ScanRoman: S -> PI
++ ScanRoman(s) forms an integer from a Roman numeral string s.
ScanFloatIgnoreSpaces: S -> F
++ ScanFloatIgnoreSpaces(s) forms a floating point number from
++ the string s ignoring any spaces. Error is generated if the
++ string is not recognised as a floating point number.
ScanFloatIgnoreSpacesIfCan: S -> Union(F, "failed")
++ ScanFloatIgnoreSpacesIfCan(s) tries to form a floating point
++ number from the string s ignoring any spaces.

NFimplementation ==>> add
import SExpression
import Symbol
replaceD: C -> C
replaced: C -> C
contract: S -> S
check: S -> Boolean

replaceD c ==
  if c = char "D" then char "E" else c

replaced c ==
  if c = char "d" then char "E" else c

contract s ==
  s:= map(replaceD,s)
  s:= map(replaced,s)

ls:List S := split(s,char " ")$String
s:= concat ls

check s ==
  NUMBERP(READ_FROM_STRING(s)$Lisp)$Lisp and
  -- if there is an "E" then there must be a "."
  -- this is not caught by code above
  -- also if the exponent is v.big the above returns false
  not (any?((c1:C):Boolean +-> c1=char "E",s)
  and not any?((c2:C):Boolean +-> c2=char ".",s) )

-- Original interpreter function:
-- )lis (defun scanstr(x) (spadcomp::|parseFromString| x))
sexfloat:SExpression:=convert(coerce("Float")@Symbol)$SExpression

ScanFloatIgnoreSpaces s ==
  s := contract s
  not check s => error "Non-numeric value"
  sex := interpret(ncParseFromString(s)$Lisp)$Lisp
  sCheck := car(car(sex))
  if (sCheck=sexfloat) = true then
    f := (cdr cdr sex) pretend Float
  else
    if integer?(cdr sex) = true then
      f := (cdr sex) pretend Integer
    f::F
def ScanFloatIgnoreSpacesIfCan(s):
    s = contract(s)
    if not check(s):
        return "failed"
    sex = interpret(ncParseFromString(s))
    sCheck = car(car(sex))
    if sCheck == sexfloat:
        f = (cdr(cdr(sex))) pretend Float
    else:
        if integer?(cdr(sex)):
            f = (cdr(sex)) pretend Integer
        else:
            return "failed"
    return f

def ScanArabic(s):
    return PARSE_INTEGER(s)

def FormatArabic(n):
    return PRINC_TO_STRING(n)

def FormatRoman(pn):
    n = pn::Integer
    # Units
    d = (n rem 10) + umin
    n = n quo 10
    s = units.d
    return s
-- Tens
d := (n rem 10) + tmin
n := n quo 10
s := concat(tens.d, s)
zero? n => s
-- Hundreds
d := (n rem 10) + hmin
n := n quo 10
s := concat(hunds.d, s)
zero? n => s
-- Thousands
d := n rem 10
n := n quo 10
s := concat(new(d::NonNegativeInteger, thou), s)
zero? n => s
-- Ten thousand and higher
for i in 2.. while not zero? n repeat
  -- Coefficient of 10**(i+2)
d := n rem 10
n := n quo 10
zero? d => "iterate"
m0:String := concat(new(i,plen),concat("I",new(i,pren)))
mm := concat([m0 for j in 1..d]$List(String))
  -- strictly speaking the blank is gratuitous
if #s > 0 then s := concat(" ", s)
s := concat(mm, s)
s
-- ScanRoman
--
-- The Algorithm:
-- Read number from right to left. When the current
-- numeral is lower in magnitude than the previous maximum
-- then subtract otherwise add.
-- Shift left and repeat until done.
ScanRoman s ==
s := upperCase s
tot: I := 0
Max: I := 0
i: I := maxIndex s
while i >= minIndex s repeat
  -- Read a single roman digit
c := s.i; i := i-1
n := romval ord c
  -- (I)=1000, ((I))=10000, (((I)))=100000, etc
if n < 0 then
c ^= pren =>
  error ["Improper character in Roman numeral: ",c]
nprens: PI := 1
while c = pren and i >= minIndex s repeat
    c := s.i; i := i-1
    if c = pren then nprens := nprens+1
    c ^= ichar =>
        error "Improper Roman numeral: (x)"
for k in 1..nprens while i >= minIndex s repeat
    c := s.i; i := i-1
    c ^= plen =>
        error "Improper Roman numeral: unbalanced ')'
    n := 10**(nprens + 2)
    if n < Max then
        tot := tot - n
    else
        tot := tot + n
    Max := n
    tot < 0 => error ["Improper Roman numeral: ", tot]
tot::PI

package NTPOLFN NumberTheoreticPolynomialFunctions

-- NumberTheoreticPolynomialFunctions.input --

)set break resume
)sys rm -f NumberTheoreticPolynomialFunctions.output
)spool NumberTheoreticPolynomialFunctions.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show NumberTheoreticPolynomialFunctions
--R
--R NumberTheoreticPolynomialFunctions(R: CommutativeRing) is a package constructor
--R Abbreviation for NumberTheoreticPolynomialFunctions is NTPOLFN
This package provides polynomials as functions on a ring.

See Also:
- )show NumberTheoreticPolynomialFunctions

---

NumberTheoreticPolynomialFunctions (NTPOLFN)

Exports:
bernoulliB  cyclotomic  eulerE
package NTPOLFN NumberTheoreticPolynomialFunctions —

)abbrev package NTPOLFN NumberTheoreticPolynomialFunctions
++ Author: Stephen M. Watt
++ Date Created: 1990
++ Date Last Updated: June 25, 1991
++ Description:
++ This package provides polynomials as functions on a ring.

NumberTheoreticPolynomialFunctions(R: CommutativeRing):Exports == Impl where
  NNI ==> NonNegativeInteger
  RN ==> Fraction Integer

Exports == with
  cyclotomic: (NNI, R) -> R
    ++ cyclotomic(n,r) undocumented
    if R has Algebra RN then
      bernoulliB: (NNI, R) -> R
        ++ bernoulliB(n,r) undocumented
      eulerE: (NNI, R) -> R
        ++ eulerE(n,r) undocumented

Impl ==> add
  import PolynomialNumberTheoryFunctions()

I ==> Integer
SUP ==> SparseUnivariatePolynomial

-- This is the wrong way to evaluate the polynomial.
cyclotomic(k, x) ==
p: SUP(I) := cyclotomic(k)
r: R := 0
while p ^= 0 repeat
d := degree p
c := leadingCoefficient p
p := reductum p
r := c*x**d + r
r

if R has Algebra RN then
eulerE(k, x) ==
p: SUP(RN) := euler(k)
r: R := 0
while p ^= 0 repeat
d := degree p
c := leadingCoefficient p
\[
\begin{align*}
p &:= \text{reductum } p \\
r &:= c \times^d + r \\
r
\end{align*}
\]

\[
\text{bernoulliB}(k, x) ==
\begin{align*}
p &:= \text{SUP(RN)} := \text{bernoulli}(k) \\
r &:= R := 0 \\
\text{while } p \neq 0 \text{ repeat} \\
d &:= \text{degree } p \\
c &:= \text{leadingCoefficient } p \\
p &:= \text{reductum } p \\
r &:= c \times^d + r
\end{align*}
\]

**package NUMERIC Numeric**

--- Numeric.input ---

)set break resume  
)sys rm -f Numeric.output  
)spool Numeric.output  
)set message test on  
)set message auto off  
)clear all

--S 1 of 1
)show Numeric

--R Numeric(S: ConvertibleTo(Float)) is a package constructor

--R Abbreviation for Numeric is NUMERIC

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NUMERIC

--R-------------------------------------- Operations --------------------------------------

--R complexNumeric : S -> Complex(Float) numeric : S -> Float
CHAPTER 15. CHAPTER N

--R complexNumeric : (S,PositiveInteger) -> Complex(Float)
--R complexNumeric : Complex(S) -> Complex(Float) if S has COMRING
--R complexNumeric : (Complex(S),PositiveInteger) -> Complex(Float) if S has COMRING
--R complexNumeric : Polynomial(Complex(S)) -> Complex(Float) if S has COMRING
--R complexNumeric : (Polynomial(Complex(S)),PositiveInteger) -> Complex(Float) if S has COMRING
--R complexNumeric : Polynomial(S) -> Complex(Float) if S has RING
--R complexNumeric : (Polynomial(S),PositiveInteger) -> Complex(Float) if S has RING
--R complexNumeric : Fraction(Polynomial(S)) -> Complex(Float) if S has INTDOM
--R complexNumeric : (Fraction(Polynomial(S)),PositiveInteger) -> Complex(Float) if S has INTDOM
--R complexNumeric : Fraction(Polynomial(Complex(S))) -> Complex(Float) if S has INTDOM
--R complexNumeric : (Fraction(Polynomial(Complex(S))),PositiveInteger) -> Complex(Float) if S has INTDOM
--R complexNumeric : Expression(S) -> Complex(Float) if S has INTDOM and S has ORDSET
--R complexNumeric : (Expression(S),PositiveInteger) -> Complex(Float) if S has INTDOM and S has ORDSET
--R complexNumeric : (Expression(Complex(S)),PositiveInteger) -> Complex(Float) if S has INTDOM and S has ORDSET
--R complexNumeric : (Expression(Complex(S)),PositiveInteger) -> Complex(Float) if S has INTDOM and S has ORDSET
--R complexNumericIfCan : Polynomial(Complex(S)) -> Union(Complex(Float),"failed") if S has COMRING
--R complexNumericIfCan : (Polynomial(Complex(S)),PositiveInteger) -> Union(Complex(Float),"failed") if S has COMRING
--R complexNumericIfCan : Polynomial(S) -> Union(Complex(Float),"failed") if S has RING
--R complexNumericIfCan : (Polynomial(S),PositiveInteger) -> Union(Complex(Float),"failed") if S has RING
--R complexNumericIfCan : Fraction(Polynomial(S)) -> Union(Complex(Float),"failed") if S has INTDOM
--R complexNumericIfCan : (Fraction(Polynomial(S)),PositiveInteger) -> Union(Complex(Float),"failed") if S has INTDOM
--R complexNumericIfCan : Fraction(Polynomial(Complex(S))) -> Union(Complex(Float),"failed") if S has INTDOM
--R complexNumericIfCan : (Fraction(Polynomial(Complex(S))),PositiveInteger) -> Union(Complex(Float),"failed") if S has INTDOM
--R complexNumericIfCan : Expression(S) -> Union(Complex(Float),"failed") if S has INTDOM and S has ORDSET
--R complexNumericIfCan : (Expression(S),PositiveInteger) -> Union(Complex(Float),"failed") if S has INTDOM and S has ORDSET
--R complexNumericIfCan : Expression(Complex(S)) -> Union(Complex(Float),"failed") if S has INTDOM and S has ORDSET
--R complexNumericIfCan : (Expression(Complex(S)),PositiveInteger) -> Union(Complex(Float),"failed") if S has INTDOM and S has ORDSET
--R numeric : (S,PositiveInteger) -> Float
--R numeric : (Polynomial(S),PositiveInteger) -> Float if S has RING
--R numeric : (Fraction(Polynomial(S)),PositiveInteger) -> Float if S has RING
--R numeric : Fraction(Polynomial(S)) -> Float if S has INTDOM
--R numeric : (Fraction(Polynomial(S)),PositiveInteger) -> Float if S has INTDOM
--R numeric : Expression(S) -> Float if S has INTDOM and S has ORDSET
--R numericIfCan : Polynomial(S) -> Union(Float,"failed") if S has RING
--R numericIfCan : (Polynomial(S),PositiveInteger) -> Union(Float,"failed") if S has RING
--R numericIfCan : Fraction(Polynomial(S)) -> Union(Float,"failed") if S has INTDOM
--R numericIfCan : (Fraction(Polynomial(S)),PositiveInteger) -> Union(Float,"failed") if S has INTDOM
--R numericIfCan : Expression(S) -> Union(Float,"failed") if S has INTDOM and S has ORDSET
--R numericIfCan : (Expression(S),PositiveInteger) -> Union(Float,"failed") if S has INTDOM and S has ORDSET

--E 1

)spool
)lisp (bye)

—— Numeric.help ——
Numeric examples

Numeric provides real and complex numerical evaluation functions for various symbolic types.

See Also:
o )show Numeric
complexNumeric: S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x up
  ++ to n decimal places.
if S has CommutativeRing then
complexNumeric: Complex S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Complex S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x up
  ++ to n decimal places.
complexNumeric: Polynomial Complex S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Polynomial Complex S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x up
  ++ to n decimal places.
if S has Ring then
numeric: Polynomial S -> Float
  ++ numeric(x) returns a real approximation of x.
numeric: (Polynomial S, PositiveInteger) -> Float
  ++ numeric(x, n) returns a real approximation of x up to n decimal
  ++ places.
complexNumeric: Polynomial S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Polynomial S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x up
  ++ to n decimal places.
if S has IntegralDomain then
numeric: Fraction Polynomial S -> Float
  ++ numeric(x) returns a real approximation of x.
numeric: (Fraction Polynomial S, PositiveInteger) -> Float
  ++ numeric(x, n) returns a real approximation of x up to n decimal
  ++ places.
complexNumeric: Fraction Polynomial S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Fraction Polynomial S, PositiveInteger) -> Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x.
complexNumeric: Fraction Polynomial Complex S -> Complex Float
  ++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Fraction Polynomial Complex S, PositiveInteger) ->
  Complex Float
  ++ complexNumeric(x, n) returns a complex approximation of x up
  ++ to n decimal places.
if S has OrderedSet then
numeric: Expression S -> Float
  ++ numeric(x) returns a real approximation of x.
numeric: (Expression S, PositiveInteger) -> Float
  ++ numeric(x, n) returns a real approximation of x up to n
  ++ decimal places.
complexNumeric: Expression S -> Complex Float
++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Expression S, PositiveInteger) -> Complex Float
++ complexNumeric(x, n) returns a complex approximation of x
++ up to n decimal places.
complexNumeric: Expression Complex S -> Complex Float
++ complexNumeric(x) returns a complex approximation of x.
complexNumeric: (Expression Complex S, PositiveInteger) -> Complex Float
++ complexNumeric(x, n) returns a complex approximation of x
++ up to n decimal places.

if S has CommutativeRing then
complexNumericIfCan: Polynomial Complex S -> Union(Complex Float,"failed")
++ complexNumericIfCan(x) returns a complex approximation of x,
++ or "failed" if \axiom{x} is not constant.
complexNumericIfCan: (Polynomial Complex S, PositiveInteger) -> Union(Complex Float,"failed")
++ complexNumericIfCan(x, n) returns a complex approximation of x up
++ to n decimal places, or "failed" if \axiom{x} is not a constant.

if S has Ring then
numericIfCan: Polynomial S -> Union(Float,"failed")
++ numericIfCan(x) returns a real approximation of x,
++ or "failed" if \axiom{x} is not a constant.
numericIfCan: (Polynomial S, PositiveInteger) -> Union(Float,"failed")
++ numericIfCan(x,n) returns a real approximation of x up to n decimal
++ places, or "failed" if \axiom{x} is not a constant.
complexNumericIfCan: Polynomial Complex S -> Union(Complex Float,"failed")
++ complexNumericIfCan(x) returns a complex approximation of x,
++ or "failed" if \axiom{x} is not a constant.
complexNumericIfCan: (Polynomial Complex S, PositiveInteger) -> Union(Complex Float,"failed")
++ complexNumericIfCan(x, n) returns a complex approximation of x
++ up to n decimal places, or "failed" if \axiom{x} is not a constant.

if S has IntegralDomain then
numericIfCan: Fraction Polynomial S -> Union(Float,"failed")
++ numericIfCan(x) returns a real approximation of x,
++ or "failed" if \axiom{x} is not a constant.
numericIfCan: (Fraction Polynomial S, PositiveInteger) -> Union(Float,"failed")
++ numericIfCan(x,n) returns a real approximation of x up to n decimal
++ places, or "failed" if \axiom{x} is not a constant.
complexNumericIfCan: Fraction Polynomial Complex S -> Union(Complex Float,"failed")
++ complexNumericIfCan(x) returns a complex approximation of x,
++ or "failed" if \axiom{x} is not a constant.
complexNumericIfCan: (Fraction Polynomial Complex S, PositiveInteger) -> Union(Complex Float,"failed")
++ complexNumericIfCan(x, n) returns a complex approximation of x
++ up to n decimal places, or "failed" if \axiom{x} is not a constant.

if S has OrderedSet then
numericIfCan: Expression S -> Union(Float,"failed")
++ numericIfCan(x) returns a real approximation of x,
++ or "failed" if \texttt{\textbackslash axiom}(x) is not a constant.
numericIfCan: (Expression S, PositiveInteger) -> Union(Float,"failed")
++ numericIfCan(x, n) returns a real approximation of x up to n
++ decimal places, or "failed" if \texttt{\textbackslash axiom}(x) is not a constant.
complexNumericIfCan: Expression S -> Union(Complex Float,"failed")
++ complexNumericIfCan(x) returns a complex approximation of x,
++ or "failed" if \texttt{\textbackslash axiom}(x) is not a constant.
complexNumericIfCan: (Expression S, PositiveInteger) ->
  Union(Complex Float,"failed")
++ complexNumericIfCan(x, n) returns a complex approximation of x
++ up to n decimal places, or "failed" if \texttt{\textbackslash axiom}(x) is not a constant.
complexNumericIfCan: Expression Complex S -> Union(Complex Float,"failed")
++ complexNumericIfCan(x) returns a complex approximation of x,
++ or "failed" if \texttt{\textbackslash axiom}(x) is not a constant.
complexNumericIfCan: (Expression Complex S, PositiveInteger) ->
  Union(Complex Float,"failed")
++ complexNumericIfCan(x, n) returns a complex approximation of x
++ up to n decimal places, or "failed" if \texttt{\textbackslash axiom}(x) is not a constant.

== add

if S has CommutativeRing then
  complexNumericIfCan(p:Polynomial Complex S) ==
    p' : Union(Complex(S),"failed") := retractIfCan p
    p' case "failed" => "failed"
    complexNumeric(p')

complexNumericIfCan(p:Polynomial Complex S,n:PositiveInteger) ==
  p' : Union(Complex(S),"failed") := retractIfCan p
  p' case "failed" => "failed"
  complexNumeric(p',n)

if S has Ring then
  numericIfCan(p:Polynomial S) ==
    p' : Union(S,"failed") := retractIfCan p
    p' case "failed" => "failed"
    numeric(p')

complexNumericIfCan(p:Polynomial S) ==
  p' : Union(S,"failed") := retractIfCan p
  p' case "failed" => "failed"
  complexNumeric(p')

complexNumericIfCan(p:Polynomial S, n:PositiveInteger) ==
  p' : Union(S,"failed") := retractIfCan p
  p' case "failed" => "failed"
  complexNumeric(p', n)

numericIfCan(p:Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numericIfCan p
digits(old)$Float
ans

if S has IntegralDomain then
numericIfCan(f:Fraction Polynomial S) ==
num := numericIfCan(numer(f))
num case "failed" => "failed"
den := numericIfCan(denom f)
den case "failed" => "failed"
num/den

complexNumericIfCan(f:Fraction Polynomial S) ==
num := complexNumericIfCan(numer f)
num case "failed" => "failed"
den := complexNumericIfCan(denom f)
den case "failed" => "failed"
num/den

complexNumericIfCan(f:Fraction Polynomial S, n:PositiveInteger) ==
num := complexNumericIfCan(numer f, n)
num case "failed" => "failed"
den := complexNumericIfCan(denom f, n)
den case "failed" => "failed"
num/den

numericIfCan(f:Fraction Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numericIfCan f
digits(old)$Float
ans

complexNumericIfCan(f:Fraction Polynomial Complex S) ==
num := complexNumericIfCan(numer f)
num case "failed" => "failed"
den := complexNumericIfCan(denom f)
den case "failed" => "failed"
num/den

complexNumericIfCan(f:Fraction Polynomial Complex S, n:PositiveInteger) ==
num := complexNumericIfCan(numer f, n)
num case "failed" => "failed"
den := complexNumericIfCan(denom f, n)
den case "failed" => "failed"
num/den

if S has OrderedSet then
numericIfCan(x:Expression S) ==
retractIfCan(map(convert, x)$ExpressionFunctions2(S, Float))
CHAPTER N

--s2cs(u:S):Complex(S) == complex(u,0)

complexNumericIfCan(x:Expression S) ==
  complexNumericIfCan map(coerce, x)$ExpressionFunctions2(S,Complex S)

numericIfCan(x:Expression S, n:PositiveInteger) ==
  old := digits(n)$Float
  x' : Expression Float := map(convert, x)$ExpressionFunctions2(S, Float)
  ans : Union(Float,"failed") := retractIfCan x'
  digits(old)$Float
  ans

complexNumericIfCan(x:Expression S, n:PositiveInteger) ==
  old := digits(n)$Float
  x' : Expression Complex S := map(coerce, x)$ExpressionFunctions2(S, Complex S)
  ans : Union(Complex Float,"failed") := complex NumericIfCan(x')
  digits(old)$Float
  ans

if S has RealConstant then
  complexNumericIfCan(x:Expression Complex S) ==
    retractIfCan(map(convert, x)$ExpressionFunctions2(Complex S,Complex Float))

complexNumericIfCan(x:Expression Complex S, n:PositiveInteger) ==
  old := digits(n)$Float
  x' : Expression Complex Float :=
    map(convert, x)$ExpressionFunctions2(Complex S,Complex Float)
  ans : Union(Complex Float,"failed") := retractIfCan x'
  digits(old)$Float
  ans
else
  convert(x:Complex S):Complex(Float)==map(convert,x)$ComplexFunctions2(S,Float)

complexNumericIfCan(x:Expression Complex S) ==
  retractIfCan(map(convert, x)$ExpressionFunctions2(Complex S,Complex Float))

complexNumericIfCan(x:Expression Complex S, n:PositiveInteger) ==
  old := digits(n)$Float
  x' : Expression Complex Float :=
    map(convert, x)$ExpressionFunctions2(Complex S,Complex Float)
  ans : Union(Complex Float,"failed") := retractIfCan x'
  digits(old)$Float
  ans
numeric(s:S) == convert(s)@Float

if S has ConvertibleTo Complex Float then
  complexNumeric(s:S) == convert(s)@Complex(Float)

complexNumeric(s:S, n:PositiveInteger) ==
old := digits(n)$Float
ans := complexNumeric s
digits(old)$Float
ans

else
complexNumeric(s:S) == convert(s)$Float :: Complex(Float)

complexNumeric(s:S,n:PositiveInteger) ==
numeric(s, n)::Complex(Float)

if S has CommutativeRing then
complexNumeric(p:Polynomial Complex S) ==
p' : Union(Complex(S),"failed") := retractIfCan p
p' case "failed" =>
  error "Cannot compute the numerical value of a non-constant polynomial"
complexNumeric(p')

complexNumeric(p:Polynomial Complex S,n:PositiveInteger) ==
p' : Union(Complex(S),"failed") := retractIfCan p
p' case "failed" =>
  error "Cannot compute the numerical value of a non-constant polynomial"
complexNumeric(p',n)

if S has RealConstant then
complexNumeric(s:Complex S) == convert(s)$Complex(S)

complexNumeric(s:Complex S, n:PositiveInteger) ==
old := digits(n)$Float
ans := complexNumeric s
digits(old)$Float
ans

else if Complex(S) has ConvertibleTo(Complex Float) then
complexNumeric(s:Complex S) == convert(s)$Complex(Float)

complexNumeric(s:Complex S, n:PositiveInteger) ==
old := digits(n)$Float
ans := complexNumeric s
digits(old)$Float
ans

else
complexNumeric(s:Complex S) ==
s' : Union(S,"failed") := retractIfCan s
s' case "failed" =>
  error "Cannot compute the numerical value of a non-constant object"
complexNumeric(s')

complexNumeric(s:Complex S, n:PositiveInteger) ==
s' : Union(S,"failed") := retractIfCan s
s' case "failed" =>
  error "Cannot compute the numerical value of a non-constant object"
old := digits(n)$Float
ans := complexNumeric s'
digits(old)$Float
ans

numeric(s:S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numeric s
digits(old)$Float
ans

if S has Ring then
  numeric(p:Polynomial S) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" => error "Can only compute the numerical value of a constant, real-valued polynomial"
numeric(p')

complexNumeric(p:Polynomial S) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" =>
  error "Cannot compute the numerical value of a non-constant polynomial"
complexNumeric(p')

complexNumeric(p:Polynomial S, n:PositiveInteger) ==
p' : Union(S,"failed") := retractIfCan p
p' case "failed" =>
  error "Cannot compute the numerical value of a non-constant polynomial"
complexNumeric(p', n)

numeric(p:Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numeric p
digits(old)$Float
ans

if S has IntegralDomain then
  numeric(f:Fraction Polynomial S) ==
    numeric(numer(f)) / numeric(denom f)

complexNumeric(f:Fraction Polynomial S) ==
  complexNumeric(numer f)/complexNumeric(denom f)

complexNumeric(f:Fraction Polynomial S, n:PositiveInteger) ==
  complexNumeric(numer f, n)/complexNumeric(denom f, n)

numeric(f:Fraction Polynomial S, n:PositiveInteger) ==
old := digits(n)$Float
ans := numeric f
digits(old)$Float
ans

complexNumeric(f:Fraction Polynomial Complex S) ==
complexNumeric(numer f)/complexNumeric(denom f)

complexNumeric(f:Fraction Polynomial Complex S, n:PositiveInteger) ==
complexNumeric(numer f, n)/complexNumeric(denom f, n)

if S has OrderedSet then
numeric(x:Expression S) ==
x' : Union(Float,"failed") :=
  retractIfCan(map(convert, x)$ExpressionFunctions2(S, Float))
x' case "failed" => error
  "Can only compute the numerical value of a constant, real-valued Expression"
x'

complexNumeric(x:Expression S) ==
x' : Union(Complex Float,"failed") := retractIfCan(  
  map(complexNumeric, x)$ExpressionFunctions2(S,Complex Float))
x' case "failed" =>
  error "Cannot compute the numerical value of a non-constant expression"
x'

numeric(x:Expression S, n:PositiveInteger) ==
old := digits(n)$Float
x' : Expression Float := map(convert, x)$ExpressionFunctions2(S, Float)
anst : Union(Float,"failed") := retractIfCan x'
digits(old)$Float
ans case "failed" => error
  "Can only compute the numerical value of a constant, real-valued Expression"
anst

complexNumeric(x:Expression S, n:PositiveInteger) ==
old := digits(n)$Float
x' : Expression Complex Float :=
  map(complexNumeric, x)$ExpressionFunctions2(S,Complex Float)
anst : Union(Complex Float,"failed") := retractIfCan x'
digits(old)$Float
ans case "failed" =>
  error "Cannot compute the numerical value of a non-constant expression"
anst

complexNumeric(x:Expression Complex S) ==
x' : Union(Complex Float,"failed") := retractIfCan(  
  map(complexNumeric, x)$ExpressionFunctions2(Complex S,Complex Float))
x' case "failed" =>
  error "Cannot compute the numerical value of a non-constant expression"
complexNumeric(x: Expression Complex S, n: PositiveInteger) ==
old := digits(n)$Float
x' : Expression Complex Float :=
    map(complexNumeric, x)$ExpressionFunctions2(Complex S, Complex Float)
ans : Union(Complex Float, "failed") := retractIfCan x'
digits(old)$Float
ans case "failed" =>
    error "Cannot compute the numerical value of a non-constant expression"
ans

-----
— NUMERIC.dotabb —

"NUMERIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMERIC"]
"COMPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=COMPCAT"]
"NUMERIC" -> "COMPCAT"

-----

package NUMODE NumericalOrdinaryDifferentialEquations

— NumericalOrdinaryDifferentialEquations.input —

)set break resume
)sys rm -f NumericalOrdinaryDifferentialEquations.output
)spool NumericalOrdinaryDifferentialEquations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericalOrdinaryDifferentialEquations
--R
--R NumericalOrdinaryDifferentialEquations is a package constructor
--R Abbreviation for NumericalOrdinaryDifferentialEquations is NUMODE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NUMODE
--R
--R----------------------------------------------------------- Operations ---------------
This package is a suite of functions for the numerical integration of an ordinary differential equation of n variables:

\[ \frac{dy}{dx} = f(y,x) \quad y \text{ is an n-vector} \]

All the routines are based on a 4-th order Runge-Kutta kernel.

These routines generally have as arguments:

- n, the number of dependent variables;
- x1, the initial point;
- h, the step size;
- y, a vector of initial conditions of length n

which upon exit contains the solution at x1 + h

derivs, a function which computes the right hand side of the ordinary differential equation:

\[ \text{derivs(dydx,y,x)} \]

computes dydx, a vector which contains the derivative information.

In order of increasing complexity:

- \text{rk4(y,n,x1,h,derivs)}

advances the solution vector to
and return the values in $y$.

$$\text{rk4}(y,n,x1,h,\text{derivs},t1,t2,t3,t4)$$

is the same as

$$\text{rk4}(y,n,x1,h,\text{derivs})$$

except that you must provide 4 scratch arrays $t1$-$t4$ of size $n$

Starting with $y$ at $x1$,

$$\text{rk4f}(y,n,x1,x2,ns,\text{derivs})$$

uses $ns$ fixed steps of a 4-th order Runge-Kutta integrator to advance the solution vector to $x2$ and return the values in $y$. Argument $x2$, is the final point, and $ns$, the number of steps to take.

$$\text{rk4qc}(y,n,x1,\text{step},\text{eps},\text{yscal},\text{derivs})$$

takes a 5-th order Runge-Kutta step with monitoring of local truncation to ensure accuracy and adjust stepsize.

The function takes two half steps and one full step and scales the difference in solutions at the final point. If the error is within $\text{eps}$, the step is taken and the result is returned.

If the error is not within $\text{eps}$, the stepsize if decreased and the procedure is tried again until the desired accuracy is reached. Upon input, an trial step size must be given and upon return, an estimate of the next step size to use is returned as well as the step size which produced the desired accuracy.

The scaled error is computed as

$$\text{error} = \text{MAX}(\text{ABS}((y2\text{steps}(i) - y1\text{step}(i))/\text{yscal}(i)))$$

and this is compared against $\text{eps}$. If this is greater than $\text{eps}$, the step size is reduced accordingly to

$$h\text{new} = 0.9 * h\text{did} * (\text{error}/\text{eps})^{(-1/4)}$$

If the error criterion is satisfied, then we check if the step size was too fine and return a more efficient one. If

$$\text{error} > \text{eps} * (6.0\text{E}-04)$$
then the next step size should be

\[ h_{\text{next}} = 0.9 \times h_{\text{did}} \times (\text{error}/\text{eps})^{-1/5} \]

Otherwise

\[ h_{\text{next}} = 4.0 \times h_{\text{did}} \]

is returned.

A more detailed discussion of this and related topics can be found in the book "Numerical Recipies" by W. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling published by Cambridge University Press.

Argument step is a record of 3 floating point numbers (try, did, next), eps is the required accuracy, yscal is the scaling vector for the difference in solutions.

On input, step.try should be the guess at a step size to achieve the accuracy. On output, step.did contains the step size which achieved the accuracy and step.next is the next step size to use.

\[
\text{rk4qc}(y,n,x1,\text{step},\text{eps},\text{yscal},\text{derivs},t1,t2,t3,t4,t5,t6,t7)
\]

is the same as

\[
\text{rk4qc}(y,n,x1,\text{step},\text{eps},\text{yscal},\text{derivs})
\]

except that the user must provide the 7 scratch arrays t1-t7 of size n.

\[
\text{rk4a}(y,n,x1,x2,\text{eps},h,\text{ns},\text{derivs})
\]

is a driver program which uses rk4qc to integrate n ordinary differential equations starting at x1 to x2, keeping the local truncation error to within eps by changing the local step size.

The scaling vector is defined as

\[ \text{yscal}(i) = \text{abs}(y(i)) + \text{abs}(h\times dydx(i)) + \text{tiny} \]

where y(i) is the solution at location x, dydx is the ordinary differential equation's right hand side, h is the current step size and tiny is 10 times the smallest positive number representable.

The user must supply an estimate for a trial step size and the maximum number of calls to rk4qc to use. Argument x2 is the final point, eps is local truncation, ns is the maximum number of call to rk4qc to use.

See Also:
NumericalOrdinaryDifferentialEquations (NUMODE)

Exports:
  rk4  rk4a  rk4f  rk4qc  rk4qc

--- package NUMODE NumericalOrdinaryDifferentialEquations ---

>abbrev package NUMODE NumericalOrdinaryDifferentialEquations
++ Author: Yurij Baransky
++ Date Created: October 90
++ Date Last Updated: October 90
++ Description:
++ This package is a suite of functions for the numerical integration of an
++ ordinary differential equation of n variables:
++ \[ \frac{dy}{dx} = f(y,x) \] \[ y \] is an n-vector
++ All the routines are based on a 4-th order Runge-Kutta kernel.
++ These routines generally have as arguments:
++ \( n \), the number of dependent variables;
++ \( x_1 \), the initial point;
++ \( h \), the step size;
++ \( y \), a vector of initial conditions of length \( n \);
++ which upon exit contains the solution at \( x_1 + h \);
++ \( \text{derivs} \), a function which computes the right hand side of the
++ ordinary differential equation: \( \text{derivs}(dydx,y,x) \) computes
++ \( \text{derivs}(dydx) \), a vector which contains the derivative information.
++ In order of increasing complexity:
++ \[ \text{rk4}(y,n,x_1,h,\text{derivs}) \] advances the solution vector to
\( \text{rk4}(y, n, x_1, h, \text{derivs}) \) is the same as \( \text{rk4}(y, n, x_1, h, \text{derivs}, t_1, t_2, t_3, t_4) \) except that you must provide 4 scratch arrays \( t_1-t_4 \) of size \( n \).

Starting with \( y \) at \( x_1 \), \( \text{rk4f}(y, n, x_1, x_2, \text{ns}, \text{derivs}) \) uses \( \text{ns} \) fixed steps of a 4-th order Runge-Kutta integrator to advance the solution vector to \( x_2 \) and return \( y \). Argument \( x_2 \), is the final point, and \( \text{ns} \), the number of steps to take.

\( \text{rk4qc}(y, n, x_1, \text{step}, \text{eps}, \text{yscal}, \text{derivs}) \) takes a 5-th order Runge-Kutta step with monitoring of local truncation to ensure accuracy and adjust stepsize.

The function takes two half steps and one full step and scales the difference in solutions at the final point. If the error is within \( \text{eps} \), the step is taken and the result is returned. If the error is not within \( \text{eps} \), the stepsize if decreased and the procedure is tried again until the desired accuracy is reached. Upon input, an trial step size must be given and upon return, an estimate of the next step size to use is returned as well as the step size which produced the desired accuracy.

The scaled error is computed as

\[ \text{error} = \text{MAX}(\text{ABS}((y_{2\text{steps}}(i) - y_{1\text{step}}(i))/\text{yscal}(i))) \]

and this is compared against \( \text{eps} \). If this is greater than \( \text{eps} \), the step size is reduced accordingly to

\[ \text{hnew} = 0.9 \times \text{hdid} \times (\text{error}/\text{eps})^{-1/4} \]

If the error criterion is satisfied, then we check if the step size was too fine and return a more efficient one. If

\[ \text{error} > (\text{6.0E-04}) \] then the next step size should be

\[ \text{hnext} = 0.9 \times \text{hdid} \times (\text{error}/\text{eps})^{-1/5} \]

Otherwise \( \text{hnext} = 4.0 \times \text{hdid} \) is returned.

A more detailed discussion of this and related topics can be found in the book "Numerical Recipies" by W. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling published by Cambridge University Press.

Argument \( \text{step} \) is a record of 3 floating point numbers \( \text{try, did, next} \), \( \text{eps} \) is the required accuracy, \( \text{yscal} \) is the scaling vector for the difference in solutions. On input, \( \text{step.try} \) should be the guess at a step size to achieve the accuracy. On output, \( \text{step.did} \) contains the step size which achieved the accuracy and \( \text{step.next} \) is the next step size to use.

\( \text{rk4qc}(y, n, x_1, \text{step}, \text{eps}, \text{yscal}, \text{derivs}, t_1, t_2, t_3, t_4, t_5, t_6, t_7) \) is the same as \( \text{rk4qc}(y, n, x_1, \text{step}, \text{eps}, \text{yscal}, \text{derivs}) \) except that the user must provide the 7 scratch arrays \( \text{t1-t7} \) of size \( n \).
CHAPTER 15. CHAPTER N

++ \spad{rk4a(y,n,x1,x2,eps,h,ns,derivs)}
++ is a driver program which uses \spad{rk4qc} to integrate n ordinary
++ differential equations starting at x1 to x2, keeping the local
++ truncation error to within \spad{eps} by changing the local step size.
++ The scaling vector is defined as:
{}^{++} \text{tab(5)} \spad{yscal(i) = abs(y(i)) + abs(h*dydx(i)) + tiny}\text{br}
++ where \spad{y(i)} is the solution at location x, \spad{dydx(i)} is the
++ ordinary differential equation’s right hand side, h is the current
++ step size and \spad{tiny} is 10 times the
++ smallest positive number representable.
++
++ The user must supply an estimate for a trial step size and
++ the maximum number of calls to \spad{rk4qc} to use.
++ Argument x2 is the final point,
++ \spad{eps} is local truncation,
++ \spad{ns} is the maximum number of call to \spad{rk4qc} to use.

NumericalOrdinaryDifferentialEquations(): Exports == Implementation where

L ==> List
V ==> Vector
B ==> Boolean
I ==> Integer
E ==> OutputForm
NF ==> Float
NNI ==> NonNegativeInteger
VOID ==> Void
OFORM ==> OutputForm
RK4STEP ==> Record(try:NF, did:NF, next:NF)

Exports == Implementation where

--header definitions here
rk4 : (V NF,I,NF,NF,NF, (V NF,V NF,NF) -> VOID) -> VOID
++ rk4(y,n,x1,h,derivs) uses a 4-th order Runge-Kutta method
++ to numerically integrate the ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector.
++ Argument y is a vector of initial conditions of length n which upon exit
++ contains the solution at \spad{x1 + h}, n is the number of dependent
++ variables, x1 is the initial point, h is the step size, and
++ \spad{derivs} is a function which computes the right hand side of the
++ ordinary differential equation.
++ For details, see \spadtype{NumericalOrdinaryDifferentialEquations}.
rk4 : (V NF,I,NF,NF, (V NF,V NF,NF) -> VOID
++ \spad{rk4(y,n,x1,h,derivs)} is the same as
++ \spad{rk4(y,n,x1,h,derivs)} except that you must provide 4 scratch
++ arrays t1-t4 of size n.
++ For details, see \spad{NumericalOrdinaryDifferentialEquations}.
rk4a : (V NF,I,NF,NF,NF,NF,I,(V NF,V NF,NF) -> VOID ) -> VOID
++ rk4a(y,n,x1,x2,eps,h,ns,derivs) is a driver function for the
++ numerical integration of an ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector
++ using a 4-th order Runge-Kutta method.
++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
++ rk4qc : (V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID) -> VOID
++ rk4qc(y,n,x1,step,eps,yscal,derivs) is a subfunction for the
++ numerical integration of an ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector
++ using a 4-th order Runge-Kutta method.
++ This function takes a 5-th order Runge-Kutta step with monitoring
++ of local truncation to ensure accuracy and adjust steps.
++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
++ rk4qc : (V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID
     ,V NF,V NF,V NF,V NF,V NF,V NF,V NF) -> VOID
++ rk4qc(y,n,x1,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,t7) is a
++ subfunction for the numerical integration of an ordinary differential
++ equation dy/dx = f(y,x) of n variables, where y is an n-vector
++ using a 4-th order Runge-Kutta method.
++ This function takes a 5-th order Runge-Kutta step with monitoring
++ of local truncation to ensure accuracy and adjust steps.
++ For details, see \con{NumericalOrdinaryDifferentialEquations}.
++ rk4f : (V NF,I,NF,NF,I,(V NF,V NF,NF) -> VOID ) -> VOID
++ rk4f(y,n,x1,x2,ns,derivs) uses a 4-th order Runge-Kutta method
++ to numerically integrate the ordinary differential equation
++ dy/dx = f(y,x) of n variables, where y is an n-vector.
++ Starting with y at x1, this function uses \spad{ns} fixed
++ steps of a 4-th order Runge-Kutta integrator to advance the
++ solution vector to x2 and return the values in y.
++ For details, see \con{NumericalOrdinaryDifferentialEquations}.

Implementation ==> add
-- some local function definitions here
rk4qclocal : (V NF,V NF,I,NF,RK4STEP,NF,V NF,(V NF,V NF,NF) -> VOID
     ,V NF,V NF,V NF,V NF,V NF,V NF,V NF) -> VOID
rk4local : (V NF,V NF,I,NF,NF,V NF,(V NF,V NF,NF) -> VOID
     ,V NF,V NF,V NF) -> VOID
import OutputPackage

------------------------------------------------------------

rk4a(ystart,nvar,x1,x2,eps,htry,nstep,derivs) ==
y : V NF := new(nvar::NNI,0.0)
yscal : V NF := new(nvar::NNI,1.0)
dydx : V NF := new(nvar::NNI,0.0)
t1 : V NF := new(nvar::NNI,0.0)
t2 : V NF := new(nvar::NNI,0.0)
t3 : V NF := new(nvar::NNI,0.0)
t4 : V NF := new(nvar::NNI,0.0)
t5 : V NF := new(nvar::NNI,0.0)
t6 : V NF := new(nvar::NNI,0.0)
step : RK4STEP := [htry,0.0,0.0]
x : NF := x1
tiny : NF := 10.0**(-(digits()+1)::I)
m : I := nvar
outlist : L OFORM := [x::E,x::E,x::E]
i : I
iter : I

eps := 1.0/eps
for i in 1..m repeat
  y(i) := ystart(i)
for iter in 1..nstep repeat
  --compute the derivative
derivs(dydx,y,x)
  --if overshoot, the set h accordingly
  if (x + step.try - x2) > 0.0 then
    step.try := x2 - x
  --find the correct scaling
  for i in 1..m repeat
    yscal(i) := abs(y(i)) + abs(step.try * dydx(i)) + tiny
  --take a quality controlled runge-kutta step
  rk4qclocal(y,dydx,nvar,x,step,eps,yscal,derivs
    ,t1,t2,t3,t4,t5,t6)
  x := x + step.did
  -- outlist.0 := x::E
  -- outlist.1 := y(0)::E
  -- outlist.2 := y(1)::E
  -- output(blankSeparate(outlist)::E)
  --check to see if done
  if (x-x2) >= 0.0 then
    leave
  --next stepsize to use
  step.try := step.next
--end nstep repeat
if iter = (nstep+1) then
  output("ode: ERROR ")
  outline.1 := nstep::E
  outline.2 := " steps to small, last h = "::E
  outline.3 := step.did::E
  output(blankSeparate(outlist))
  output(" y= ",y::E)
for i in 1..m repeat
  ystart(i) := y(i)

----------------------------------------------------------------
rk4qc(y,n,x,step,eps,yscal,derivs) ==
t1 : V NF := new(n::NNI,0.0)
t2 : V NF := new(n::NNI,0.0)
t3 : V NF := new(n::NNI,0.0)
t4 : V NF := new(n::NNI,0.0)
t5 : V NF := new(n::NNI,0.0)
t6 : V NF := new(n::NNI,0.0)
t7 : V NF := new(n::NNI,0.0)
derivs(t7,y,x)
eps := 1.0/eps
rk4qclocal(y,t7,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6)

--------------------------------------------------------

rk4qc(y,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6,dydx) ==
derivs(dydx,y,x)
eps := 1.0/eps
rk4qclocal(y,dydx,n,x,step,eps,yscal,derivs,t1,t2,t3,t4,t5,t6)

--------------------------------------------------------

rk4qclocal(y,dydx,n,x,step,eps,yscal,derivs,t1,t2,t3,ysav,dysav,ytemp) ==
xsav : NF := x
h : NF := step.try
fcor : NF := 1.0/15.0
safety : NF := 0.9
grow : NF := -0.20
shrink : NF := -0.25
ercon : NF := 0.6E-04 --(this is 4/safety)**(1/grow)
hh : NF
ermax : NF
i : I
m : I := n
--
for i in 1..m repeat
dysav(i) := dydx(i)
ysav(i) := y(i)
--cut down step size till error criterion is met
repeat
--take two little steps to get to x + h
hh := 0.5 * h
rk4local(ysav,dysav,n,xsav,hh,ytemp,derivs,t1,t2,t3)
x := xsav + hh
derivs(dydx,ytemp,x)
rk4local(ytemp,dydx,n,x,hh,y,derivs,t1,t2,t3)
x := xsav + h
--take one big step get to x + h
rk4local(ysav,dysav,n,xsav,h,ytemp,derivs,t1,t2,t3)
--compute the maximum scaled difference
ermax := 0.0
for i in 1..m repeat
ytemp(i) := y(i) - ytemp(i)
ermax := max(errmax,abs(ytemp(i)/yscal(i)))
--scale relative to required accuracy
errmax := errmax * eps
--update integration stepsise
if (errmax > 1.0) then
    h := safety * h * (errmax ** shrink)
else
    step.did := h
    if errmax > errcon then
        step.next := safety * h * (errmax ** grow)
    else
        step.next := 4 * h
    leave
--make fifth order with 4-th order error estimate
for i in 1..m repeat
    y(i) := y(i) + ytemp(i) * fcor

--------------------------------------------
rk4f(y,nvar,x1,x2,nstep,derivs) ==
yt : V NF := new(nvar::NNI,0.0)
dyt : V NF := new(nvar::NNI,0.0)
dym : V NF := new(nvar::NNI,0.0)
dydx : V NF := new(nvar::NNI,0.0)
ynew : V NF := new(nvar::NNI,0.0)
h : NF := (x2-x1) / (nstep::NF)
x : NF := x1
i : I
j : I
-- start integrating
for i in 1..nstep repeat
    derivs(dydx,y,x)
rk4local(y,dydx,nvar,x,y,derivs,yt,dyt,dym)
x := x + h

--------------------------------------------------------
rk4(y,n,x,h,derivs) ==
t1 : V NF := new(n::NNI,0.0)
t2 : V NF := new(n::NNI,0.0)
t3 : V NF := new(n::NNI,0.0)
t4 : V NF := new(n::NNI,0.0)
derivs(t1,y,x)
rk4local(y,t1,n,x,h,y,derivs,t2,t3,t4)

------------------------------------------------------------
rk4(y,n,x,h,derivs,t1,t2,t3,t4) ==
derivs(t1,y,x)
rk4local(y,t1,n,x,h,y,derivs,t2,t3,t4)
rk4local(y,dydx,n,x,h,yout,derivs,yt,dyt,dym) ==
  hh : NF := h*0.5
  h6 : NF := h/6.0
  xh : NF := x+hh
  m : I := n
  i : I
  -- first step
  for i in 1..m repeat
    yt(i) := y(i) + hh*dydx(i)
  -- second step
  derivs(dyt,yt,xh)
  for i in 1..m repeat
    yt(i) := y(i) + hh*dyt(i)
  -- third step
  derivs(dym,yt,xh)
  for i in 1..m repeat
    yt(i) := y(i) + h*dym(i)
    dym(i) := dyt(i) + dym(i)
  -- fourth step
  derivs(dyt,yt,x+h)
  for i in 1..m repeat
    yout(i) := y(i) + h6*( dydx(i) + 2.0*dym(i) + dyt(i) )

——

— NUMODE.dotabb —

"NUMODE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMODE"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"NUMODE" -> "IVECTOR"

——

package NUMQUAD NumericalQuadrature

— NumericalQuadrature.input —

)set break resume
)sys rm -f NumericalQuadrature.output
)spool NumericalQuadrature.output
)set message test on
)set message auto off
NumericalQuadrature examples

This suite of routines performs numerical quadrature using algorithms derived from the basic trapezoidal rule. Because the error term of this rule contains only even powers of the step size (for open and closed versions), fast convergence can be obtained if the integrand is sufficiently smooth.

Each routine returns a Record of type TrapAns, which contains

- value Float: estimate of the integral
- error Float: estimate of the error in the computation
- totalpts Integer: total number of function evaluations
- success Boolean: if the integral was computed within the user specified error criterion

To produce this estimate, each routine generates an internal sequence of sub-estimates, denoted by S(i), depending on the
routine, to which the various convergence criteria are applied. The user must supply a relative accuracy, \( \epsilon_r \), and an absolute accuracy, \( \epsilon_a \). Convergence is obtained when either

\[
\text{ABS}(S(i) - S(i-1)) < \epsilon_r \times \text{ABS}(S(i-1))
\]

or

\[
\text{ABS}(S(i) - S(i-1)) < \epsilon_a
\]

are true statements.

The routines come in three families and three flavors:

- closed: romberg, simpson, trapezoidal
- open: rombergo, simpsono, trapezoidalo
- adaptive closed: aromberg, asimpson, atrapezoidal

The \( S(i) \) for the trapezoidal family is the value of the integral using an equally spaced abscissa trapezoidal rule for that level of refinement.

The \( S(i) \) for the simpson family is the value of the integral using an equally spaced abscissa simpson rule for that level of refinement.

The \( S(i) \) for the romberg family is the estimate of the integral using an equally spaced abscissa romberg method. For the \( i \)-th level, this is an appropriate combination of all the previous trapezodial estimates so that the error term starts with the \( 2^{*(i+1)} \) power only.

The three families come in a closed version, where the formulas include the endpoints, an open version where the formulas do not include the endpoints and an adaptive version, where the user is required to input the number of subintervals over which the appropriate closed family integrator will apply with the usual convergence parameters for each subinterval. This is useful where a large number of points are needed only in a small fraction of the entire domain.

Each routine takes as arguments:

- \( f \) integrand
- \( a \) starting point
- \( b \) ending point
- \( \epsilon_r \) relative error
- \( \epsilon_a \) absolute error
- \( n_{\text{min}} \) refinement level when to start checking for convergence (> 1)
- \( n_{\text{max}} \) maximum level of refinement

The adaptive routines take as an additional parameter, \( n_{\text{int}} \), the number of independent intervals to apply a closed family integrator of the same name.

Note that closed family level \( i \) uses \( 1 + 2^{*i} \) points.
Open family level \( i \) uses \( 1 + 3^{*i} \) points.

See Also:
NumericalQuadrature (NUMQUAD)

Exports:
aromberg asimpson atrapezoidal romberg rombergo
simpson simpsono trapezoidal trapezoidalo

-- package NUMQUAD NumericalQuadrature --
The routines come in three families and three flavors:

- **closed**: romberg, simpson, trapezoidal
- **open**: rombergo, simpsono, trapezoidalo
- **adaptive closed**: aromberg, asimpson, atrapezoidal

The \( S(i) \) for the trapezoidal family is the value of the integral using an equally spaced absicca trapezoidal rule for that level of refinement.

The \( S(i) \) for the simpson family is the value of the integral using an equally spaced absicca simpson rule for that level of refinement.

The \( S(i) \) for the romberg family is the estimate of the integral using an equally spaced absicca romberg method. For the \( i \)-th level, this is an appropriate combination of all the previous trapezoidal estimates so that the error term starts with the \( 2^*(i+1) \) power only.

The three families come in a closed version, where the formulas include the endpoints, an open version where the formulas do not include the endpoints and an adaptive version, where the user is required to input the number of subintervals over which the appropriate closed family integrator will apply with the usual convergence parameters for each subinterval. This is useful where a large number of points are needed only in a small fraction of the entire domain.

Each routine takes as arguments:

- \( f \) integrand
- \( a \) starting point
- \( b \) ending point
- \( \text{eps}_r \) relative error
- \( \text{eps}_a \) absolute error
- \( n_{\min} \) refinement level when to start checking for convergence (> 1)
- \( n_{\max} \) maximum level of refinement

The adaptive routines take as an additional parameter, \( n_{\text{int}} \), the number of independent intervals to apply a closed family integrator of the same name.

Notes:

- Closed family level \( i \) uses \( \text{spad}(1 + 2^*i) \) points.
- Open family level \( i \) uses \( \text{spad}(1 + 3^*i) \) points.

NumericalQuadrature(): Exports == Implementation where
\textbf{Exports} with

- \texttt{aromberg} : \((F \to F,F,F,F,I,I,I) \to \text{TrapAns}\)
  + uses the adaptive romberg method to numerically integrate function
  + \texttt{\textbackslash spad}(fn) over the closed interval from \texttt{\textbackslash spad}(a) to \texttt{\textbackslash spad}(b),
  + with relative accuracy \texttt{\textbackslash spad}(\texttt{epsrel}) and absolute accuracy
  + \texttt{\textbackslash spad}(\texttt{epsabs}), with the refinement levels for convergence checking
  + vary from \texttt{\textbackslash spad}(\texttt{nmin}) to \texttt{\textbackslash spad}(\texttt{nmax}), and where \texttt{\textbackslash spad}(\texttt{nint})
  + is the number of independent intervals to apply the integrator.
  + The value returned is a record containing the value of the integral,
  + the estimate of the error in the computation, the total number of
  + function evaluations, and either a boolean value which is true if
  + the integral was computed within the user specified error criterion.
  + See \texttt{\textbackslash spadtype\{NumericalQuadrature\}} for details.

- \texttt{asimpson} : \((F \to F,F,F,F,I,I,I) \to \text{TrapAns}\)
  + uses the adaptive simpson method to numerically integrate function \texttt{\textbackslash spad}(fn)
  + over the closed interval from \texttt{\textbackslash spad}(a) to \texttt{\textbackslash spad}(b), with relative
  + accuracy \texttt{\textbackslash spad}(\texttt{epsrel}) and absolute accuracy \texttt{\textbackslash spad}(\texttt{epsabs}), with the
  + refinement levels for convergence checking vary from \texttt{\textbackslash spad}(\texttt{nmin})
  + to \texttt{\textbackslash spad}(\texttt{nmax}), and where \texttt{\textbackslash spad}(\texttt{nint}) is the number of independent
  + intervals to apply the integrator. The value returned is a record
  + containing the value of the integral, the estimate of the error in
  + the computation, the total number of function evaluations, and
  + either a boolean value which is true if the integral was computed
  + within the user specified error criterion.
  + See \texttt{\textbackslash spadtype\{NumericalQuadrature\}} for details.

- \texttt{atrapezoidal} : \((F \to F,F,F,F,I,I,I) \to \text{TrapAns}\)
  + uses the adaptive trapezoidal method to numerically integrate function \texttt{\textbackslash spad}(fn)
  + over the closed interval from \texttt{\textbackslash spad}(a) to \texttt{\textbackslash spad}(b), with relative
  + accuracy \texttt{\textbackslash spad}(\texttt{epsrel}) and absolute accuracy \texttt{\textbackslash spad}(\texttt{epsabs}),
  + with the refinement levels for convergence checking vary from \texttt{\textbackslash spad}(\texttt{nmin})
  + to \texttt{\textbackslash spad}(\texttt{nmax}), and where \texttt{\textbackslash spad}(\texttt{nint}) is the number
  + of independent intervals to apply the integrator. The value returned
  + is a record containing the value of the integral, the estimate of
  + the error in the computation, the total number of function
  + evaluations, and either a boolean value which is true if
  + the integral was computed within the user specified error criterion.
  + See \texttt{\textbackslash spadtype\{NumericalQuadrature\}} for details.
romberg : (F -> F,F,F,F,F,I,I) -> TrapAns
  ++ romberg(fn,a,b,epsrel,epsabs,nmin,nmax) uses the romberg
  ++ method to numerically integrate function \spad{fn} over the closed
  ++ interval \spad{a} to \spad{b}, with relative accuracy \spad{epsrel}
  ++ and absolute accuracy \spad{epsabs}, with the refinement levels
  ++ for convergence checking vary from \spad{nmin} to \spad{nmax}.
  ++ The value returned is a record containing the value
  ++ of the integral, the estimate of the error in the computation, the
  ++ total number of function evaluations, and either a boolean value
  ++ which is true if the integral was computed within the user specified
  ++ error criterion. See \spadtype{NumericalQuadrature} for details.
simpson : (F -> F,F,F,F,F,I,I) -> TrapAns
  ++ simpson(fn,a,b,epsrel,epsabs,nmin,nmax) uses the simpson
  ++ method to numerically integrate function \spad{fn} over the closed
  ++ interval \spad{a} to \spad{b}, with
  ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
  ++ with the refinement levels for convergence checking vary from
  ++ \spad{nmin} to \spad{nmax}. The value returned
  ++ is a record containing the value of the integral, the estimate of
  ++ the error in the computation, the total number of function
  ++ evaluations, and either a boolean value which is true if
  ++ the integral was computed within the user specified error criterion.
  ++ See \spadtype{NumericalQuadrature} for details.
trapezoidal : (F -> F,F,F,F,F,I,I) -> TrapAns
  ++ trapezoidal(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
  ++ trapezoidal method to numerically integrate function \spad{fn} over
  ++ the closed interval \spad{a} to \spad{b}, with
  ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
  ++ with the refinement levels for convergence checking vary
  ++ from \spad{nmin} to \spad{nmax}. The value returned
  ++ is a record containing the value of the integral, the estimate of
  ++ the error in the computation, the total number of function
  ++ evaluations, and either a boolean value which is true if
  ++ the integral was computed within the user specified error criterion.
  ++ See \spadtype{NumericalQuadrature} for details.
rombergo : (F -> F,F,F,F,F,I,I) -> TrapAns
  ++ rombergo(fn,a,b,epsrel,epsabs,nmin,nmax) uses the romberg
  ++ method to numerically integrate function \spad{fn} over
  ++ the open interval from \spad{a} to \spad{b}, with
  ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
  ++ with the refinement levels for convergence checking vary from
  ++ \spad{nmin} to \spad{nmax}. The value returned
  ++ is a record containing the value of the integral, the estimate of
  ++ the error in the computation, the total number of function
  ++ evaluations, and either a boolean value which is true if
  ++ the integral was computed within the user specified error criterion.
  ++ See \spadtype{NumericalQuadrature} for details.
simpsono : (F -> F,F,F,F,F,I,I) -> TrapAns
  ++ simpsono(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
  ++ simpson method to numerically integrate function \spad{fn} over
  ++ the open interval from \spad{a} to \spad{b}, with
  ++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
  ++ with the refinement levels for convergence checking vary from
  ++ \spad{nmin} to \spad{nmax}. The value returned
  ++ is a record containing the value of the integral, the estimate of
  ++ the error in the computation, the total number of function
  ++ evaluations, and either a boolean value which is true if
  ++ the integral was computed within the user specified error criterion.
  ++ See \spadtype{NumericalQuadrature} for details.
++ the open interval from \spad{a} to \spad{b}, with
++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
++ with the refinement levels for convergence checking vary from
++ \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

trapezoidalo : (F \to F,F,F,F,F,I,I) \to TrapAns
++ trapezoidalo(fn,a,b,epsrel,epsabs,nmin,nmax) uses the
++ trapezoidal method to numerically integrate function \spad{fn}
++ over the open interval from \spad{a} to \spad{b}, with
++ relative accuracy \spad{epsrel} and absolute accuracy \spad{epsabs},
++ with the refinement levels for convergence checking vary from
++ \spad{nmin} to \spad{nmax}. The value returned
++ is a record containing the value of the integral, the estimate of
++ the error in the computation, the total number of function
++ evaluations, and either a boolean value which is true if
++ the integral was computed within the user specified error criterion.
++ See \spadtype{NumericalQuadrature} for details.

Implementation \Rightarrow add

trappclosed : (F \to F,F,F,F,I) \to F
trapopen : (F \to F,F,F,F,I) \to F
import OutputPackage

-- ---------------------------------

aromberg(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
  ans : TrapAns
  sum : F := 0.0
  err : F := 0.0
  pts : I := 1
  done : B := true
  hh : F := (b-a) / nint
  x1 : F := a
  x2 : F := a + hh
  io : L OFORM := [x1::E,x2::E]
  i : I
  for i in 1..nint repeat
    ans := romberg(func,x1,x2,epsrel,epsabs,nmin,nmax)
    if (not ans.success) then
      io.1 := x1::E
      io.2 := x2::E
      print blankSeparate cons("accuracy not reached in interval":E,io)
    sum := sum + ans.value
    err := err + abs(ans.error)
    pts := pts + ans.totalpts-1
    done := (done and ans.success)
\begin{verbatim}
x1  := x2  
x2  := x2 + hh
return( [sum, err, pts, done] )

---------------------------------------------------
asimpson(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
ans : TrapAns
sum : F := 0.0
err : F := 0.0
pts : I := 1
done : B := true
hh : F := (b-a) / nint
x1 : F := a
x2 : F := a + hh
io : L OFORM := [x1::E,x2::E]
i : I
for i in 1..nint repeat
   ans := simpson(func,x1,x2,epsrel,epsabs,nmin,nmax)
   if (not ans.success) then
      io.1 := x1::E
      io.2 := x2::E
      print blankSeparate cons("accuracy not reached in interval":E,io)
   sum := sum + ans.value
   err := err + abs(ans.error)
   pts := pts + ans.totalpts-1
   done := (done and ans.success)
x1 := x2
x2 := x2 + hh
return( [sum, err, pts, done] )

---------------------------------------------------
atrapezoidal(func,a,b,epsrel,epsabs,nmin,nmax,nint) ==
ans : TrapAns
sum : F := 0.0
err : F := 0.0
pts : I := 1
i : I
done : B := true
hh : F := (b-a) / nint
x1 : F := a
x2 : F := a + hh
io : L OFORM := [x1::E,x2::E]
for i in 1..nint repeat
   ans := trapezoidal(func,x1,x2,epsrel,epsabs,nmin,nmax)
   if (not ans.success) then
      io.1 := x1::E
      io.2 := x2::E
      print blankSeparate cons("accuracy not reached in interval":E,io)
\end{verbatim}
sum := sum + ans.value
err := err + abs(ans.error)
pts := pts + ans.totalpts-1
done := (done and ans.success)
x1 := x2
x2 := x2 + hh

return( [sum , err , pts , done] )

-----------------------------

romberg(func,a,b,epsrel,epsabs,nmin,nmax) ==
    length : F := (b-a)
delta : F := length
newsum : F := 0.5 * length * (func(a)+func(b))
newest : F := 0.0
oldsum : F := 0.0
oldest : F := 0.0
change : F := 0.0
qx1 : F := newsum
table : V F := new((nmax+1)::PI,0.0)
n : I := 1
pts : I := 1
four : I
j : I
i : I

if (nmin < 2) then
    output("romberg: nmin to small (nmin > 1) nmin = ",nmin::E)
    return([0.0,0.0,0,false])

if (nmax < nmin) then
    output("romberg: nmax < nmin : nmax = ",nmax::E)
    output(" nmin = ",nmin::E)
    return([0.0,0.0,0,false])

if (a = b) then
    output("romberg: integration limits are equal = ",a::E)
    return([0.0,0.0,1,true])

if (epsrel < 0.0) then
    output("romberg: eps_r < 0.0 eps_r = ",epsrel::E)
    return([0.0,0.0,0,false])

if (epsabs < 0.0) then
    output("romberg: eps_a < 0.0 eps_a = ",epsabs::E)
    return([0.0,0.0,0,false])

for n in 1..nmax repeat
    oldsum := newsum
    newsum := trapclosed(func,a,delta,oldsum,pts)
    newest := (4.0 * newsum - oldsum) / 3.0
    four := 4
    table(n) := newest
    for j in 2..n repeat
        i := n+1-j
        four := four * 4
table(i) := table(i+1) + (table(i+1)-table(i)) / (four-1)
if n > nmin then
  change := abs(table(1) - qx1)
  if change < abs(epsrel*qx1) then
    return( [table(1) , change , 2*pts+1 , true] )
  if change < epsabs then
    return( [table(1) , change , 2*pts+1 , true] )
oldsum := newsum
oldest := newest
delta := 0.5*delta
pts := 2*pts
qx1 := table(1)
return( [table(1) , 1.25*change , pts+1 ,false] )

---------------------------------------------------
simpson(func,a,b,epsrel,epsabs,nmin,nmax) ==
  length : F := (b-a)
delta : F := length
  newsum : F := 0.5*(b-a)*(func(a)+func(b))
  newest : F := 0.0
  oldsum : F := 0.0
  oldest : F := 0.0
  change : F := 0.0
  n : I := 1
  pts : I := 1
  if (nmin < 2) then
    output("simpson: nmin to small (nmin > 1) nmin = ",nmin::E)
    return([0.0,0.0,0,false])
  if (nmax < nmin) then
    output("simpson: nmax < nmin : nmax = ",nmax::E)
    output(""nmin = ",nmin::E)
    return([0.0,0.0,0,false])
  if (a = b) then
    output("simpson: integration limits are equal = ",a::E)
    return([0.0,0.0,1,true])
  if (epsrel < 0.0) then
    output("simpson: eps_r < 0.0 : eps_r = ",epsrel::E)
    return([0.0,0.0,0,false])
  if (epsabs < 0.0) then
    output("simpson: eps_a < 0.0 : eps_a = ",epsabs::E)
    return([0.0,0.0,0,false])
  for n in 1..nmax repeat
    oldsum := newsum
    newsum := trapclosed(func,a,delta,oldsum,pts)
    newest := (4.0 * newsum - oldsum) / 3.0
    if n > nmin then
      change := abs(newest-oldest)
      if change < abs(epsrel*oldest) then
        return( [newest , 1.25*change , 2*pts+1 , true] )
if change < epsabs then
    return( [newest, 1.25*change, 2*pts+1, true] )
oldsum := newsum
oldest := newest
delta := 0.5*delta
pts := 2*pts
return( [newest, 1.25*change, pts+1, false] )

---------------------------------------------------

trapezoidal(func,a,b,epsrel,epsabs,nmin,nmax) ==
length := F := (b-a)
delta := F := length
newsum := F := 0.5*(b-a)*(func(a)+func(b))
change := F := 0.0
oldsum := F
n := I := 1
pts := I := 1
if (nmin < 2) then
    output("trapezoidal: nmin to small (nmin > 1) nmin = ",nmin::E)
    return([0.0,0.0,0,0,false])
if (nmax < nmin) then
    output("trapezoidal: nmax < nmin : nmax = ",nmax::E)
    output(" nmin = ",nmin::E)
    return([0.0,0.0,0,0,false])
if (a = b) then
    output("trapezoidal: integration limits are equal = ",a::E)
    return([0.0,0.0,1,true])
if (epsrel < 0.0) then
    output("trapezoidal: eps_r < 0.0 : eps_r = ",epsrel::E)
    return([0.0,0.0,0,0,false])
if (epsabs < 0.0) then
    output("trapezoidal: eps_a < 0.0 : eps_a = ",epsabs::E)
    return([0.0,0.0,0,0,false])
for n in 1..nmax repeat
    oldsum := newsum
    newsum := trapclosed(func,a,delta,oldsum,pts)
    if n > nmin then
        change := abs(newsum-oldsum)
        if change < abs(epsrel*oldsum) then
            return( [newsum, 1.25*change, 2*pts+1, true] )
        if change < epsabs then
            return( [newsum, 1.25*change, 2*pts+1, true] )
        delta := 0.5*delta
        pts := 2*pts
        return( [newsum, 1.25*change, pts+1, false] )

---------------------------------------------------

rombergo(func,a,b,epsrel,epsabs,nmin,nmax) ==

simpsono(func,a,b,epsrel,epsabs,nmin,nmax) ==
    length : F := (b-a)
    delta  : F := length / 3.0
    newsum : F := length * func( 0.5*(a+b) )
    newest : F := 0.0
    oldsum : F := 0.0
    oldest : F := 0.0
    change : F := 0.0
    qx1   : F := newsum
    table  : V F := new((nmax+1)::PI,0.0)
    four   : I
    j      : I
    i      : I
    n      : I := 1
    pts    : I := 1
    for n in 1..nmax repeat
        oldsum := newsum
        newsum := trapopen(func,a,delta,oldsum,pts)
        newest := (9.0 * newsum - oldsum) / 8.0
        table(n) := newest
        nine := 9
        output(newest::E)
        for j in 2..n repeat
            i := n+1-j
            nine := nine * 9
            table(i) := table(i+1) + (table(i+1)-table(i)) / (nine-1)
        if n > nmin then
            change := abs(table(1) - qx1)
            if change < abs(epsrel*qx1) then
                return( [table(1) , 1.5*change , 3*pts , true] )
            if change < epsabs then
                return( [table(1) , 1.5*change , 3*pts , true] )
        output(table::E)
        oldsum := newsum
        oldest := newest
        delta := delta / 3.0
        pts := 3*pts
        qx1 := table(1)
    return( [table(1) , 1.5*change , pts ,false] )

---------------------------------------------------

simpsono(func,a,b,epsrel,epsabs,nmin,nmax) ==
    length : F := (b-a)
    delta  : F := length / 3.0
    newsum : F := length * func( 0.5*(a+b) )
    newest : F := 0.0
    oldsum : F := 0.0
    oldest : F := 0.0
    change : F := 0.0
    n      : I := 1

length : F := (b-a)
delta  : F := length / 3.0
newsum : F := length * func( 0.5*(a+b) )
newest : F := 0.0
oldsum : F := 0.0
oldest : F := 0.0
change : F := 0.0
qx1   : F := newsum
table  : V F := new((nmax+1)::PI,0.0)
four   : I
j      : I
i      : I
n      : I := 1
pts    : I := 1
for n in 1..nmax repeat
    oldsum := newsum
    newsum := trapopen(func,a,delta,oldsum,pts)
    newest := (9.0 * newsum - oldsum) / 8.0
    table(n) := newest
    nine := 9
    output(newest::E)
    for j in 2..n repeat
        i := n+1-j
        nine := nine * 9
        table(i) := table(i+1) + (table(i+1)-table(i)) / (nine-1)
    if n > nmin then
        change := abs(table(1) - qx1)
        if change < abs(epsrel*qx1) then
            return( [table(1) , 1.5*change , 3*pts , true] )
        if change < epsabs then
            return( [table(1) , 1.5*change , 3*pts , true] )
    output(table::E)
    oldsum := newsum
    oldest := newest
    delta := delta / 3.0
    pts := 3*pts
    qx1 := table(1)
    return( [table(1) , 1.5*change , pts ,false] )

---------------------------------------------------
pts : I := 1
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapopen(func,a,delta,oldsum,pts)
  newest := (9.0 * newsum - oldsum) / 8.0
  output(newest::E)
  if n > nmin then
    change := abs(newest - oldest)
    if change < abs(epsrel*oldest) then
      return([newest, 1.5*change, 3*pts, true])
    if change < epsabs then
      return([newest, 1.5*change, 3*pts, true])
  oldsum := newsum
  oldest := newest
  delta := delta / 3.0
  pts := 3*pts
  return([newest, 1.5*change, pts,false])

---------------------------------------------------
trapezoidal(func,a,b,epsrel,epsabs,nmin,nmax) ==
  length : F := (b-a)
  delta : F := length/3.0
  newsum : F := length*func( 0.5*(a+b) )
  change : F := 0.0
  pts : I := 1
  oldsum : F
  n : I
for n in 1..nmax repeat
  oldsum := newsum
  newsum := trapopen(func,a,delta,oldsum,pts)
  output(newsum::E)
  if n > nmin then
    change := abs(newsum-oldsum)
    if change < abs(epsrel*oldsum) then
      return([newsum, 1.5*change, 3*pts, true])
    if change < epsabs then
      return([newsum, 1.5*change, 3*pts, true])
  delta := delta / 3.0
  pts := 3*pts
  return([newsum, 1.5*change, pts,false])

---------------------------------------------------
trapclosed(func,start,h,oldsum,numpoints) ==
  x : F := start + 0.5*h
  sum : F := 0.0
  i : I
for i in 1..numpoints repeat
  sum := sum + func(x)
\[ x := x + h \]
\[ \text{return}(0.5 \times (\text{oldsum} + \text{sum} \times h)) \]

---

\[
\text{trapopen(func, start, del, oldsum, numpoints)} ==
\]
\[
ddel : F := 2.0 \times del \\
x : F := start + 0.5 \times del \\
sum : F := 0.0 \\
i : I \\
\text{for i in 1..numpoints repeat} \\
\quad \text{sum} := \text{sum} + \text{func}(x) \\
\quad x := x + ddel \\
\quad \text{sum} := \text{sum} + \text{func}(x) \\
\quad x := x + del \\
\text{return}(\frac{\text{oldsum}}{3.0} + \text{sum} \times del) \]

---

-- NUMQUAD.dotabb --
"NUMQUAD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMQUAD"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"NUMQUAD" -> "IVECTOR"

---

package NCEP NumericComplexEigenPackage

--- NumericComplexEigenPackage.input ---

)set break resume
)sys rm -f NumericComplexEigenPackage.output
)spool NumericComplexEigenPackage.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show NumericComplexEigenPackage
--R
--R NumericComplexEigenPackage(Par: Join(Field,OrderedRing)) is a package constructor
--R Abbreviation for NumericComplexEigenPackage is NCEP
--R This constructor is exposed in this frame.
NCEP --R Issue )edit bookvol10.4.pamphlet to see algebra source code for NCEP --R
--R-------------------------------------------------- Operations -------------------------------
--R characteristicPolynomial : Matrix(Complex(Fraction(Integer))) -> Polynomial(Complex(Fraction(Integer)))
--R characteristicPolynomial : (Matrix(Complex(Fraction(Integer))),Symbol) -> Polynomial(Complex(Fraction(Integer)))
--R complexEigenvalues : (Matrix(Complex(Fraction(Integer))),Par) -> List(Complex(Par))
--R complexEigenvectors : (Matrix(Complex(Fraction(Integer))),Par) -> List(Record(outval: Complex(Par),outmult: Integer,outvect: List(Matrix(Complex(Par))))
--E 1

)spool
)lisp (bye)

- NumericComplexEigenPackage.help -

====================================================================
NumericComplexEigenPackage examples
====================================================================

This package computes explicitly eigenvalues and eigenvectors of matrices with entries over the complex rational numbers. The results are expressed either as complex floating numbers or as complex rational numbers depending on the type of the precision parameter.

See Also:
  o )show NumericComplexEigenPackage

-----

NumericComplexEigenPackage (NCEP)
Exports:

characteristicPolynomial complexEigenvalues complexEigenvectors

package NCEP NumericComplexEigenPackage

)abbrev package NCEP NumericComplexEigenPackage
++ Author: P. Gianni
++ Date Created: Summer 1990
++ Date Last Updated: Spring 1991
++ Description:
++ This package computes explicitly eigenvalues and eigenvectors of
++ matrices with entries over the complex rational numbers.
++ The results are expressed either as complex floating numbers or as
++ complex rational numbers depending on the type of the precision parameter.

NumericComplexEigenPackage(Par) : C == T
where
Par : Join(Field,OrderedRing) -- Float or RationalNumber
SE ==> Symbol()
RN ==> Fraction Integer
I ==> Integer
NF ==> Float
CF ==> Complex Float
GRN ==> Complex RN
GI ==> Complex Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
MRN ==> Matrix RN
MCF ==> Matrix CF
MGRN ==> Matrix GRN
MCPar ==> Matrix Complex Par
SUPGRN ==> SparseUnivariatePolynomial GRN
outForm ==> Record(outval:Complex Par, outmult:Integer, outvect:List MCPar)

C == with
characteristicPolynomial : MGRN -> Polynomial GRN
++ characteristicPolynomial(m) returns the characteristic polynomial
++ of the matrix m expressed as polynomial
++ over complex rationals with a new symbol as variable.
-- while the function in EigenPackage returns Fraction P GRN.
characteristicPolynomial : (MGRN,SE) -> Polynomial GRN
++ characteristicPolynomial(m,x) returns the characteristic polynomial
++ of the matrix m expressed as polynomial
++ over Complex Rationals with variable x.
-- while the function in EigenPackage returns Fraction P GRN.
complexEigenvalues : (MGRN,Par) -> List Complex Par
++ complexEigenvalues(m,eps) computes the eigenvalues of the matrix
++ m to precision eps. The eigenvalues are expressed as complex
++ floats or complex rational numbers depending on the type of
++ eps (float or rational).
complexEigenvectors : (MGRN,Par) -> List(outForm)
++ complexEigenvectors(m,eps) returns a list of
++ records each one containing
++ a complex eigenvalue, its algebraic multiplicity, and a list of
++ associated eigenvectors. All these results
++ are computed to precision eps and are expressed as complex floats
++ or complex rational numbers depending on the type of
++ eps (float or rational).
T == add

import InnerNumericEigenPackage(GRN,Complex Par,Par)

characteristicPolynomial(m:MGRN) : Polynomial GRN ==
x:SE:=new($SE
multivariate(charpol m, x)

characteristicPolynomial(A:MGRN,x:SE):Polynomial GRN ==
multivariate(charpol A, x)

complexEigenvalues(m:MGRN,eps:Par) : List Complex Par ==
solve1(charpol m, eps)

complexEigenvectors(m:MGRN,eps:Par) :List outForm ==
innerEigenvectors(m,eps,factor$ComplexFactorization(RN,SUPGRN))

package NCNTFRAC NumericContinuedFraction

— NumericContinuedFraction.input —
PACKAGE NCNTFRAC NUMERICCONTINUEDFRACTION

)sys rm -f NumericContinuedFraction.output
)spool NumericContinuedFraction.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericContinuedFraction
--R
--R NumericContinuedFraction(F: FloatingPointSystem) is a package constructor
--R Abbreviation for NumericContinuedFraction is NCNTFRAC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NCNTFRAC
--R
--R-------------------------------- Operations --------------------------------
--R continuedFraction : F -> ContinuedFraction(Integer)
--R
--E 1

)spool
)lisp (bye)

----------

— NumericContinuedFraction.help —

====================================================================
NumericContinuedFraction examples
====================================================================

NumericContinuedFraction provides functions for converting floating point numbers to continued fractions.

See Also:
  o )show NumericContinuedFraction

----------
NumericContinuedFraction (NCNTFRAC)

Exports:
continuedFraction

--- package NCNTFRAC NumericContinuedFraction ---

)abbrev package NCNTFRAC NumericContinuedFraction
++ Author: Clifton J. Williamson
++ Date Created: 12 April 1990
++ Description:
++ \spadtype{NumericContinuedFraction} provides functions
++ for converting floating point numbers to continued fractions.

NumericContinuedFraction(F): Exports == Implementation where
  F : FloatingPointSystem
  CFC ==> ContinuedFraction Integer
  I ==> Integer
  ST ==> Stream I

Exports ==> with
  continuedFraction: F -> CFC
  ++ continuedFraction(f) converts the floating point number
  ++ \spad{f} to a reduced continued fraction.

Implementation ==> add

  cfc: F -> ST
  cfc(a) == delay
    aa := wholePart a
    zero?(b := a - (aa :: F)) => concat(aa,empty()$ST)
    concat(aa,cfc inv b)

  continuedFraction a ==
    aa := wholePart a
    zero?(b := a - (aa :: F)) =>
      reducedContinuedFraction(aa,empty()$ST)
if negative? b then (aa := aa - 1; b := b + 1)
reducedContinuedFraction(aa,cfc inv b)

---

NCNTFRAC.dotabb

"NCNTFRAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NCNTFRAC"]
"FIELD"  [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"NCNTFRAC" -> "FIELD"
"NCNTFRAC" -> "RADCAT"

---

package NREP NumericRealEigenPackage

--- NumericRealEigenPackage.input ---

)set break resume
)sys rm -f NumericRealEigenPackage.output
)spool NumericRealEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericRealEigenPackage
--R
--R NumericRealEigenPackage(Par: Join(Field,OrderedRing)) is a package constructor
--R Abbreviation for NumericRealEigenPackage is NREP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NREP
--R
--R---------------------------------- Operations ----------------------------------
--R characteristicPolynomial : Matrix(Fraction(Integer)) -> Polynomial(Fraction(Integer))
--R characteristicPolynomial : (Matrix(Fraction(Integer)),Symbol) -> Polynomial(Fraction(Integer))
--R realEigenvalues : (Matrix(Fraction(Integer)),Par) -> List(Par)
--R realEigenvectors : (Matrix(Fraction(Integer)),Par) -> List(Record(outval: Par,outmult: Integer,outvect:...
--R
--E 1

)spool
)lisp (bye)
Chapter 15. Chapter N

---

--- NumericRealEigenPackage.help ---

====================================================================
NumericRealEigenPackage examples
====================================================================

This package computes explicitly eigenvalues and eigenvectors of matrices with entries over the Rational Numbers. The results are expressed as floating numbers or as rational numbers depending on the type of the parameter Par.

See Also:
- )show NumericRealEigenPackage

---

NumericRealEigenPackage (NREP)

Exports:
characteristicPolynomial realEigenvalues realEigenvectors

--- package NREP NumericRealEigenPackage ---

)abbrev package NREP NumericRealEigenPackage
+ Author:P. Gianni
+ Date Created:Summer 1990
+ Date Last Updated:Spring 1991
+ Description:
+ This package computes explicitly eigenvalues and eigenvectors of matrices with entries over the Rational Numbers.
+ The results are expressed as floating numbers or as rational numbers
 NumericRealEigenPackage(Par) : C == T

where

Par : Join(Field,OrderedRing) -- Float or RationalNumber

SE ==> Symbol()
RN ==> Fraction Integer
I ==> Integer
NF ==> Float
CF ==> Complex Float
GRN ==> Complex RN
GI ==> Complex Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
MRN ==> Matrix RN

MPar ==> Matrix Par
outForm ==> Record(outval:Par,outmult:Integer,outvect:List MPar)

C == with

characteristicPolynomial : MRN -> Polynomial RN
++ characteristicPolynomial(m) returns the characteristic polynomial
++ of the matrix m expressed as polynomial
++ over RN with a new symbol as variable.
-- while the function in EigenPackage returns Fraction P RN.

characteristicPolynomial : (MRN,SE) -> Polynomial RN
++ characteristicPolynomial(m,x) returns the characteristic polynomial
++ of the matrix m expressed as polynomial
++ over RN with variable x.
-- while the function in EigenPackage returns
++ Fraction P RN.

realEigenvalues : (MRN,Par) -> List Par
++ realEigenvalues(m,eps) computes the eigenvalues of the matrix
++ m to precision eps. The eigenvalues are expressed as floats or
++ rational numbers depending on the type of eps (float or rational).

realEigenvectors : (MRN,Par) -> List(outForm)
++ realEigenvectors(m,eps) returns a list of
++ records each one containing
++ a real eigenvalue, its algebraic multiplicity, and a list of
++ associated eigenvectors. All these results
++ are computed to precision eps as floats or rational
++ numbers depending on the type of eps.

T == add

import InnerNumericEigenPackage(RN, Par, Par)

classicntPolynomial(m:MRN) : Polynomial RN ==
x:SE := new()$SE
multivariate(charpol(m),x)

---- characteristic polynomial of a matrix A ----
characteristicPolynomial(A:MRN,x:SE):Polynomial RN ==
multivariate(charpol(A),x)

realEigenvalues(m:MRN,eps:Par) : List Par ==
solve1(charpol m, eps)

realEigenvectors(m:MRN,eps:Par) :List outForm ==
innerEigenvectors(m,eps,factor$GenUFactorize(RN))

package NUMTUBE NumericTubePlot

— NumericTubePlot.input —

)set break resume
)sys rm -f NumericTubePlot.output
)spool NumericTubePlot.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show NumericTubePlot
--R
--R NumericTubePlot(Curve: PlottableSpaceCurveCategory) is a package constructor
--R Abbreviation for NumericTubePlot is NUMTUBE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for NUMTUBE
--R
--R------------------------------------------------------ Operations ...........................
--R tube : (Curve,DoubleFloat,Integer) -> TubePlot(Curve)
--R
--E 1

)spool
)lisp (bye)

——

— NumericTubePlot.help —

====================================================================
NumericTubePlot examples
====================================================================

Package for constructing tubes around 3-dimensional parametric curves.

See Also:
o )show NumericTubePlot

——

NumericTubePlot (NUMTUBE)

Exports:
tube

—— package NUMTUBE NumericTubePlot ——

)abbrev package NUMTUBE NumericTubePlot
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Description:
++ Package for constructing tubes around 3-dimensional parametric curves.

NumericTubePlot(Curve):Exports == Implementation where
Curve : PlottableSpaceCurveCategory
B ==> Boolean
I ==> Integer
SF ==> DoubleFloat
L ==> List
S ==> String
SEG ==> Segment
Pt ==> Point SF
TUBE ==> TubePlot Curve
Triad ==> Record(tang:Pt,norm:Pt,bin:Pt)

Exports ==> with
tube: (Curve,SF,I) -> TUBE
++ tube(c,r,n) creates a tube of radius r around the curve c.

Implementation ==> add
import TubePlotTools

LINMAX := convert(0.995)@SF
XHAT := point(1,0,0,0)
YHAT := point(0,1,0,0)
PREVO := point(1,1,0,0)
PREV := PREVO

colinearity: (Pt,Pt) -> SF
colinearity(x,y) == dot(x,y)**2/(dot(x,x) * dot(y,y))

orthog: (Pt,Pt) -> Pt
orthog(x,y) ==
  if colinearity(x,y) > LINMAX then y := PREV
  if colinearity(x,y) > LINMAX then
    y := (colinearity(x,XHAT) < LINMAX => XHAT; YHAT)
  a := -dot(x,y)/dot(x,x)
  PREV := a*x + y

poTriad:(Pt,Pt,Pt) -> Triad
poTriad(pl,po,pr) ==
  -- use divided difference for t.
  t := unitVector(pr - pl)
  -- compute n as orthogonal to t in plane containing po.
  pol := pl - po
  n := unitVector orthog(t,pol)
  [t,n,cross(t,n)]

curveTriads: L Pt -> L Triad
curveTriads l ==
(k := #l) < 2 => error "Need at least 2 points to specify a curve"

PREV := PREVO
k = 2 =>
t := unitVector(second l - first l)
n := unitVector(t - XHAT)
b := cross(t,n)
triad : Triad := [t,n,b]
[triad,triad]

-- compute interior triads using divided differences
midtriads : L Triad :=
[poTriad(pl,po,pr) for pl in l for po in rest l for pr in rest rest l]

-- compute first triad using a forward difference
x := first midtriads
t := unitVector(second l - first l)
n := unitVector orthog(t,x.norm)
begtriad : Triad := [t,n,cross(t,n)]

-- compute last triad using a backward difference
x := last midtriads
-- efficiency!!
t := unitVector(l.k - l.(k-1))
n := unitVector orthog(t,x.norm)
endtriad : Triad := [t,n,cross(t,n)]
concat(begtriad,concat(midtriads,endtriad))

curveLoops: (L Pt,SF,I) -> L L Pt
curveLoops(pts,r,nn) ==
triads := curveTriads pts
cosSin := cosSinInfo nn
loops : L L Pt := nil()
for pt in pts for triad in triads repeat
  n := triad.norm; b := triad.bin
  loops := concat(loopPoints(pt,n,b,r,cosSin),loops)
reverse_! loops

tube(curve,r,n) ==
n < 3 => error "tube: n should be at least 3"
brans := listBranches curve
loops : L L Pt := nil()
for bran in brans repeat
  loops := concat(loops,curveLoops(bran,r,n))
tube(curve,loops,false)

 NUMTUBE.dotabb 

"NUMTUBE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=NUMTUBE"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"NUMTUBE" -> "FIELD"
"NUMTUBE" -> "RADCAT"
"NUMTUBE" -> "FLAGG"
"NUMTUBE" -> "FLAGG-"
Chapter 16

Chapter O

package OCTCT2 OctonionCategoryFunctions2

-- OctonionCategoryFunctions2.input --

)set break resume
)sys rm -f OctonionCategoryFunctions2.output
)spool OctonionCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OctonionCategoryFunctions2
--R
--R OctonionCategoryFunctions2(OR: OctonionCategory(R),R: CommutativeRing,OS: OctonionCategory(S),S: CommutativeRing)
--R Abbreviation for OctonionCategoryFunctions2 is OCTCT2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for OCTCT2
--R
--R---------------------------------- Operations ----------------------------------
--R map : ((R -> S),OR) -> OS
--R
--E 1

)spool
)lisp (bye)

-----

-- OctonionCategoryFunctions2.help --

3179
OctonionCategoryFunctions2 implements functions between two octonion domains defined over different rings. The function map is used to coerce between octonion types.

See Also:
- )show OctonionCategoryFunctions2

---

OctonionCategoryFunctions2 (OCTCT2)

Exports:
- map

---

)abbrev package OCTCT2 OctonionCategoryFunctions2
++ Author: Johannes Grabmeier
++ Date Created: 10 September 1990
++ Date Last Updated: 10 September 1990
++ Description:
++ OctonionCategoryFunctions2 implements functions between
++ two octonion domains defined over different rings.
++ The function map is used to coerce between octonion types.

OctonionCategoryFunctions2(OR,R,OS,S) : Exports ==
Implementation where
- R : CommutativeRing
- S : CommutativeRing
OR : OctonionCategory R
OS : OctonionCategory S
Exports == with
  map: (R -> S, OR) -> OS
  ++ map(f,u) maps f onto the component parts of the octonion
  ++ u.
Implementation == add
  map(fn : R -> S, u : OR): OS ==
  octon(fn real u, fn imagi u, fn imagj u, fn imagk u,_,
  fn imagE u, fn imagI u, fn imagJ u, fn imagK u)$OS

— OCTCT2.dotabb —
"OCTCT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OCTCT2"]
"OC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OC"]
"OCTCT2" -> "OC"

package ODEINT ODEIntegration

— ODEIntegration.input —

)set break resume
)sys rm -f ODEIntegration.output
)spool ODEIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ODEIntegration
--R
--R ODEIntegration(R: Join(OrderedSet,EuclideanDomain,RetractableTo(Integer),LinearlyExplicitRingOver(Integer),Commutative)
--R Abbreviation for ODEIntegration is ODEINT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODEINT
--R
--R------------------------------- Operations --------------------------------
--R diff : Symbol -> (F -> F) expint : (F,Symbol) -> F
--R int : (F,Symbol) -> F
--R
ODEIntegration provides an interface to the integrator. This package is intended for use by the differential equations solver but not at top-level.

See Also:
- )show ODEIntegration

ODEIntegration (ODEINT)

Exports:
- diff  expint  int

--- package ODEINT ODEIntegration ---

)abbrev package ODEINT ODEIntegration
++ Author: Manuel Bronstein
++ Date Created: 4 November 1991
++ Date Last Updated: 2 February 1994
++ Description:
++ \spadtype{ODEIntegration} provides an interface to the integrator.
++ This package is intended for use
++ by the differential equations solver but not at top-level.

ODEIntegration(R, F): Exports == Implementation where
R: Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
   LinearlyExplicitRingOver Integer, CharacteristicZero)
F: Join(AlgebraicallyClosedFunctionSpace R, TranscendentalFunctionCategory,
   PrimitiveFunctionCategory)

Q == Fraction Integer
UQ == Union(Q, "failed")
SY == Symbol
K == Kernel F
P ==> SparseMultivariatePolynomial(R, K)
REC ==> Record(coef:Q, logand:F)

Exports ==> with
  int : (F, SY) -> F
  ++ int(f, x) returns the integral of f with respect to x.
  expint: (F, SY) -> F
  ++ expint(f, x) returns e"{the integral of f with respect to x}.
  diff : SY -> (F -> F)
  ++ diff(x) returns the derivation with respect to x.

Implementation ==> add
import FunctionSpaceIntegration(R, F)
import ElementaryFunctionStructurePackage(R, F)

isQ : List F -> UQ
isQlog: F -> Union(REC, "failed")
mkprod: List REC -> F

diff x == (f1:F):F ++-> differentiate(f1, x)

-- This is the integration function to be used for quadratures
int(f, x) ==
  (u := integrate(f, x)) case F => u::F
  first(u::List(F))

-- mkprod([q1, f1],...,[qn,fn]) returns */(fi^qi) but groups the
-- qi having the same denominator together
mkprod l ==
  empty? l => 1
  rec := first l
  d := denom(rec.coef)
  ll := select((z1:REC):Boolean ++-> denom(z1.coef) = d, l)
nthRoot(*/[r.logand ** numer(r.coef) for r in ll], d) *
mkprod setDifference(l, ll)

-- computes \( \exp(int(f,x)) \) in a non-naive way
expint(f, x) ==
a := int(f, x)
(u := validExponential(tower a, a, x)) case F => u::F
da := denom a
l :=
(v := isPlus(na := numer a)) case List(P) => v::List(P)
[na]
exponent::F := 0
lrec::List(REC) := empty()
for term in l repeat
  if (w := isQlog(term / da)) case REC then
    lrec := concat(w::REC, lrec)
  else
    exponent := exponent + term
  mkprod(lrec) * exp(exponent / da)

-- checks if all the elements of l are rational numbers, returns their product
isQ l ==
prod::Q := 1
for x in l repeat
  (u := retractIfCan(x)@UQ) case "failed" => return "failed"
  prod := prod * u::Q
prod

-- checks if a non-sum expr is of the form \( c \cdot \log(g) \) for a rational number \( c \)
isQlog f ==
is?(f, "log"::SY) => [1, first argument(retract(f)@K)]
(v := istimes f) case List(F) and (#(l := v::List(F)) <= 3) =>
l := reverse_! sort_! l
is?(first l, "log"::SY) and ((u := isQ rest l) case Q) =>
  [u::Q, first argument(retract(first(l))@K)]
"failed"
"failed"
package ODETOOLS ODETools

--- ODETools.input ---

)set break resume
)set sys rm -f ODETools.output
)spool ODETools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ODETools
--R
--R ODETools(F: Field,LODO: LinearOrdinaryDifferentialOperatorCategory(F)) is a package constructor
--R Abbreviation for ODETools is ODETOOLS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODETOOLS
--R
--R----------------------------------- Operations -----------------------------------
--R particularSolution : (LODO,F,List(F),(F -> F)) -> Union(F,"failed")
--R variationOfParameters : (LODO,F,List(F)) -> Union(Vector(F),"failed")
--R wronskianMatrix : List(F) -> Matrix(F)
--R wronskianMatrix : (List(F),NonNegativeInteger) -> Matrix(F)
--R
--E 1

)spool
)lisp (bye)

-----

--- ODETools.help ---

====================================================================
ODETools examples
====================================================================

ODETools provides tools for the linear ODE solver.

See Also:
o )show ODETools

-----
ODETools (ODETOOLS)

Exports:
  particularSolution variationOfParameters wronskianMatrix

— package ODETOOLS ODETools —

)abbrev package ODETOOLS ODETools
++ Author: Manuel Bronstein
++ Date Created: 20 March 1991
++ Date Last Updated: 2 February 1994
++ Description:
++ \spad{ODETools} provides tools for the linear ODE solver.

ODETools(F, LODO): Exports == Implementation where
  N ==> NonNegativeInteger
  L ==> List F
  V ==> Vector F
  M ==> Matrix F
  F: Field
  LODO: LinearOrdinaryDifferentialOperatorCategory F

Exports ==> with
  wronskianMatrix: L -> M
    ++ wronskianMatrix([f1,...,fn]) returns the \spad{n x n} matrix m
    ++ whose i^th row is \spad{[f1^(i-1),...,fn^(i-1)]}.
  wronskianMatrix: (L, N) -> M
    ++ wronskianMatrix([f1,...,fn], q, D) returns the \spad{q x n} matrix m
    ++ whose i^th row is \spad{[f1^(i-1),...,fn^(i-1)]}.
  variationOfParameters: (LODO, F, L) -> Union(V, "failed")
    ++ variationOfParameters(op, g, [f1,...,fm])
    ++ returns \spad{[u1,...,um]} such that a particular solution of the
    ++ equation \spad{op y = g} is \spad{f1 int(u1) + ... + fm int(um)}
    ++ where \spad{[f1,...,fm]} are linearly independent and \spad{op(fi)=0}.
    ++ The value "failed" is returned if \spad{m < n} and no particular
    ++ solution is found.
particularSolution: (LODO, F, L, F -> F) -> Union(F, "failed")
  ++ particularSolution(op, g, [f1,...,fm], I) returns a particular
  ++ solution h of the equation \( \text{spad}(\text{op } y = g) \) where \( \text{spad}([f1,...,fm]) \)
  ++ are linearly independent and \( \text{spad}(\text{op}(f_i)=0) \).
  ++ The value "failed" is returned if no particular solution is found.
  ++ Note that the method of variations of parameters is used.

Implementation ==> add
  import LinearSystemMatrixPackage(F, V, V, M)
  diff := D()$LODO
  wronskianMatrix l == wronskianMatrix(l, #l)
  wronskianMatrix(l, q) ==
    v:V := vector l
    m:M := zero(q, #v)
    for i in minRowIndex m .. maxRowIndex m repeat
      setRow_!(m, i, v)
      v := map_!(f1:F):F +-> diff f1, v)
    m
  variationOfParameters(op, g, b) ==
    empty? b => "failed"
    v:V := new(n := degree op, 0)
    qsetelt_!(v, maxIndex v, g / leadingCoefficient op)
    particularSolution(wronskianMatrix(b, n), v)
  particularSolution(op, g, b, integration) ==
    zero? g => 0
    (sol := variationOfParameters(op, g, b)) case "failed" => "failed"
    ans:F := 0
    for f in b for i in minIndex(s := sol::V) .. repeat
      ans := ans + integration(qelt(s, i)) * f
    ans

| ODETOOLS.dotabb |
| ODETOOLS | IVECTOR |
| ODETOOLS | IVECTOR |

"ODETOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODETOOLS"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ODETOOLS" -> "IVECTOR"
package ARRAY12 OneDimensionalArrayFunctions2

--- OneDimensionalArrayFunctions2.input ---

)set break resume
)sys rm -f OneDimensionalArrayFunctions2.output
)spool OneDimensionalArrayFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OneDimensionalArrayFunctions2
--R
--R OneDimensionalArrayFunctions2(A: Type,B: Type) is a package constructor
--R Abbreviation for OneDimensionalArrayFunctions2 is ARRAY12
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ARRAY12
--R
--R------------------------------- Operations --------------------------------
--R map : ((A -> B),OneDimensionalArray(A)) -> OneDimensionalArray(B)
--R reduce : (((A,B) -> B),OneDimensionalArray(A),B) -> B
--R scan : (((A,B) -> B),OneDimensionalArray(A),B) -> OneDimensionalArray(B)
--R
--E 1

)spool
)lisp (bye)

---

--- OneDimensionalArrayFunctions2.help ---

====================================================================
OneDimensionalArrayFunctions2 examples
====================================================================

This package provides tools for operating on one-dimensional arrays
with unary and binary functions involving different underlying types

See Also:
  o )show OneDimensionalArrayFunctions2

---
OneDimensionalArrayFunctions2 (ARRAY12)

Exports:
map reduce scan

— package ARRAY12 OneDimensionalArrayFunctions2 —

)abbrev package ARRAY12 OneDimensionalArrayFunctions2
++ Description:
++ This package provides tools for operating on one-dimensional arrays
++ with unary and binary functions involving different underlying types

OneDimensionalArrayFunctions2(A, B): Exports == Implementation where
A, B: Type

VA ==> OneDimensionalArray A
VB ==> OneDimensionalArray B
O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)

Exports ==> with
scan : ((A, B) -> B, VA, B) -> VB
++ scan(f,a,r) successively applies
++ \spad{reduce(f,x,r)} to more and more leading sub-arrays
++ x of one-dimensional array \spad{a}.
++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
++ \spad{scan(f,a,r)} returns
++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
++
++X T1:=OneDimensionalArrayFunctions2(Integer,Integer)
++X adder(a:Integer,b:Integer):Integer == a+b
++X scan(adder,[i for i in 1..10],0)$T1

reduce : ((A, B) -> B, VA, B) -> B
++ reduce(f,a,r) applies function f to each
++ successive element of the
++ one-dimensional array \spad{a} and an accumulant initialized to r.
++ For example, \spad{reduce(_+$_Integer,[1,2,3],0)}
\( \text{spad}\{3+(2+(1+0))\} \). Note that third argument \( r \) may be regarded as the identity element for the function \( f \).

\[ X \ T1 := \text{OneDimensionalArrayFunctions2}(\text{Integer}, \text{Integer}) \]
\[ X \ \text{adder}(a: \text{Integer}, b: \text{Integer}): \text{Integer} = a + b \]
\[ X \ \text{reduce}(\text{adder}, [i \text{ for } i \text{ in } 1..10], 0) \]$T1

\[ \text{map} : (A \to B, VA) \to VB \]
\[ \text{map}(f, a) \text{ applies function } f \text{ to each member of one-dimensional array } \text{spad}\{a\} \text{ resulting in a new one-dimensional array over a possibly different underlying domain.} \]
\[ X \ T1 := \text{OneDimensionalArrayFunctions2}(\text{Integer}, \text{Integer}) \]
\[ X \ \text{map}(x+\to x+2, [i \text{ for } i \text{ in } 1..10])$T1

\[
\begin{align*}
\text{Implementation} & \Rightarrow \text{add} \\
\text{map}(f, v) & \Rightarrow \text{map}(f, v)$O2 \\
\text{scan}(f, v, b) & \Rightarrow \text{scan}(f, v, b)$O2 \\
\text{reduce}(f, v, b) & \Rightarrow \text{reduce}(f, v, b)$O2
\end{align*}
\]

---

**ARRAY12.dotabb**

"ARRAY12" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ARRAY12"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"ARRAY12" -> "A1AGG"

---

package ONECOMP2 OnePointCompletionFunctions2

--- OnePointCompletionFunctions2.input ---

\[ \text{}}set \text{break resume} \]
\[ \text{sys rm -f OnePointCompletionFunctions2.output} \]
\[ \text{spool OnePointCompletionFunctions2.output} \]
\[ \text{set message test on} \]
\[ \text{set message auto off} \]
\[ \text{clear all} \]

--S 1 of 1
\[ \text{}}show OnePointCompletionFunctions2 \]
--R
--R OnePointCompletionFunctions2(R: SetCategory,S: SetCategory) is a package constructor
--R Abbreviation for OnePointCompletionFunctions2 is ONECOMP2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ONECOMP2
--R
--R-------------------------------- Operations --------------------------------
--R map : ((R -> S),OnePointCompletion(R)) -> OnePointCompletion(S)
--R map : ((R -> S),OnePointCompletion(R),OnePointCompletion(S)) -> OnePointCompletion(S)
--R

)spool
)lisp (bye)

---

--- OnePointCompletionFunctions2.help ---

====================================================================
OnePointCompletionFunctions2 examples
====================================================================

Lifting of maps to one-point completions.

See Also:
o )show OnePointCompletionFunctions2

---

OnePointCompletionFunctions2 (ONECOMP2)

Exports:
map
package ONECOMP2 OnePointCompletionFunctions2

)abbrev package ONECOMP2 OnePointCompletionFunctions2
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Lifting of maps to one-point completions.
OnePointCompletionFunctions2(R, S): Exports == Implementation where
   R, S: SetCategory
   OPR ==> OnePointCompletion R
   OPS ==> OnePointCompletion S
Exports ==
   with
   map: (R -> S, OPR) -> OPS
   ++ map(f, r) lifts f and applies it to r, assuming that
   ++ f(infinity) = infinity.
   map: (R -> S, OPR, OPS) -> OPS
   ++ map(f, r, i) lifts f and applies it to r, assuming that
   ++ f(infinity) = i.
Implementation ==
   add
   map(f, r) == map(f, r, infinity())
   map(f, r, i) ==
      (u := retractIfCan r) case R => (f(u::R)::OPS
      i

---

"ONECOMP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ONECOMP2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"ONECOMP2" -> "BASTYPE"
"ONECOMP2" -> "KOERCE"
package OMPKG OpenMathPackage

— OpenMathPackage.input —

)set break resume
)sys rm -f OpenMathPackage.output
)spool OpenMathPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathPackage
--R
--R OpenMathPackage is a package constructor
--R Abbreviation for OpenMathPackage is OMPKG
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for OMPKG
--R
--R------------------------------------------------------------- Operations ----------------------------------
--R OMlistCDs : () -> List(String) OMread : OpenMathDevice -> Any
--R OMreadFile : String -> Any OMreadStr : String -> Any
--R OMsupportsCD? : String -> Boolean
--R OMlistSymbols : String -> List(String)
--R OMsupportsSymbol? : (String,String) -> Boolean
--R OMunhandledSymbol : (String,String) -> Exit
--R
--E 1

)spool
)lisp (bye)

———

— OpenMathPackage.help —

====================================================================
OpenMathPackage examples
====================================================================

OpenMathPackage provides some simple utilities to make reading
OpenMath objects easier.

See Also:
o )show OpenMathPackage

———
OpenMathPackage (OMPKG)

Exports:
OMlistCDs  OMread  OMreadFile  OMreadStr  OMsupportsCD?
OMlistSymbols  OMsupportsSymbol?  OMunhandledSymbol

— package OMPKG OpenMathPackage —

)abbrev package OMPKG OpenMathPackage
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathPackage} provides some simple utilities
++ to make reading OpenMath objects easier.

OpenMathPackage(): with
OMread  : OpenMathDevice -> Any
++ OMread(dev) reads an OpenMath object from \texttt{axiom\{dev\}} and passes it
++ to AXIOM.
OMreadFile  : String -> Any
++ OMreadFile(f) reads an OpenMath object from \texttt{axiom\{f\}} and passes it
++ to AXIOM.
OMreadStr  : String -> Any
++ OMreadStr(f) reads an OpenMath object from \texttt{axiom\{f\}} and passes it
++ to AXIOM.
OMlistCDs  : () -> List(String)
++ OMlistCDs() lists all the CDs supported by AXIOM.
OMlistSymbols  : String -> List(String)
++ OMlistSymbols(cd) lists all the symbols in \texttt{axiom\{cd\}}.
OMsupportsCD?  : String -> Boolean
++ OMsupportsCD?(cd) returns true if AXIOM supports \texttt{axiom\{cd\}}, false
++ otherwise.
OMsupportsSymbol?  : (String, String) -> Boolean
++ OMsupportsSymbol?(s,cd) returns true if AXIOM supports symbol \texttt{axiom\{s\}}
++ from CD \texttt{axiom\{cd\}}, false otherwise.
OMUnhandledSymbol : (String, String) -> Exit
++ OMUnhandledSymbol(s,cd) raises an error if AXIOM reads a symbol which it
++ is unable to handle. Note that this is different from an unexpected
++ symbol.
== add
import OpenMathEncoding
import OpenMathDevice
import String

OMUnhandledSymbol(u,v) ==
  error concat ["AXIOM is unable to process the symbol ",u," from CD ",v,"."]

OMread(dev: OpenMathDevice): Any ==
  interpret(OM_-READ(dev)$Lisp :: InputForm)

OMreadFile(filename: String): Any ==
  dev := OMopenFile(filename, "r", OMencodingUnknown())
  res: Any := interpret(OM_-READ(dev)$Lisp :: InputForm)
  OMclose(dev)
  res

OMreadStr(str: String): Any ==
  strp := OM_-STRINGTOSTRINGPTR(str)$Lisp
  dev := OMopenString(strp pretend String, OMencodingUnknown())
  res: Any := interpret(OM_-READ(dev)$Lisp :: InputForm)
  OMclose(dev)
  res

OMlistCDs(): List(String) ==
  OM_-LISTCDS()$Lisp pretend List(String)

OMlistSymbols(cd: String): List(String) ==
  OM_-LISTSYMBOLS(cd)$Lisp pretend List(String)

import SExpression

OMsupportsCD?(cd: String): Boolean ==
  not null? OM_-SUPPORTSCD(cd)$Lisp

OMsupportsSymbol?(cd: String, name: String): Boolean ==
  not null? OM_-SUPPORTSSYMBOL(cd, name)$Lisp

— OMPKG.dotabb —

"OMPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMPKG"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"OMPKG" -> "STRING"

package OMSERVER OpenMathServerPackage

--- OpenMathServerPackage.input ---

)set break resume
)sys rm -f OpenMathServerPackage.output
)spool OpenMathServerPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OpenMathServerPackage
--R
--R OpenMathServerPackage is a package constructor
--R Abbreviation for OpenMathServerPackage is OMSERVER
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for OMSERVER
--R
--R----------------------------------- Operations -----------------------------------
--R OMreceive : OpenMathConnection -> Any
--R OMsend : (OpenMathConnection,Any) -> Void
--R OMserve : (SingleInteger,SingleInteger) -> Void
--R
--E 1

)spool
)lisp (bye)

---

--- OpenMathServerPackage.help ---

====================================================================
OpenMathServerPackage examples
====================================================================

OpenMathServerPackage provides the necessary operations to run Axiom as an OpenMath server, reading/writing objects to/from a port. Please note the facilities available here are very basic. The idea is that a user calls e.g. Omserve(4000,60) and then another process sends
OpenMath objects to port 4000 and reads the result.

See Also:
o )show OpenMathServerPackage

---

OpenMathServerPackage (OMSERVER)

Exports:
OMreceive  OMsend  OMserve

--- package OMSERVER OpenMathServerPackage ---

)abbrev package OMSERVER OpenMathServerPackage
++ Author: Vilya Harvey
++ Description:
++ \spadtype{OpenMathServerPackage} provides the necessary
++ operations to run AXIOM as an OpenMath server, reading/writing objects
++ to/from a port. Please note the facilities available here are very basic.
++ The idea is that a user calls e.g. \axiom{Omserve(4000,60)} and then
++ another process sends OpenMath objects to port 4000 and reads the result.

OpenMathServerPackage(): with
OMreceive : OpenMathConnection -> Any
++ OMreceive(c) reads an OpenMath object from connection \axiom{c} and
++ returns the appropriate AXIOM object.
OMsend : (OpenMathConnection, Any) -> Void
++ OMsend(c,u) attempts to output \axiom{u} on \axiom{c} in OpenMath.
OMserve : (SingleInteger, SingleInteger) -> Void
++ OMserve(portnum,timeout) puts AXIOM into server mode on port number
++ \axiom{portnum}. The parameter \axiom{timeout} specifies the timeout
++ period for the connection.
== add
import OpenMathDevice
import OpenMathConnection
import OpenMathPackage
import OpenMath

OMreceive(conn: OpenMathConnection): Any ==
  dev: OpenMathDevice := OMconnInDevice(conn)
  OMsetEncoding(dev, OMencodingUnknown);
  OMread(dev)

OMsend(conn: OpenMathConnection, value: Any): Void ==
  dev: OpenMathDevice := OMconnOutDevice(conn)
  OMsetEncoding(dev, OMencodingXML);
  --retractable?(value)$AnyFunctions1(Expression Integer) =>
  -- OMwrite(dev, retract(value)$AnyFunctions1(Expression Integer), true)
  retractable?(value)$AnyFunctions1(Integer) =>
    OMwrite(dev, retract(value)$AnyFunctions1(Integer), true)
  retractable?(value)$AnyFunctions1(Float) =>
    OMwrite(dev, retract(value)$AnyFunctions1(Float), true)
  retractable?(value)$AnyFunctions1(SingleInteger) =>
    OMwrite(dev, retract(value)$AnyFunctions1(SingleInteger), true)
  retractable?(value)$AnyFunctions1(DoubleFloat) =>
    OMwrite(dev, retract(value)$AnyFunctions1(DoubleFloat), true)
  retractable?(value)$AnyFunctions1(String) =>
    OMwrite(dev, retract(value)$AnyFunctions1(String), true)

OMserve(portNum: SingleInteger, timeout: SingleInteger): Void ==
  conn: OpenMathConnection := OMmakeConn(timeout)
  OMbindTCP(conn, portNum)
  val: Any
  while true repeat
    val := OMreceive(conn)
    OMsend(conn, val)

——  OMSERVER.dotabb  ——

"OMSERVER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OMSERVER"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"OMSERVER" -> "STRING"

——
package OPQUERY OperationsQuery

— OperationsQuery.input —

)set break resume
)sys rm -f OperationsQuery.output
)spool OperationsQuery.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OperationsQuery
--R
--R OperationsQuery is a package constructor
--R Abbreviation for OperationsQuery is OPQUERY
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for OPQUERY
--R
--R------------------------------------------ Operations --------------------------------
--R getDatabase : String -> Database(IndexCard)
--E 1

)spool
)lisp (bye)

———

— OperationsQuery.help —

====================================================================
OperationsQuery examples
====================================================================

This package exports tools to create Axiom Library information databases.

See Also:
o )show OperationsQuery

———
OperationsQuery (OPQUERY)

Exports:
getDatabase

— package OPQUERY OperationsQuery —

)abbrev package OPQUERY OperationsQuery
++ Description:
++ This package exports tools to create AXIOM Library information databases.

OperationsQuery(): Exports == Implementation where
Exports == with
    getDatabase: String -> Database(IndexCard)
    ++ getDatabase("char") returns a list of appropriate entries in the
    ++ browser database. The legal values for "char" are "o" (operations),
    ++ "k" (constructors), "d" (domains), "c" (categories) or "p" (packages).
Implementation == add
    getDatabase(s) == getBrowseDatabase(s)$Lisp

— OPQUERY.dotabb —

"OPQUERY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OPQUERY"]
"ORDSET" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ORDSET"]
"OPQUERY" -> "ORDSET"
package ORDCOMP2 OrderedCompletionFunctions2

— OrderedCompletionFunctions2.input —

)set break resume
)sys rm -f OrderedCompletionFunctions2.output
)spool OrderedCompletionFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrderedCompletionFunctions2
--R
--R OrderedCompletionFunctions2(R: SetCategory,S: SetCategory) is a package constructor
--R Abbreviation for OrderedCompletionFunctions2 is ORDCOMP2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ORDCOMP2
--R
--R------------------------------------------------- Operations -----------------------------
--R map : ((R -> S),OrderedCompletion(R)) -> OrderedCompletion(S)
--R map : ((R -> S),OrderedCompletion(R),OrderedCompletion(S),OrderedCompletion(S)) -> OrderedCompletion(S)
--R
--E 1

)spool
)lisp (bye)

——

— OrderedCompletionFunctions2.help —

====================================================================
OrderedCompletionFunctions2 examples
====================================================================

Lifting of maps to ordered completions.

See Also:
o )show OrderedCompletionFunctions2

——
OrderedCompletionFunctions2 (ORDCOMP2)

Exports:
map

— package ORDCOMP2 OrderedCompletionFunctions2 —

)abbrev package ORDCOMP2 OrderedCompletionFunctions2
++ Author: Manuel Bronstein
++ Date Created: 4 Oct 1989
++ Date Last Updated: 4 Oct 1989
++ Description:
++ Lifting of maps to ordered completions.

OrderedCompletionFunctions2(R, S):Exports == Implementation where
  R, S: SetCategory
  ORR ==> OrderedCompletion R
  ORS ==> OrderedCompletion S

Exports ==> with
  map: (R -> S, ORR) -> ORS
  -- map(f, r) lifts f and applies it to r, assuming that
  -- f(plusInfinity) = plusInfinity and that
  -- f(minusInfinity) = minusInfinity.
  map: (R -> S, ORR, ORS, ORS) -> ORS
  -- map(f, r, p, m) lifts f and applies it to r, assuming that
  -- f(plusInfinity) = p and that f(minusInfinity) = m.

Implementation ==> add
  map(f, r) == map(f, r, plusInfinity(), minusInfinity())

  map(f, r, p, m) ==
    zero?(n := whatInfinity r) => (f retract r)::ORS
    one? n => p
    (n = 1) => p
    m
package ORDFUNS OrderingFunctions

--- OrderingFunctions.input ---

)set break resume
)sys rm -f OrderingFunctions.output
)spool OrderingFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show OrderingFunctions
--R
--R OrderingFunctions(dim: NonNegativeInteger,S: OrderedAbelianMonoid) is a package constructor
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ORDFUNS
--R
--R-------------------------------- Operations --------------------------------
--R pureLex : (Vector(S),Vector(S)) -> Boolean
--R reverseLex : (Vector(S),Vector(S)) -> Boolean
--R totalLex : (Vector(S),Vector(S)) -> Boolean
--R
--E 1

)spool
)lisp (bye)
CHAPTER 16.  CHAPTER O

— OrderingFunctions.help —

====================================================================
OrderingFunctions examples
====================================================================

This package provides ordering functions on vectors which are suitable parameters for OrderedDirectProduct.

See Also:
o )show OrderingFunctions

———

OrderingFunctions (ORDFUNS)

Exports:
pureLex  reverseLex  totalLex

—— package ORDFUNS OrderingFunctions ——

)abbrev package ORDFUNS OrderingFunctions
++ Author: Barry Trager
++ Description:
++ This package provides ordering functions on vectors which
++ are suitable parameters for OrderedDirectProduct.

OrderingFunctions(dim,S) : T == C where
  dim : NonNegativeInteger
  S     : OrderedAbelianMonoid
  VS    == Vector S
T == with
  pureLex : (VS,VS) -> Boolean
  ++ pureLex(v1,v2) return true if the vector v1 is less than the
  ++ vector v2 in the lexicographic ordering.
  totalLex : (VS,VS) -> Boolean
  ++ totalLex(v1,v2) return true if the vector v1 is less than the
  ++ vector v2 in the ordering which is total degree refined by
  ++ lexicographic ordering.
  reverseLex : (VS,VS) -> Boolean
  ++ reverseLex(v1,v2) return true if the vector v1 is less than the
  ++ vector v2 in the ordering which is total degree refined by
  ++ the reverse lexicographic ordering.

C == add
  n:NonNegativeInteger:=dim

-- pure lexicographical ordering
  pureLex(v1:VS,v2:VS) : Boolean ==
    for i in 1..n repeat
      if qelt(v1,i) < qelt(v2,i) then return true
      if qelt(v2,i) < qelt(v1,i) then return false
    false

-- total ordering refined with lex
  totalLex(v1:VS,v2:VS) : Boolean ==
    n1:S:=0
    n2:S:=0
    for i in 1..n repeat
      n1:= n1+qelt(v1,i)
      n2:=n2+qelt(v2,i)
    n1<n2 => true
    n2<n1 => false
    for i in 1..n repeat
      if qelt(v1,i) < qelt(v2,i) then return true
      if qelt(v2,i) < qelt(v1,i) then return false
    false

-- reverse lexicographical ordering
  reverseLex(v1:VS,v2:VS) : Boolean ==
    n1:S:=0
    n2:S:=0
    for i in 1..n repeat
      n1:= n1+qelt(v1,i)
      n2:=n2+qelt(v2,i)
    n1<n2 => true
    n2<n1 => false
    for i in reverse(1..n) repeat
      if qelt(v2,i) < qelt(v1,i) then return true
      if qelt(v1,i) < qelt(v2,i) then return false
    false
package ORTHPOL OrthogonalPolynomialFunctions

-- orthogonalPolynomialFunctions.input --

)set break resume
/sys rm -f OrthogonalPolynomialFunctions.output
/spool OrthogonalPolynomialFunctions.output
)set message test on
)set message auto off
clear all
--S 1 of 1
/show OrthogonalPolynomialFunctions
--R
--R OrthogonalPolynomialFunctions(R: CommutativeRing) is a package constructor
--R Abbreviation for OrthogonalPolynomialFunctions is ORTHPOL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ORTHPOL
--R
--R-------------------------------- Operations --------------------------------
--R chebyshevT : (NonNegativeInteger,R) -> R
--R chebyshevU : (NonNegativeInteger,R) -> R
--R hermiteH : (NonNegativeInteger,R) -> R
--R laguerreL : (NonNegativeInteger,R) -> R
--R laguerreL : (NonNegativeInteger,NonNegativeInteger,R) -> R
--R legendreP : (NonNegativeInteger,R) -> R if R has ALGEBRA(FRAC(INT))
--R
--E 1

)spool
)lisp (bye)
OrthogonalPolynomialFunctions (ORTHPOL)

Exports:
chebyshevT  chebyshevU  hermiteH  laguerreL  laguerreL  legendreP

— package ORTHPOL OrthogonalPolynomialFunctions —

)abbrev package ORTHPOL OrthogonalPolynomialFunctions
++ Author: Stephen M. Watt
++ Date Created: 1990
++ Date Last Updated: June 25, 1991
++ Description:
++ This package provides orthogonal polynomials as functions on a ring.

OrthogonalPolynomialFunctions(R: CommutativeRing):Exports == Impl where
   NNI ==> NonNegativeInteger
   RN  ==> Fraction Integer

  Exports ==> with
chebyshevT: (NNI, R) -> R
  ++ chebyshevT(n,x) is the n-th Chebyshev polynomial of the first
  ++ kind, \spad{T[n](x)}. These are defined by
  ++ \spad{(1-t*x)/(1-2*t*x+t**2) = sum(T[n](x) *t**n, n = 0..)}.

chebyshevU: (NNI, R) -> R
  ++ chebyshevU(n,x) is the n-th Chebyshev polynomial of the second
  ++ kind, \spad{U[n](x)}. These are defined by
  ++ \spad{1/(1-2*t*x+t**2) = sum(T[n](x) *t**n, n = 0..)}.

hermiteH: (NNI, R) -> R
  ++ hermiteH(n,x) is the n-th Hermite polynomial, \spad{H[n](x)}.
  ++ These are defined by
  ++ \spad{exp(2*t*x-t**2) = sum(H[n](x)*t**n/n!, n = 0..)}.

laguerreL: (NNI, R) -> R
  ++ laguerreL(n,x) is the n-th Laguerre polynomial, \spad{L[n](x)}.
  ++ These are defined by
  ++ \spad{exp(-t*x/(1-t))/(1-t) = sum(L[n](x)*t**n/n!, n = 0..)}.

laguerreL: (NNI, NNI, R) -> R
  ++ laguerreL(m,n,x) is the associated Laguerre polynomial,
  ++ \spad{L<m>[n](x)}. This is the m-th derivative of \spad{L[n](x)}.

if R has Algebra RN then
  legendreP: (NNI, R) -> R
  ++ legendreP(n,x) is the n-th Legendre polynomial,
  ++ \spad{P[n](x)}. These are defined by
  ++ \spad{1/sqrt(1-2*x*t+t**2) = sum(P[n](x) *t**n, n = 0..)}.

Impl ==> add
p0, p1: R
import IntegerCombinatoricFunctions()

laguerreL(n, x) ==
  n = 0 => 1
  {p1, p0} := (-x + 1, 1)
  for i in 1..n-1 repeat
    (p1, p0) := ((2*i::R + 1 - x)*p1 - i**2*p0, p1)
  p1

laguerreL(m, n, x) ==
  ni := n::Integer
  mi := m::Integer
  cx := (-1)**m * binomial(ni,ni-mi) * factorial(ni)
  p0 := 1
  p1 := cx::R
  for j in 1..ni-mi repeat
cx := -cx*(ni-mi-j+1)
cx := (cx exquo ((mi+j)*j))::Integer
p0 := p0 * x
p1 := p1 + cx*p0
p1

chebyshevT(n, x) ==
  n = 0 => 1
  (p1, p0) := (x, 1)
  for i in 1..n-1 repeat
    (p1, p0) := (2*x*p1 - p0, p1)
p1

chebyshevU(n, x) ==
  n = 0 => 1
  (p1, p0) := (2*x, 1)
  for i in 1..n-1 repeat
    (p1, p0) := (2*x*p1 - 2*i*p0, p1)
p1

if R has Algebra RN then
  legendreP(n, x) ==
    n = 0 => 1
    p0 := 1
    p1 := x
    for i in 1..n-1 repeat
      c: RN := 1/(i+1)
      (p1, p0) := (c*((2*i+1)*x*p1 - i*p0), p1)
p1

---

"ORTHPOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ORTHPOL"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"ORTHPOL" -> "PID"
"ORTHPOL" -> "OAGROUP"

---
package OUT OutputPackage

<table>
<thead>
<tr>
<th>OutputPackage.input</th>
</tr>
</thead>
<tbody>
<tr>
<td>)set break resume</td>
</tr>
<tr>
<td>)sys rm -f OutputPackage.output</td>
</tr>
<tr>
<td>)spool OutputPackage.output</td>
</tr>
<tr>
<td>)set message test on</td>
</tr>
<tr>
<td>)set message auto off</td>
</tr>
<tr>
<td>)clear all</td>
</tr>
</tbody>
</table>

---
| OutputPackage is a package constructor |
| Abbreviation for OutputPackage is OUT |
| This constructor is exposed in this frame. |
| Issue )edit bookvol10.4.pamphlet to see algebra source code for OUT |

---
| Operations |
| output : String -> Void |
| output : OutputForm -> Void |
| output : (String,OutputForm) -> Void |
| outputList : List(Any) -> Void |

---
| OutputPackage.help |

| OutputPackage examples |

OutPackage allows pretty-printing from programs.

See Also:
  o )show OutputPackage
OutputPackage (OUT)

Exports:
output outputList

— package OUT OutputPackage —

)abbrev package OUT OutputPackage
++ Author: Stephen M. Watt
++ Date Created: February 1986
++ Date Last Updated: October 27 1995 (MCD)
++ Description:
++ OutPackage allows pretty-printing from programs.

OutputPackage: with

    output: String -> Void
    ++ output(s) displays the string s on the ‘‘algebra output’’
    ++ stream, as defined by \spad{set output algebra}.

    output: OutputForm -> Void
    ++ output(x) displays the output form x on the
    ++ ‘‘algebra output’’ stream, as defined by
    ++ \spad{set output algebra}.

    output: (String, OutputForm) -> Void
    ++ output(s,x) displays the string s followed by the form x
    ++ on the ‘‘algebra output’’ stream, as defined by
    ++ \spad{set output algebra}.

    outputList: (List Any) -> Void
    ++ outputList(l) displays the concatenated components of the
    ++ list l on the ‘‘algebra output’’ stream, as defined by
    ++ \spad{set output algebra}; quotes are stripped
    ++ from strings.

== add
-- ExpressionPackage()
E ==> OutputForm
putout ==> mathprint$Lisp

s: String
e: OutputForm
l: List Any

output e ==
    mathprint(e)$Lisp
    void()

-- Note that we have to do the pretend here because otherwise we will
-- try to load STRING which is not yet compiled during build.
output s ==
    output(s pretend OutputForm)

output(s,e) ==
    output blankSeparate [s pretend OutputForm, e]

outputList(l) ==
    -- MGR
    output hconcat
    [if retractable?(x)$AnyFunctions1(String) then
        message(retract(x)$AnyFunctions1(String))$OutputForm
    else
        x::OutputForm
    for x in l]

——

— OUT.dotabb —

"OUT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=OUT"]
"Package" [color="#FF4488"]
"OUT" -> "Package"

——
Chapter 17

Chapter P

package PAFF PackageForAlgebraicFunctionField

--- PackageForAlgebraicFunctionField.input ---

)set break resume
)set rm -f PackageForAlgebraicFunctionField.output
)spool PackageForAlgebraicFunctionField.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForAlgebraicFunctionField
--R
--R PackageForAlgebraicFunctionField(K: Field,symb: List(Symbol),BLMET: BlowUpMethodCategory) is a package
--R Abbreviation for PackageForAlgebraicFunctionField is PAFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAFF
--R
--R----------------------------------- Operations -----------------------------------
--R fullDesTree : () -> Void    fullInfClsPt : () -> Void
--R genus : () -> NonNegativeInteger   genusNeg : () -> Integer
--R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if K has FINITE
--R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if K has FINITE
--R adjunctionDivisor : () -> Divisor(Places(K))
--R classNumber : () -> Integer if K has FINITE
--R desingTree : () -> List(DesingTree(InfClsPt(K,symb,BLMET))))
--R desingTreeNoFullParam : () -> List(DesingTree(InfClsPt(K,symb,BLMET))))
--R eval : (DistributedMultivariatePolynomial(symb,K),Places(K)) -> K

3213
--R eval : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),Places(K)) -> K
--R eval : (Fraction(DistributedMultivariatePolynomial(symb,K)),Places(K)) -> K
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),Places(K)) -> Union(K,"failed")
--R evalIfCan : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),Places(K)) -> Union(K,"failed")
--R findOrderOfDivisor : (Divisor(Places(K)),Integer,Integer) -> Record(ord: Integer,num: DistributedMultivariatePolynomial(symb,K),den: DistributedMultivariatePolynomial(symb,K),upTo: Integer)
--R goppaCode : (Divisor(Places(K)),Divisor(Places(K))) -> Matrix(K)
--R goppaCode : (Divisor(Places(K)),List(Places(K))) -> Matrix(K)
--R homogenize : (DistributedMultivariatePolynomial(symb,K),Integer) -> DistributedMultivariatePolynomial(symb,K)
--R interpolateForms : (Divisor(Places(K)),NonNegativeInteger) -> List(DistributedMultivariatePolynomial(symb,K))
--R interpolateFormsForFact : (Divisor(Places(K)),List(DistributedMultivariatePolynomial(symb,K))) -> List(DistributedMultivariatePolynomial(symb,K))
--R intersectionDivisor : DistributedMultivariatePolynomial(symb,K) -> Divisor(DistributedMultivariatePolynomial(symb,K))
--R lBasis : (Divisor(Places(K)),NonNegativeInteger) -> List(Fraction(DistributedMultivariatePolynomial(symb,K)))
--R lBasis : Divisor(Places(K)) -> Record(num: List(DistributedMultivariatePolynomial(symb,K)),den: DistributedMultivariatePolynomial(symb,K),upTo: Integer)
--R numberOfPlacesOfDegree : PositiveInteger -> Integer if K has FINITE
--R numberOfPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if K has FINITE
--R numberRatPlacesExtDeg : PositiveInteger -> Integer if K has FINITE
--R parametrize : (DistributedMultivariatePolynomial(symb,K),Places(K)) -> NeitherSparseOrDensePowerSeries(K)
--R placesAbove : ProjectivePlane(K) -> List(Places(K))
--R placesOfDegree : PositiveInteger -> List(Places(K)) if K has FINITE
--R pointDominateBy : Places(K) -> ProjectivePlane(K)
--R rationalPlaces : () -> List(Places(K))
--R rationalPoints : () -> List(ProjectivePlane(K))
--R setCurve : DistributedMultivariatePolynomial(symb,K) -> DistributedMultivariatePolynomial(symb,K)
--R setSingularPoints : List(ProjectivePlane(K)) -> List(ProjectivePlane(K))
--R singularPoints : () -> List(ProjectivePlane(K))
--R theCurve : () -> DistributedMultivariatePolynomial(symb,K)

--E 1

)spool
)lisp (bye)
PackageForAlgebraicFunctionField (PAFF)

Exports:
adjunctionDivisor  classNumber  desingTree
desingTreeWoFullParam  eval  evalIfCan
findOrderOfDivisor  fullDesTree  fullInfClsPt
genus  genusNeg  goppaCode
homogenize  interpolateForms  interpolateFormsForFact
intersectionDivisor  IBasis  LPolynomial
numberOfPlacesOfDegree  numberPlacesDegExtDeg  numberRatPlacesExtDeg
parametrize  placesAbove  placesOfDegree
pointDominateBy  projectivePoint  rationalPlaces
rationalPoints  setCurve  setSingularPoints
singularPoints  theCurve  ZetaFunction

— package PAFF PackageForAlgebraicFunctionField —

)abbrev package PAFF PackageForAlgebraicFunctionField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that implements the Brill-Noether algorithm.
++ Part of the PAFF package
PackageForAlgebraicFunctionField(K,symb,BLMET):Exports == Implementation where
K:Field
symb : List(Symbol)
PolyRing ==> DistributedMultivariatePolynomial(symb,K)
E ==> DirectProduct(#symb,NonNegativeInteger)
BLMET : BlowUpMethodCategory
AFP    ==> AffinePlane(K)
ProjPt  ==> ProjectivePlane(K)
PCS    ==> NeitherSparseOrDensePowerSeries(K)
Plc ==> Places(K)
DIVISOR ==> Divisor(Plc)
InfClsPoint ==> InfClsPt(K,symb,BLMET)
DesTree ==> DesingTree(InfClsPoint)
FRACPOLY ==> Fraction PolyRing
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
UTSZ ==> UnivariateTaylorSeriesCZero(Integer,t)
PAFFPC ==> GeneralPackageForAlgebraicFunctionField
PACKPOLY ==> PackageForPoly(K,PolyRing,E,#symb)
BP ==> PAFFPC(K,symb,PolyRing,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)

Exports ==> with
    homogenize: (PolyRing,Integer) -> PolyRing

    interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing)

    fullDesTree: () -> Void

    fullInfClsPt: () -> Void

    setCurve: PolyRing -> PolyRing

    projectivePoint: List K -> ProjPt

    pointDominateBy : Plc -> ProjPt
        ++ pointDominateBy(pl) returns the projective point dominated
        ++ by the place pl.

    placesAbove: ProjPt -> List Plc

    setSingularPoints: List ProjPt -> List ProjPt

    goppaCode: (DIVISOR,DIVISOR) -> Matrix K

    goppaCode: (DIVISOR,List(Plc)) -> Matrix K

    rationalPlaces: () -> List Plc
        ++ rationalPlaces returns all the rational places of the
        ++ curve defined by the polynomial given to the package.

    theCurve: () -> PolyRing
        ++ theCurve returns the specified polynomial for the package.

    genus: () -> NNI
        ++ genus returns the genus of the curve defined by the polynomial
        ++ given to the package.

    genusNeg: () -> Integer
desingTreeWoFullParam : () -> List DesTree
++ desingTreeWoFullParam returns the desingularisation trees at all
++ singular points of the curve defined by the polynomial given to
++ the package. The local parametrizations are not computed.

desingTree: () -> List DesTree
++ desingTree returns the desingularisation trees at all singular
++ points of the curve defined by the polynomial given to the package.

rationalPoints: () -> List(ProjPt)
singularPoints: () -> List(ProjPt)
++ rationalPoints() returns the singular points of the
++ curve defined by the polynomial given to the package.
++ If the singular points lie in an extension of the specified
++ ground field an error message is issued specifying the extension
++ degree needed to find all singular points.

parametrize: (PolyRing,Plc) -> PCS
++ parametrize(f,pl) returns a local parametrization of f at the place pl.

lBasis: (DIVISOR,NNI) -> List FRACPOLY

lBasis: DIVISOR -> Record(num:List PolyRing, den: PolyRing)
++ lBasis computes a basis associated to the specified divisor

findOrderOfDivisor: (DIVISOR,Integer,Integer) -> _.
Record(ord:Integer,num:PolyRing,den:PolyRing,upTo:Integer)

interpolateForms: (DIVISOR,NNI) -> List(PolyRing)
++ interpolateForms(d,n) returns a basis of the interpolate forms of
++ degree n of the divisor d.

eval: (PolyRing,Plc) -> K
++ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing,PolyRing,Plc) -> K
++ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FRACPOLY,Plc) -> K
++ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,pl) evaluate f at the place pl
++ (returns "failed" if it is a pole).

evalIfCan: (PolyRing,PolyRing,Plc) -> Union(K,"failed")
++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
++ (returns "failed" if it is a pole).
evalIfCan: (FRACPOLY, Plc) -> Union(K, "failed")
++ evalIfCan(u, pl) evaluate the function u at the place pl
++ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
++ intersectionDivisor(pol) compute the intersection divisor (the
++ Cartier divisor) of the form pol with the curve. If some
++ intersection points lie in an extension of the ground field,
++ an error message is issued specifying the extension degree
++ needed to find all the intersection points.
++ (If pol is not homogeneous an error message is issued).

adjunctionDivisor: () -> DIVISOR
++ adjunctionDivisor computes the adjunction divisor of the plane
++ curve given by the polynomial set with the function setCurve.

if K has Finite then --should we say LocallyAlgebraicallyClosedField??

LPolynomial: () -> SparseUnivariatePolynomial Integer
++ Returns the L-Polynomial of the curve.

LPolynomial: PI -> SparseUnivariatePolynomial Integer
++ LPolynomial(d) returns the L-Polynomial of the curve in
++ constant field extension of degree d.

classNumber: () -> Integer
++ Returns the class number of the curve.

placesOfDegree: PI -> List Plc
++ placesOfDegree(d) returns all places of degree d of the curve.

numberOfPlacesOfDegree: PI -> Integer
++ returns the number of places of the given degree

numberRatPlacesExtDeg: PI -> Integer
++ numberRatPlacesExtDeg(n) returns the number of rational
++ places in the constant field extension of degree n

numberPlacesDegExtDeg: (PI, PI) -> Integer
++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
++ places of degree d in the constant field extension of degree n

ZetaFunction: () -> UTSZ
++ Returns the Zeta function of the curve. Calculated by
++ using the L-Polynomial

ZetaFunction: PI -> UTSZ
++ Returns the Zeta function of the curve in constant field
++ extension. Calculated by using the L-Polynomial
Implementation ==> add
import BP

homogenize(pol,n) == homogenize(pol,n)$PACKPOLY

pointDominatesBy(pl)== pointDominatesBy(pl)$BP

placesAbove(pt)== placesAbove(pt)$BP

setSingularPoints(lspt)== setSingularPoints(lspt)$BP

projectivePoint(lpt)==projectivePoint(lpt)$ProjPt

interpolateFormsForFact(d,lm)==
  interpolateFormsForFact(d,lm)$BP

if K has Finite then

  goppaCode(d:DIVISOR,lp:List(Plc))==
    lb:=lBasis(d)
    dd:=lb.den
    ll:=[eval(f,dd,pl) for pl in lp for f in lb.num]
    matrix ll

  goppaCode(d:DIVISOR,p:DIVISOR)==
    lp:=supp p
    goppaCode(d,lp)

ZetaFunction == ZetaFunction()$BP

ZetaFunction(d) == ZetaFunction(d)$BP

numberOfPlacesOfDegree(i)==numberOfPlacesOfDegree(i)$BP

placesOfDegree(i) ==placesOfDegree(i)$BP

numberOfRatPlacesExtDeg(extDegree)==numberRatPlacesExtDeg(extDegree)$BP

numberOfPlacesDegExtDeg(degree,extDegree)==
  numberPlacesDegExtDeg(degree,extDegree)$BP

LPolynomial == LPolynomial()$BP

LPolynomial(extDeg)==LPolynomial(extDeg)$BP

classNumber== classNumber()$BP

rationalPlaces == rationalPlaces()$BP

rationalPoints==rationalPoints()$BP
crvLocal:PolyRing

eval(f:PolyRing,pl:Plc)==
  dd := degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  eval(f,pl)$BP

evalIfCan(f:PolyRing,pl:Plc)==
  dd := degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  evalIfCan(f,pl)$BP

setCurve(pol)==setCurve(pol)$BP

1Basis(divis)==1Basis(divis)$BP

genus==genus()$BP

genusNeg==genusNeg()$BP

theCurve==theCurve()$BP

desingTree==desingTree()$BP

desingTreeWoFullParam== desingTreeWoFullParam()$BP

-- compute the adjunction divisor of the curve using
-- adjunctionDivisor from DesingTreePackage
adjunctionDivisor == adjunctionDivisor()$BP

singularPoints==singularPoints()$BP

parametrize(f,pl)==parametrize(f,pl)$BP

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==interpolateForms(d,n)$BP

eval(f:PolyRing,g:PolyRing,pl:Plc)==eval(f,g,pl)$BP

eval(u:FRACPOLY,pl:Plc)==
  ff:=numer u
  gg:=denom u
  eval(ff,gg,pl)

evalIfCan(f:PolyRing,g:PolyRing,pl:Plc)==evalIfCan(f,g,pl)$BP

evalIfCan(u:FRACPOLY,pl:Plc)==
  ff:=numer u
  gg:=denom u
evalIfCan(ff,gg,pl)
intersectionDivisor(pol)==intersectionDivisor(pol)$BP
fullDesTree==
  fullOutput()$DesTree => fullOutput(false())$DesTree
  fullOutput(true())$DesTree
fullInfClsPt==
  fullOutput()$InfClsPoint => fullOutput(false())$InfClsPoint
  fullOutput(true())$InfClsPoint

package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField

--R PackageForAlgebraicFunctionFieldOverFiniteField(K: FiniteFieldCategory,symb: List(Symbol),BLMET: BlowUpMethodCategory) is a package constructor
--R Abbreviation for PackageForAlgebraicFunctionFieldOverFiniteField is PAFFFF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAFFFF
CHAPTER 17. CHAPTER P

---R

---R----------------------------- Operations -----------------------------
---R fullDesTree : () -> Void fullInfClsPt : () -> Void
---R genus : () -> NonNegativeInteger genusNeg : () -> Integer
---R LPolynomial : () -> SparseUnivariatePolynomial(Integer) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R LPolynomial : PositiveInteger -> SparseUnivariatePolynomial(Integer) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R ZetaFunction : () -> UnivariateTaylorSeriesCZero(Integer,t) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R ZetaFunction : PositiveInteger -> UnivariateTaylorSeriesCZero(Integer,t) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R adjunctionDivisor : () -> Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
---R classNumber : () -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R desingTree : () -> List(DesingTree(InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)))
---R desingTreeWoFullParam : () -> List(DesingTree(InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)))
---R eval : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> K
---R eval : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> K
---R eval : (Fraction(DistributedMultivariatePolynomial(symb,K)),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> K
---R evalIfCan : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Union(K,"failed")
---R evalIfCan : (DistributedMultivariatePolynomial(symb,K),DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Union(K,"failed")
---R evalIfCan : (Fraction(DistributedMultivariatePolynomial(symb,K)),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> Union(K,"failed")
---R findOrderOfDivisor : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),Integer,Integer) -> Record(ord: Integer,num: DistributedMultivariatePolynomial(symb,K),den: DistributedMultivariatePolynomial(symb,K),upTo: Integer)
---R goppaCode : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))) -> Matrix(K)
---R goppaCode : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))) -> Matrix(K)
---R homogenize : (DistributedMultivariatePolynomial(symb,K),Integer) -> DistributedMultivariatePolynomial(symb,K)
---R interpolateForms : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),NonNegativeInteger) -> List(DistributedMultivariatePolynomial(symb,K))
---R interpolateFormsForFact : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),List(DistributedMultivariatePolynomial(symb,K))) -> List(DistributedMultivariatePolynomial(symb,PseudoAlgebraicClosureOfFiniteField(K)))
---R intersectionDivisor : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))) -> Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
---R lBasis : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),NonNegativeInteger) -> List(DistributedMultivariatePolynomial(symb,K))
---R lBasis : (Divisor(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)),Integer) -> Record(num: List(DistributedMultivariatePolynomial(symb,K)),den: DistributedMultivariatePolynomial(symb,K))
---R numberOfPlacesOfDegree : PositiveInteger -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R numberPlacesDegExtDeg : (PositiveInteger,PositiveInteger) -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R numberRatPlacesExtDeg : PositiveInteger -> Integer if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R parametrize : (DistributedMultivariatePolynomial(symb,K),PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> NeitherSparseOrDensePowerSeries(PseudoAlgebraicClosureOfFiniteField(K))
---R placesAbove : ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K) -> List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
---R placesOfDegree : PositiveInteger -> List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) if PseudoAlgebraicClosureOfFiniteField(K) has FINITE
---R pointDominateBy : PlacesOverPseudoAlgebraicClosureOfFiniteField(K) -> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
---R projectivePoint : List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K)) -> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
---R rationalPlaces : () -> List(PlacesOverPseudoAlgebraicClosureOfFiniteField(K))
---R rationalPoints : () -> List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
---R setCurve : DistributedMultivariatePolynomial(symb,K) -> DistributedMultivariatePolynomial(symb,K)
---R setSingularPoints : List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)) -> List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
---R singularPoints : () -> List(ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
---R theCurve : () -> DistributedMultivariatePolynomial(symb,K)
---R translateToOrigin : (DistributedMultivariatePolynomial(symb,K),ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K))
--- PackageForAlgebraicFunctionFieldOverFiniteField.help ---

PackageForAlgebraicFunctionFieldOverFiniteField examples

Part of the PAFF package

See Also:
o )show PackageForAlgebraicFunctionFieldOverFiniteField

---

PackageForAlgebraicFunctionFieldOverFiniteField (PAFFFF)

Exports:
adjunctionDivisor  
desingTreeWoFullParam  
findOrderOfDivisor  
genus  
homogenize  
intersectionDivisor  
numberOfPlacesOfDegree  
parametrize  
pointDominateBy  
rationalPoints  
singularPoints  
ZetaFunction  

classNumber  
eval  
fullDesTree  
genusNeg  
interpolateForms  
LPolynomial  
numberPlacesDegExtDeg  
placesAbove  
projectivePoint  
setCurve  
theCurve  
desingTree  
evalIfCan  
fullInfClsPt  
goppaCode  
interpolateFormsForFact  
numberRatPlacesExtDeg  
placesOfDegree  
rationalPlaces  
setSingularPoints  
translateToOrigin

--- package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField ---
### CHAPTER 17. CHAPTER P

)abbrev package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField
++ Author: Gaetan Hache
++ Date created: June 1995
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ A package that implements the Brill-Noether algorithm.
++ Part of the PAFF package
PackageForAlgebraicFunctionFieldOverFiniteField(K,symb,BLMET):Exp == Impl where

K:FiniteFieldCategory -- Field
symb : List(Symbol)
BLMET : BlowUpMethodCategory
DK ==> PseudoAlgebraicClosureOfFiniteField(K)
PolyRing ==> DistributedMultivariatePolynomial(symb,K)
PolyRing2 ==> DistributedMultivariatePolynomial(symb,DK)
ProjPt ==> ProjectivePlaneOverPseudoAlgebraicClosureOfFiniteField(K)
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
UTSZ ==> UnivariateTaylorSeriesCZero(Integer,t)
PAFFPC ==> GeneralPackageForAlgebraicFunctionField
PCS ==> NeitherSparseOrDensePowerSeries(DK)
Plc ==> PlacesOverPseudoAlgebraicClosureOfFiniteField(K)
DIVISOR ==> Divisor(Plc)
InfClsPoint ==> InfinitlyClosePointOverPseudoAlgebraicClosureOfFiniteField(K,symb,BLMET)
DesTree ==> DesingTree(InfClsPoint)
FracPoly ==> Fraction PolyRing
PackPoly ==> PackageForPoly(DK,PolyRing2,E,#symb)
E ==> DirectProduct(#symb,NNI)
BP ==> PAFFPC(DK,symb,PolyRing2,E,ProjPt,PCS,Plc,DIVISOR,InfClsPoint,DesTree,BLMET)

Exp == with
  homogenize: (PolyRing,Integer) -> PolyRing
  fullDesTree: () -> Void
  fullInfClsPt: () -> Void
  setCurve: PolyRing -> PolyRing
  translateToOrigin: (PolyRing, ProjPt) -> PolyRing2
  goppaCode: (DIVISOR,DIVISOR) -> Matrix K
  goppaCode: (DIVISOR,List(Plc)) -> Matrix K
  pointDominateBy : Plc -> ProjPt
    ++ pointDominateBy(pl) returns the projective point dominated
    ++ by the place pl.
placesAbove: ProjPt -> List Plc
projectivePoint: List DK -> ProjPt
setSingularPoints: List ProjPt -> List ProjPt
rationalPlaces: () -> List Plc
   ++ rationalPlaces returns all the rational places of the curve defined by the polynomial given to the package.
theCurve: () -> PolyRing
   ++ theCurve returns the specified polynomial for the package.
genus: () -> NNI
   ++ genus returns the genus of the curve defined by the polynomial given to the package.
genusNeg: () -> Integer
desingTreeWoFullParam: () -> List DesTree
   ++ desingTreeWoFullParam returns the desingularisation trees at all singular points of the curve defined by the polynomial given to the package. The local parametrizations are not computed.
desingTree: () -> List DesTree
   ++ desingTree returns the desingularisation trees at all singular points of the curve defined by the polynomial given to the package.
rationalPoints: () -> List(ProjPt)
singularPoints: () -> List(ProjPt)
   ++ rationalPoints() returns the singular points of the curve defined by the polynomial given to the package.
   ++ If the singular points lie in an extension of the specified ground field an error message is issued specifying the extension degree needed to find all singular points.
parametrize: (PolyRing,Plc) -> PCS
   ++ parametrize(f,pl) returns a local parametrization of f at the place pl.
lBasis: (DIVISOR,NNI) -> List FracPoly
   ++ lBasis computes a basis associated to the specified divisor.
findOrderOfDivisor: (DIVISOR,Integer,Integer) -> 
   Record(ord:Integer,num:PolyRing,den:PolyRing,upTo:Integer)
interpolateFormsForFact: (DIVISOR,List PolyRing) -> List(PolyRing2)
interpolateForms: (DIVISOR, NNI) -> List(PolyRing)
  ++ interpolateForms(d,n) returns a basis of the interpolate forms of
  ++ degree n of the divisor d.

eval: (PolyRing, Plc) -> K
  ++ eval(f,pl) evaluate f at the place pl.

eval: (PolyRing, PolyRing, Plc) -> K
  ++ eval(f,g,pl) evaluate the function f/g at the place pl.

eval: (FracPoly, Plc) -> K
  ++ eval(u,pl) evaluate the function u at the place pl.

evalIfCan: (PolyRing, Plc) -> Union(K, "failed")
  ++ evalIfCan(f,pl) evaluate f at the place pl
  ++ (returns "failed" if it is a pole).

evalIfCan: (PolyRing, PolyRing, Plc) -> Union(K, "failed")
  ++ evalIfCan(f,g,pl) evaluate the function f/g at the place pl
  ++ (returns "failed" if it is a pole).

evalIfCan: (FracPoly, Plc) -> Union(K, "failed")
  ++ evalIfCan(u,pl) evaluate the function u at the place pl
  ++ (returns "failed" if it is a pole).

intersectionDivisor: PolyRing -> DIVISOR
  ++ intersectionDivisor(pol) compute the intersection divisor of the
  ++ form pol with the curve.
  ++ (If pol is not homogeneous an error message is issued).

adjunctionDivisor: () -> DIVISOR
  ++ adjunctionDivisor computes the adjunction divisor of the plane
  ++ curve given by the polynomial defined by setCurve.

if DK has Finite then --should we say LocallyAlgebraicallyClosedField??

LPolynomial: () -> SparseUnivariatePolynomial Integer
  ++ Returns the L-Polynomial of the curve.

LPolynomial: PI -> SparseUnivariatePolynomial Integer
  ++ LPolynomial(d) returns the L-Polynomial of the curve in
  ++ constant field extension of degree d.

classNumber: () -> Integer
  ++ Returns the class number of the curve.

placesOfDegree: PI -> List Plc
  ++ placesOfDegree(d) returns all places of degree d of the
  ++ curve.
numberOfPlacesOfDegree: PI -> Integer
++ returns the number of places of the given degree

numberRatPlacesExtDeg: PI -> Integer
++ numberRatPlacesExtDeg(n) returns the number of rational
++ places in the constant field extension of degree n

numberPlacesDegExtDeg: (PI, PI) -> Integer
++ numberRatPlacesExtDegExtDeg(d, n) returns the number of
++ places of degree d in the constant field extension of
++ degree n

ZetaFunction: () -> UTSZ
++ Returns the Zeta function of the curve. Calculated by
++ using the L-Polynomial

ZetaFunction: PI -> UTSZ
++ Returns the Zeta function of the curve in constant field
++ extension. Calculated by using the L-Polynomial

Impl ==> add
import BP

homogenize(pol,n) == homogenize(pol,n)$PackageForPoly(K,PolyRing,E,#symb)
toPolyRing2: PolyRing -> PolyRing2
toPolyRing: PolyRing2 -> PolyRing

projectivePoint(lpt)==projectivePoint(lpt)$ProjPt

pointDominateBy(pl)== pointDominateBy(pl)$BP
placesAbove(pt)== placesAbove(pt)$BP
setSingularPoints(lapt)== setSingularPoints(lapt)$BP

findOrderOfDivisor(divis,lb,hb) ==
  ens:=findOrderOfDivisor(divis,lb,hb)$BP
  [ens.ord, toPolyRing ens.num, toPolyRing ens.den, ens.upTo]

setCurve(pol)==
  ooo:=setCurve(toPolyRing2 pol)$BP
  pol

ZetaFunction == ZetaFunction()$BP
ZetaFunction(d) == ZetaFunction(d)$BP

numberOfPlacesOfDegree(i)==numberOfPlacesOfDegree(i)$BP
placesOfDegree(i) == placesOfDegree(i)$BP

numberRatPlacesExtDeg(extDegree) == numberRatPlacesExtDeg(extDegree)$BP

numberPlacesDegExtDeg(degree, extDegree) ==
  numberPlacesDegExtDeg(degree, extDegree)$BP

LPolynomial == LPolynomial()$BP

LPolynomial(extDeg) == LPolynomial(extDeg)$BP

classNumber == classNumber()$BP

rationalPlaces == rationalPlaces()$BP

rationalPoints == rationalPoints()$BP

goppaCode(d: DIVISOR, lp: List(Plc)) ==
  lb := lBasis(d)
  dd := lb.den
  ll := [eval(f, dd, pl) for pl in lp] for f in lb.num
  matrix ll

goppaCode(d: DIVISOR, p: DIVISOR) ==
  lp := supp p
goppaCode(d, lp)

toPolyRing(pol) ==
  zero?(pol) => 0$PolyRing
  lc := leadingCoefficient pol
  lce : K := retract lc
  lm := leadingMonomial pol
  lt := degree lm
  monomial(lce, lt)$PolyRing + toPolyRing(reductum pol)

toPolyRing2(pol) ==
  zero?(pol) => 0$PolyRing2
  lc := leadingCoefficient pol
  lce : DK := lc :: DK
  lm := leadingMonomial pol
  lt := degree lm
  monomial(lce, lt)$PolyRing2 + toPolyRing2(reductum pol)

evalIfCan(f: PolyRing, pl: Plc) ==
  dd := degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  ee := evalIfCan(toPolyRing2 f, pl)$BP
  ee case "failed" => "failed"
  retract ee
eval(f:PolyRing,pl:P1c)==
   dd:= degree pl
   "one?(dd) => error " cannot evaluate at place of degree greater than one"
   ee:=eval(toPolyRing2 f,pl)$BP
   retract ee

lBasis(divis)==
   ans:=lBasis(divis)$BP
   nn:=ans.num
   dd:=ans.den
   nnd:=[toPolyRing pol for pol in nn]
   ddd:=toPolyRing dd
   [nnd,ddd]

genus==genus()$BP

genusNeg==genusNeg()$BP

theCurve==
   ccc:= theCurve()$BP
   toPolyRing ccc

desingTree==desingTree()$BP

desingTreeWoFullParam== desingTreeWoFullParam()$BP

-- compute the adjunction divisor of the curve using
-- adjunctionDivisor from DesingTreePackage
adjunctionDivisor == adjunctionDivisor()$BP

singularPoints==singularPoints()$BP

parametrize(f,pl)==
   ff:= toPolyRing2 f
   parametrize(ff,pl)$BP

-- compute the interpolating forms (see package InterpolateFormsPackage)
interpolateForms(d,n)==
   ans:=interpolateForms(d,n)$BP
   [toPolyRing pol for pol in ans]

interpolateFormsForFact(d,lm)==
   lm2:List PolyRing2 := [ toPolyRing2 p for p in lm]
   interpolateFormsForFact(d,lm2)$BP

evalIfCan(ff:PolyRing,gg:PolyRing,pl:P1c)==
   dd:= degree pl
   "one?(dd) => error " cannot evaluate at place of degree greater than one"
   f:=toPolyRing2 ff
g:=toPolyRing2 gg
ee:=evalIfCan(f,g,pl)$BP
ee case "failed" => "failed"
retract ee

eval(ff:PolyRing,gg:PolyRing,pl:Plc)==
  dd:= degree pl
  "one?(dd) => error " cannot evaluate at place of degree greater than one"
  f:=toPolyRing2 ff
  g:=toPolyRing2 gg
  ee:=eval(f,g,pl)$BP
  retract ee

evalIfCan(u:FracPoly,pl:Plc)==
  ff:=numer u
  gg:=denom u
  evalIfCan(ff,gg,pl)

eval(u:FracPoly,pl:Plc)==
  ff:=numer u
  gg:=denom u
  eval(ff,gg,pl)

intersectionDivisor(pol)==
  polu:=toPolyRing2 pol
  intersectionDivisor(polu)$BP

fullDesTree==
  fullOutput()$DesTree => fullOutput(false())$DesTree
  fullOutput(true())$DesTree

fullInfClsPt==
  fullOutput()$InfClsPoint => fullOutput(false())$InfClsPoint
  fullOutput(true())$InfClsPoint
package PFORP PackageForPoly

--- PackageForPoly.input ---

)set break resume
)sys rm -f PackageForPoly.output
)spool PackageForPoly.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PackageForPoly
--R
--R PackageForPoly(R: Ring,PolyRing: FiniteAbelianMonoidRing(R,E),E: DirectProductCategory(dim,NonNegativeInteger),dim: NonNegativeInteger)
--R Abbreviation for PackageForPoly is PFORP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFORP
--R
--R------------------------------- Operations --------------------------------
--R constant : PolyRing -> R firstExponent : PolyRing -> E
--R listAllMonoExp : Integer -> List(E) listVariable : () -> List(PolyRing)
--R minimalForm : PolyRing -> PolyRing
--R degOneCoef : (PolyRing,PositiveInteger) -> R
--R degree : (PolyRing,Integer) -> NonNegativeInteger
--R degreeOfMinimalForm : PolyRing -> NonNegativeInteger
--R homogenize : (PolyRing,Integer) -> PolyRing
--R listAllMono : NonNegativeInteger -> List(PolyRing)
--R mapExponents : ((E -> E),PolyRing) -> PolyRing
--R monomials : PolyRing -> List(PolyRing)
--R replaceVarByOne : (PolyRing,Integer) -> PolyRing
--R replaceVarByZero : (PolyRing,Integer) -> PolyRing
--R subs1stVar : (PolyRing,PolyRing) -> PolyRing
--R subs2ndVar : (PolyRing,PolyRing) -> PolyRing
--R subsInVar : (PolyRing,PolyRing,Integer) -> PolyRing
--R totalDegree : PolyRing -> NonNegativeInteger
--R translate : (PolyRing,List(R),Integer) -> PolyRing
--R translate : (PolyRing,List(R)) -> PolyRing
--R univariate : PolyRing -> SparseUnivariatePolynomial(R)
--R
--E 1

)spool
)lisp (bye)

--- PackageForPoly.help ---
CHAPTER 17. CHAPTER P

PackageForPoly examples

The following is part of the PAFF package

See Also:
- )show PackageForPoly

PackageForPoly (PFORP)

- package PFORP PackageForPoly —

)abbrev package PFORP PackageForPoly
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
PackageForPoly(R,PolyRing,E,dim): public == private where
   R: Ring -- was Field but change for SolveTree package. 21/01/98
   dim: NonNegativeInteger
E: DirectProductCategory(dim,NonNegativeInteger)
PolyRing : FiniteAbelianMonoidRing(R,E)

Term ==> Record(k:E,c:R)
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
INT ==> Integer

public ==
mapExponents: (E->E, PolyRing) -> PolyRing
degree: (PolyRing , Integer) -> NNI
univariate: PolyRing -> SparseUnivariatePolynomial(R)
totalDegree: PolyRing -> NNI
subs1stVar: (PolyRing , PolyRing) -> PolyRing
subs2ndVar: (PolyRing , PolyRing) -> PolyRing
subsInVar: (PolyRing, PolyRing, Integer) -> PolyRing
minimalForm: PolyRing -> PolyRing
  ++ minimalForm(pol) returns the minimal forms of the polynomial pol.
firstExponent: PolyRing -> E
  ++ firstExponent(pol) returns the exponent of the first term in the
  ++ representation of pol. Not to be confused with the leadingExponent
  ++ which is the highest exponent according to the order
  ++ over the monomial.
replaceVarByZero: (PolyRing,Integer) -> PolyRing
  ++ replaceVarByZero(pol,a) evaluate to zero the variable in pol
  ++ specified by the integer a.
replaceVarByOne: (PolyRing,Integer) -> PolyRing
  ++ replaceVarByOne(pol,a) evaluate to one the variable in pol
  ++ specified by the integer a.
translate: (PolyRing,List R,Integer) -> PolyRing
  ++ translate(pol,[a,b,c],3) apply to pol the
  ++ linear change of coordinates, x->x+a, y->y+b, z->1.
translate: (PolyRing,List R) -> PolyRing
  ++ translate(pol,[a,b,c]) apply to pol the
  ++ linear change of coordinates, x->x+a, y->y+b, z->z+c
degOneCoef: (PolyRing,PI) -> R
  ++ degOneCoef(pol,n) returns the coefficient in front of the monomial
constant: PolyRing -> R
  ++ constant(pol) returns the constant term of the polynomial.

homogenize: (PolyRing,INT) -> PolyRing
  ++ homogenize(pol,n) returns the homogenized polynomial of pol
  ++ with respect to the n-th variable.

listAllMonoExp: Integer -> List E
  ++ listAllMonoExp(l) returns all the exponents of degree l

listAllMono: NNI -> List PolyRing
  ++ listAllMono(l) returns all the monomials of degree l

degreeOfMinimalForm: PolyRing -> NNI
  ++ degreeOfMinimalForm does what it says

listVariable: () -> List PolyRing

monomials: PolyRing -> List PolyRing

private == add
import PolyRing

monomials(pol)==
  zero? pol => empty()
  lt:=leadingMonomial pol
  cons( lt , monomials reductum pol )

lll: Integer -> E
lll(i) ==
  le:=new( dim , 0$NNI)$List(NNI)
  le.i := 1
  directProduct( vector(le)$Vector(NNI) )$E

listVariable==
  [monomial(1,ee)$PolyRing for ee in [lll(i) for i in 1..dim]]

univariate(pol)==
  zero? pol => 0
  d:=degree pol
  lc:=leadingCoefficient pol
  td := reduce("+", entries d)
  monomial(lc,td)$SparseUnivariatePolynomial(R)+univariate(reductum pol)

collectExpon: List Term -> PolyRing

translateLocal: (PolyRing,List R,Integer) -> PolyRing
lA: (Integer,Integer) -> List List NNI

toListRep: PolyRing -> List Term

exponentEntryToZero: (E,Integer) -> E

exponentEntryZero?: (E,Integer) -> Boolean

homogenizeExp: (E,NNI,INT) -> E

translateMonomial: (PolyRing,List R,INT,R) -> PolyRing

leadingTerm: PolyRing -> Term

mapExponents(f,pol)==
  zero?(pol) => 0
  lt:=leadingTerm pol
  newExp:E:= f(lt.k)
  newMono:PolyRing:= monomial(lt.c,newExp)$PolyRing
  newMono + mapExponents(f,reductum pol)

collectExpon(pol)==
  empty? pol => 0
  ft:=first pol
  monomial(ft.c,ft.k) + collectExpon( rest pol )

subs1stVar(pol, spol)==
  zero? pol => 0
  lexpE:E:= degree pol
  lexp:List NNI:= parts lexpE
  coef:= leadingCoefficient pol
  coef * spol ** lexp.1 * second(listVariable())**lexp.2 _
    + subs1stVar( reductum pol, spol )

subs2ndVar(pol, spol)==
  zero? pol => 0
  lexpE:E:= degree pol
  lexp:List NNI:= parts lexpE
  coef:= leadingCoefficient pol
  coef * first(listVariable())**lexp.1 * spol ** lexp.2 _
    + subs2ndVar( reductum pol, spol )

subsInVar( pol, spol, n)==
  one?( n ) => subs1stVar( pol, spol)
  subs2ndVar(pol,spol)

translate(pol,lpt)==
  zero? pol => 0
  lexpE:E:= degree pol
\( \text{exp} : \text{List \ NNI} := \text{parts} \text{\expE} \)
\( \text{coef} := \text{leadingCoefficient} \text{\pol} \)
\( \text{trVar} :=[(\text{listVariable().i + (lpt.i)::PolyRing}^{\text{\exp.i}} \text{for i in 1..dim}] \)
\( \text{coef} \ast \text{reduce}("\ast",\text{trVar},1) + \text{translate} (\text{reductum} \text{\pol} , \text{lpt}) \)

\( \text{translate} (\text{pol}, \text{lpt}, \text{nV}) == \)
\( \text{pol} := \text{replaceVarByOne} (\text{pol}, \text{nV}) \)
\( \text{translateLocal} (\text{pol}, \text{lpt}, \text{nV}) \)

\( \text{translateLocal} (\text{pol}, \text{lpt}, \text{nV}) == \)
\( \text{zero?}(\text{pol}) \Rightarrow 0 \)
\( \text{lll:List R} := [l \text{for l in lpt |} \text{\^zero?}(l)] \)
\( \text{nbOfNonZero} := \# \text{lll} \)
\( \text{ltk} := \text{leadingMonomial} \text{\pol} \)
\( \text{ltc} := \text{leadingCoefficient} \text{\pol} \)
\( \text{if one?}(\text{nbOfNonZero}) \text{then} \)
\( \text{pol} \)
\( \text{else} \)
\( \text{translateMonomial} (\text{ltk}, \text{lpt}, \text{nV}, \text{ltc}) + \)
\( \text{translateLocal} (\text{reductum} (\text{pol}), \text{lpt}, \text{nV}) \)

\( \text{exponentEntryToZero} (\text{\exp}, \text{nV}) == \)
\( \text{pexp} := \text{parts} \text{\exp} \)
\( \text{pexp(nV)} := 0 \)
\( \text{directProduct} (\text{vector(pexp)}$\text{Vector} (\text{NonNegativeInteger})) \)

\( \text{exponentEntryZero?} (\text{\exp}, \text{nV}) == \)
\( \text{pexp} := \text{parts} \text{\exp} \)
\( \text{zero?}(\text{pexp(nV)}) \)

\( \text{replaceVarByZero} (\text{pol}, \text{nV}) == \)
\( \text{replaceVarByOne} (\text{pol}, \text{nV}) == \)
\( \text{replaceVarByOne} (\text{pol}, \text{nV}) == \)

\( \text{homogenizeExp} (\text{\exp}, \text{\deg}, \text{nV}) == \)
\( \text{lv:List NNI} := \text{parts} (\text{exp}) \)
\( \text{lv.nV} := (\text{\deg} + \text{\lv.nV} \ast \text{reduce}("\ast",\text{lv})) \text{pretend NNI} \)
\( \text{directProduct} (\text{vector(lv)}$\text{Vector} (\text{NNI}))$E \)

\( \text{listTerm: PolyRing} \rightarrow \text{List E} \)
listTerm(pol) ==
  zero? pol => empty
  cons( degree pol, listTerm reductum pol )

degree( a : PolyRing, n : Integer ) ==
  zero? a => error "Degree for 0 is not defined for this degree fnc"
  "max" / [ ee.n for ee in listTerm a ]

totalDegree p ==
  zero? p => 0
  "max"/[reduce("+", t::(Vector NNI), 0) for t in listTerm p]

homogenize(pol,nV)==
  degP:=totalDegree(pol)
  mapExponents(homogenizeExp(#1,degP,nV),pol)

degOneCoef(p:PolyRing,i:PI)==
  vv:=new(dim,0)$Vector(NNI)
  vv.i:=1
  pd:=directProduct(vv)$E
  lp:=toListRep p
  lc:=[t.c for t in lp | t.k=pd]
  reduce("+",lc,0)

constant(p)==
  vv:=new(dim,0)$Vector(NNI)
  pd:=directProduct(vv)$E
  lp:=toListRep p
  lc:=[t.c for t in lp | t.k=pd]
  reduce("+",lc,0)

degreeOfMinimalForm(pol)==
  totalDegree minimalForm pol

minimalForm(pol)==
  zero?(pol) => pol
  lpol:=toListRep pol
  actTerm:Term:= first lpol
  minDeg:NNI:=reduce("+", parts(actTerm.k))
  actDeg:NNI
  lminForm:List(Term):= [actTerm]
  for p in rest(lpol) repeat
    actDeg:= reduce("+", parts(p.k))
    if actDeg = minDeg then
      lminForm := concat(lminForm,p)
    if actDeg < minDeg then
      minDeg:=actDeg
      lminForm:=[p]
  collectExpon lminForm
-- le code de collectExponSort a ete emprunte a D. Augot.

leadingTerm(pol)==
    zero?(pol) => error "no leading term for 0 (message from package)"

lcoef:R:=leadingCoefficient(pol)$PolyRing
lterm:PolyRing:=leadingMonomial(pol)$PolyRing


listAllMono(l)==
    [monomial(1,e)$PolyRing for e in listAllMonoExp(l)]
PACKAGE PADEPAC PADEAPPROXIMANTPACKAGE

3239

package PADEPAC PadeApproximantPackage
— PadeApproximantPackage.input —
)set break resume
)sys rm -f PadeApproximantPackage.output
)spool PadeApproximantPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PadeApproximantPackage
--R
--R PadeApproximantPackage(R: Field,x: Symbol,pt: R) is a package constructor
--R Abbreviation for PadeApproximantPackage is PADEPAC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PADEPAC
--R
--R------------------------------- Operations ---------------------------------R pade : (NonNegativeInteger,NonNegativeInteger,UnivariateTaylorSeries(R,x,pt),UnivariateTaylorSeries(R,x,
--R pade : (NonNegativeInteger,NonNegativeInteger,UnivariateTaylorSeries(R,x,pt)) -> Union(Fraction(Univaria
--R
--E 1
)spool
)lisp (bye)

———— PadeApproximantPackage.help —
====================================================================
PadeApproximantPackage examples
====================================================================
This package computes reliable Pad&ea. approximants using a
generalized Viskovatov continued fraction algorithm.
See Also:
o )show PadeApproximantPackage

———-


PadeApproximantPackage (PADEPAC)

Exports:
pade

— package PADEPAC PadeApproximantPackage —

)abbrev package PADEPAC PadeApproximantPackage
++ Authors: Trager,Burge, Hassner & Watt.
++ Date Created: April 1987
++ Date Last Updated: 12 April 1990
++ References:
++ George A. Baker and Peter Graves-Morris
++ ‘‘Pade Approximants’’
++ Description:
++ This package computes reliable Padé approximants using
++ a generalized Viskovatov continued fraction algorithm.

PadeApproximantPackage(R: Field, x:Symbol, pt:R): Exports == Implementation where
PS ==> UnivariateTaylorSeries(R,x,pt)
UP ==> UnivariatePolynomial(x,R)
QF ==> Fraction UP
CF ==> ContinuedFraction UP
NNI ==> NonNegativeInteger
Exports == with
pade: (NNI,NNI,PS,PS) -> Union(QF,"failed")
++ pade(nd,dd,ns,ds) computes the approximant as a quotient of polynomials
++ (if it exists) for arguments
++ nd (numerator degree of approximant),
++ dd (denominator degree of approximant),
++ ns (numerator series of function), and
++ ds (denominator series of function).
pade: (NNI,NNI,PS) -> Union(QF,"failed")
++ pade(nd,dd,s)
++ computes the quotient of polynomials
++ (if it exists) with numerator degree at
++ most nd and denominator degree at most dd
++ which matches the series s to order \spad{nd + dd}.

\begin{verbatim}
Implementation ==> add
n,m : NNI
u,v : PS
pa := PadeApproximants(R,PS,UP)
pade(n,m,u,v) ==
    ans:=pade(n,m,u,v)$pa
    ans case "failed" => ans
    pt = 0 => ans
    num := numer(ans::QF)
    den := denom(ans::QF)
    xpt : UP := monomial(1,1)-monomial(pt,0)
    num := num(xpt)
    den := den(xpt)
    num/den
    pade(n,m,u) == pade(n,m,u,1)
\end{verbatim}

| PADEPAC.dotabb |
"PADEPAC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PADEPAC"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PADEPAC" -> "PFECAT"

package PADE PadeApproximants

| PadeApproximants.input |
)set break resume
)sys rm -f PadeApproximants.output
)spool PadeApproximants.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PadeApproximants

--R
--R PadeApproximants(R: Field,PS: UnivariateTaylorSeriesCategory(R),UP: UnivariatePolynomialCategory(R)) in
--R Abbreviation for PadéApproximants is PADE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PADE
--R
--R------------------------------- Operations --------------------------------
--R pade : (NonNegativeInteger,NonNegativeInteger,PS,PS) -> Union(Fraction(UP),"failed")
--R padecf : (NonNegativeInteger,NonNegativeInteger,PS,PS) -> Union(ContinuedFraction(UP),"failed")
--E 1

)spool
)lisp (bye)

— PadeApproximants.help —

====================================================================
PadeApproximants examples
====================================================================

See Also:
o )show PadeApproximants

PadeApproximants (PADE)

Exports:
pade  padecf

— package PADE PadeApproximants —


```lisp
)abbrev package PADE PadeApproximants
++ Authors: Burge, Hassner & Watt.
++ Date Created: April 1987
++ Date Last Updated: 12 April 1990
++ References:
++ George A. Baker and Peter Graves-Morris
++ ‘‘Pade Approximants’’
++ Description:
++ This package computes reliable Padé approximants using
++ a generalized Viskovatov continued fraction algorithm.

PadeApproximants(R,PS,UP): Exports == Implementation where
  R: Field -- IntegralDomain
  PS: UnivariateTaylorSeriesCategory R
  UP: UnivariatePolynomialCategory R

  NNI ==> NonNegativeInteger
  QF ==> Fraction UP
  CF ==> ContinuedFraction UP

Exports ==> with
  pade: (NNI,NNI,PS,PS) -> Union(QF,"failed")
    ++ pade(nd,dd,ns,ds)
    ++ computes the approximant as a quotient of polynomials
    ++ (if it exists) for arguments
    ++ nd (numerator degree of approximant),
    ++ dd (denominator degree of approximant),
    ++ ns (numerator series of function), and
    ++ ds (denominator series of function).
  padecf: (NNI,NNI,PS,PS) -> Union(CF,"failed")
    ++ padecf(nd,dd,ns,ds)
    ++ computes the approximant as a continued fraction of
    ++ polynomials (if it exists) for arguments
    ++ nd (numerator degree of approximant),
    ++ dd (denominator degree of approximant),
    ++ ns (numerator series of function), and
    ++ ds (denominator series of function).

Implementation ==> add
  -- The approximant is represented as
  -- p0 + x**(a1)/(p1 + x**(a2)/(...))

  PadeRep ==> Record(ais: List UP, degs: List NNI) -- #ais= #degs
  PadeU ==> Union(PadeRep,"failed") -- #ais= #degs+1

  constInner(up:UP):PadeU == [[up], []]

  truncPoly(p:UP,n:NNI):UP ==
    while n < degree p repeat p := reductum p
```

p

truncSeries(s:PS,n:NNI):UP ==
  p: UP := 0
  for i in 0..n repeat p := p + monomial(coefficient(s,i),i)
  p

-- Assumes s starts with a<x><x><n> + ... and divides out x<n>.  
divOutDegree(s:PS,n:NNI):PS ==
  for i in 1..n repeat s := quoByVar s  
s
padeNormalize: (NNI,NNI,PS,PS) -> PadeU
padeInner: (NNI,NNI,PS,PS) -> PadeU

pade(l,m,gps,dps) ==
  (ad := padeNormalize(l,m,gps,dps)) case "failed" => "failed"
  plist := ad.ais; dlist := ad.degs
  approx := first(plist) :: QF
  for d in dlist for p in rest plist repeat
    approx := p::QF + (monomial(1,d)$UP :: QF)/approx
  approx

padecf(l,m,gps,dps) ==
  (ad := padeNormalize(l,m,gps,dps)) case "failed" => "failed"
  alist := reverse(ad.ais)
  blist := [monomial(1,d)$UP for d in reverse ad.degs]
  continuedFraction(first(alist),_.
    blist::Stream UP,(rest alist) :: Stream UP)

padeNormalize(l,m,gps,dps) ==
  zero? dps => "failed"
  zero? gps => constInner 0
  -- Normalize so numerator or denominator has constant term.
  ldeg:= min(order dps,order gps)
  if ldeg > 0 then
    dps := divOutDegree(dps,ldeg)
    gps := divOutDegree(gps,ldeg)
    padeInner(l,m,gps,dps)

padeInner(l, m, gps, dps) ==
  zero? coefficient(gps,0) and zero? coefficient(dps,0) =>
    error "Pade' problem not normalized."
  plist: List UP := nil()
  alist: List NNI := nil()
  -- Ensure denom has constant term.
  if zero? coefficient(dps,0) then
    -- g/d = 0 + z**0/(d/g)
    (gps,dps) := (dps,gps)
    (l,m) := (m,l)
    padeInner(l, m, gps, dps)
plist := concat(0,plist)
alist := concat(0,alist)

-- Ensure l >= m, maintaining coef(dps,0)^=0.
if l < m then
  -- (a<n>*x**n + a<n+1>*x**n+1 + ...)/b
  -- = x**n/b + (a<n> + a<n+1>*x + ...)/b
  alpha := order gps
  if alpha > l then return "failed"
  gps := divOutDegree(gps, alpha)
  (l,m) := (m,(l-alpha) :: NNI)
  (gps,dps) := (dps,gps)
  plist := concat(0,plist)
  alist := concat(alpha,alist)

degbd: NNI := l + m + 1
for j in 0.. repeat
  -- Normalize d so constant coefs cancel. (B&G-M is wrong)
  d0 := coefficient(d,0)
  d := (1/d0) * d; g := (1/d0) * g
  p : UP := 0; s := g
  for k in 1..degbd repeat
    pk := coefficient(s,0)
    p := p + monomial(pk,(k-1) :: NNI)
    s := s - pk*d
    s := (s exquo monomial(1,1)) :: UP
  plist := concat(p,plist)
  s = 0 => return [plist,alist]
  alpha := minimumDegree(s) + degbd
  alpha > l + m => return [plist,alist]
  alpha > l => return "failed"
  alist := concat(alpha,alist)
  h := (s exquo monomial(1,minimumDegree s)) :: UP
  degbd := (1 + m - alpha) :: NNI
  g := truncPoly(d,degbd)
  d := truncPoly(h,degbd)
  (l,m) := (m,(l-alpha) :: NNI)

"PADE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PADE"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"PADE" -> "UTSCAT"
package PWFFINTB PAdicWildFunctionFieldIntegralBasis

-- PAdicWildFunctionFieldIntegralBasis.input --

)set break resume
)sys rm -f PAdicWildFunctionFieldIntegralBasis.output
)spool PAdicWildFunctionFieldIntegralBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PAdicWildFunctionFieldIntegralBasis
--R PAdicWildFunctionFieldIntegralBasis(K: FiniteFieldCategory,R: UnivariatePolynomialCategory(K),UP: UnivariatePolynomialCategory(R),F: MonogenicAlgebra(R,UP)) is a package constructor
--R Abbreviation for PAdicWildFunctionFieldIntegralBasis is PWFFINTB
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PWFFINTB
--R
--R------------------------------- Operations --------------------------------
--R reducedDiscriminant : UP -> R
--R integralBasis : () -> Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R))
--R localIntegralBasis : R -> Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R))
--R
--E 1

)spool
)lisp (bye)

-- PAdicWildFunctionFieldIntegralBasis.help --

====================================================================
PAdicWildFunctionFieldIntegralBasis examples
====================================================================

In this package K is a finite field, R is a ring of univariate polynomials over K, and F is a monogenic algebra over R.

We require that F is monogenic, i.e. that F = K[x,y]/(f(x,y)), because the integral basis algorithm used will factor the polynomial f(x,y). The package provides a function to compute the integral
closure of \( R \) in the quotient field of \( F \) as well as a function to compute a "local integral basis" at a specific prime.

See Also:
- )show PAdicWildFunctionFieldIntegralBasis

---

**PAdicWildFunctionFieldIntegralBasis (PWFFINTB)**

Exports:
- reducedDiscriminant
- integralBasis
- localIntegralBasis

---

)abbrev package PWFFINTB PAdicWildFunctionFieldIntegralBasis
++ Author: Clifton Williamson
++ Date Created: 5 July 1993
++ Date Last Updated: 17 August 1993
++ Description:
++ In this package \( K \) is a finite field, \( R \) is a ring of univariate
++ polynomials over \( K \), and \( F \) is a monogenic algebra over \( R \).
++ We require that \( F \) is monogenic, i.e. that \( \text{spad}(F = K[x,y]/(f(x,y))) \),
++ because the integral basis algorithm used will factor the polynomial
++ \( \text{spad}(f(x,y)) \). The package provides a function to compute the integral
++ closure of \( R \) in the quotient field of \( F \) as well as a function to compute
++ a "local integral basis" at a specific prime.

PAdicWildFunctionFieldIntegralBasis(K,R,UP,F): Exports == Implementation where
K : FiniteFieldCategory
R : UnivariatePolynomialCategory K
UP : UnivariatePolynomialCategory R
F : MonogenicAlgebra(R,UP)
I ==> Integer
L ==> List
L2 ==> ListFunctions2
Mat ==> Matrix R
NNI ==> NonNegativeInteger
PI ==> PositiveInteger
Q ==> Fraction R
SAE ==> SimpleAlgebraicExtension
SUP ==> SparseUnivariatePolynomial
CDEN ==> CommonDenominator
DDFACT ==> DistinctDegreeFactorize
WFFINTBS ==> WildFunctionFieldIntegralBasis
Result ==> Record(basis: Mat, basisDen: R, basisInv: Mat)
IResult ==> Record(basis: Mat, basisDen: R, basisInv: Mat, discr: R)
IBPTOOLS ==> IntegralBasisPolynomialTools
IBACHIN ==> ChineseRemainderToolsForIntegralBases
IRREDFFX ==> IrredPolyOverFiniteField
GHEN ==> GeneralHenselPackage

Exports ==> with

integralBasis : () -> Result
++ \spad{integralBasis()} returns a record
++ \spad{
[basis,basisDen,basisInv]  }
++ containing information regarding
++ the integral closure of \spad{R} in the quotient field of the framed
++ algebra \spad{F}. \spad{F} is a framed algebra with \spad{R}-module basis
++ \spad{\{w_1,w_2,...,w_n\}}.
++ If 'basis' is the matrix \spad{\{a_{ij}, i = 1..n, j = 1..n\}}, then
++ the \spad{i}th element of the integral basis is
++ \spad{v_i = (1/basisDen) * \sum(a_{ij} * w_j, j = 1..n)}, i.e. the
++ \spad{i}th row of 'basis' contains the coordinates of the
++ \spad{i}th basis vector. Similarly, the \spad{i}th row of the
++ matrix 'basisInv' contains the coordinates of \spad{\{v_1,...,v_n\}} with respect
++ to the basis \spad{\{w_1,...,w_n\}}: if 'basisInv' is the matrix
++ \spad{\{b_{ij}, i = 1..n, j = 1..n\}}, then
++ \spad{w_i = \sum(b_{ij} * v_j, j = 1..n)}.

localIntegralBasis : R -> Result
++ \spad{localIntegralBasis(p)} returns a record
++ \spad{\{basis,basisDen,basisInv\}} containing information regarding
++ the local integral closure of \spad{R} at the prime \spad{p} in the quotient
++ field of the framed algebra \spad{F}. \spad{F} is a framed algebra with \spad{R}-module
++ basis \spad{\{w_1,w_2,...,w_n\}}.
++ If 'basis' is the matrix \spad{\{a_{ij}, i = 1..n, j = 1..n\}}, then
++ the \spad{i}th element of the local integral basis is
++ \spad{v_i = (1/basisDen) * \sum(a_{ij} * w_j, j = 1..n)}, i.e. the
++ \spad{i}th row of 'basis' contains the coordinates of the
++ \spad{i}th basis vector. Similarly, the \spad{i}th row of the
++ matrix 'basisInv' contains the coordinates of \spad{\{w_1,...,w_n\}} with respect
++ to the basis \spad{\{v_1,...,v_n\}}: if 'basisInv' is the matrix
++ \spad{\{b_{ij}, i = 1..n, j = 1..n\}}, then
++ \spad{wi = \sum(bij \ast vj, j = 1..n)}.
reducedDiscriminant: UP -> R
++ reducedDiscriminant(up) \ undocumented

Implementation ==> add
import IntegralBasisTools(R, UP, F)
import GeneralHenselPackage(R,UP)
import ModularHermitianRowReduction(R)
import TriangularMatrixOperations(R, Vector R, Vector R, Matrix R)

reducedDiscriminant f ==
  ff : SUP Q := mapUnivariate((r1:R):Q+->r1 :: Q,f)$IBPTOOLS(R,UP,SUP UP,Q)
  ee := extendedEuclidean(ff,differentiate ff)
  cc := concat(coefficients(ee.coef1),coefficients(ee.coef2))
  cden := splitDenominator(cc)$CDEN(R,Q,L Q)
  denom := cden.den
  gg := gcd map(numer,cden.num)$L2(Q,R)
  (ans := denom exquo gg) case "failed" =>
    error "PWFFINTB: error in reduced discriminant computation"
  ans :: R

compLocalBasis: (UP,R) -> Result
compLocalBasis(poly,prime) ==
  -- compute a local integral basis at 'prime' for k[x,y]/(poly(x,y)).
  sae := SAE(R,UP,poly)
  localIntegralBasis(prime)$WFFINTBS(K,R,UP,sae)

compLocalBasisOverExt: (UP,R,UP,NNI) -> Result
compLocalBasisOverExt(poly0,prime0,irrPoly0,k) ==
  -- poly0 = irrPoly0**k (mod prime0)
  n := degree poly0; disc0 := discriminant poly0
  (disc0 exquo prime0) case "failed" =>
    [scalarMatrix(n,1), 1, scalarMatrix(n,1)]
  r := degree irrPoly0
  -- extend scalars:
  -- construct irreducible polynomial of degree r over K
  irrPoly := generateIrredPoly(r :: PI)$IRREDFFX(K)
  -- construct extension of degree r over K
  E := SAE(K,SUP K,irrPoly)
  -- lift coefficients to elements of E
  poly := mapBivariate((k1:K):E +-> k1::E,poly0)$IBPTOOLS(K,R,UP,E)
  redDisc0 := reducedDiscriminant poly0
  redDisc := mapUnivariate((k1:K):E +-> k1::E,redDisc0)$IBPTOOLS(K,R,UP,E)
  prime := mapUnivariate((k1:K):E +-> k1::E,prime0)$IBPTOOLS(K,R,UP,E)
  sae := SAE(E,SUP E,prime)
  -- reduction (mod prime) of polynomial of which poly is the kth power
  redIrрPoly :=
    pp := mapBivariate((k1:K):E +-> k1::E,irrPoly0)$IBPTOOLS(K,R,UP,E)
    mapUnivariate(reduce,pp)$IBPTOOLS(SUP E,SUP E,SUP E,SUP E,sae)
  -- factor the reduction
factorListSAE := factors factor(redIrrPoly)$DDFACT(sae,SUP sae)

-- list the 'primary factors' of the reduction of poly
redFactors : List SUP sae := [(f.factor)**k for f in factorListSAE]

-- lift these factors to elements of SUP SUP E
primaries : List SUP SUP E :=
    [mapUnivariate(lift,ff)$IBPTOOLS(SUP E,SUP SUP E,SUP SUP SUP E,sae) _
     for ff in redFactors]

-- lift the factors to factors modulo a suitable power of 'prime'
deg := (1 + order(redDisc,prime) * degree(prime)) :: PI

henselInfo := HenselLift(poly,primaries,prime,deg)$GHEN(SUP E,SUP SUP E)
henselFactors := henselInfo.plist

psi1 := first henselFactors

FF := SAE(SUP E,SUP SUP E,psi1)

factorIb := localIntegralBasis(prime)$WFFINTBS(E,SUP E,SUP SUP E,FF)

bs := listConjugateBases(factorIb,size()$K,r)$IBACHIN(E,SUP E,SUP SUP E)

ib := chineseRemainder(henselFactors,bs,n)$IBACHIN(E,SUP E,SUP SUP E)

b : Matrix R :=
    bas := mapMatrixIfCan(retractIfCan,ib.basis)$IBPTOOLS(K,R,UP,E)
    bas case "failed" => error "retraction of basis failed"
    bas :: Matrix R

bInv : Matrix R :=
    --bas := mapMatrixIfCan(ric,ib.basisInv)$IBPTOOLS(K,R,UP,E)
    bas := mapMatrixIfCan(retractIfCan,ib.basisInv)$IBPTOOLS(K,R,UP,E)
    bas case "failed" => error "retraction of basis inverse failed"
    bas :: Matrix R

bDen : R :=
    p := mapUnivariateIfCan(retractIfCan,ib.basisDen)$IBPTOOLS(K,R,UP,E)
    p case "failed" => error "retraction of basis denominator failed"
    p :: R

[b,bDen,bInv]

padicLocalIntegralBasis: (UP,R,R,R) -> IResult

padicLocalIntegralBasis(p,disc,redDisc,prime) ==

    -- polynomials in x modulo 'prime'
sae := SAE(K,R,prime)

    -- find the factorization of 'p' modulo 'prime' and lift the
    -- prime powers to elements of UP:
    -- reduce 'p' modulo 'prime'
    reducedP := mapUnivariate(reduce,p)$IBPTOOLS(R,UP,SUP UP,sae)

    -- factor the reduced polynomial
    factorListSAE := factors factor(reducedP)$DDFACT(sae,SUP sae)

    -- if only one prime factor, perform usual integral basis computation
    (factorListSAE) = 1 =>
        ib := localIntegralBasis(prime)$WFFINTBS(K,R,UP,F)
        index := diagonalProduct(ib.basisInv)
        [ib.basis,ib.basisDen,ib.basisInv,disc quo (index * index)]

    -- list the 'prime factors' of the reduced polynomial
    redPrimes : List SUP sae :=
        [f.factor for f in factorListSAE]

    -- lift these factors to elements of UP
primes : List UP :=
    [mapUnivariate(lift,ff)$IBPTOOLS(R,UP,SUP UP,sae) for ff in redPrimes]
-- list the exponents
expons : List NNI := [(f.exponent) :: NNI) for f in factorListSAE]
-- list the 'primary factors' of the reduced polynomial
redPrimitives : List SUP sae :=
    [(f.factor) **((f.exponent) :: NNI) for f in factorListSAE]
-- lift these factors to elements of UP
primaries : List UP :=
    [mapUnivariate(lift,ff)$IBPTOOLS(R,UP,SUP UP,sae) for ff in redPrimaries]
-- lift the factors to factors modulo a suitable power of 'prime'
deg := (1 + order(redDisc,prime) * degree(prime)) :: PI
henselInfo := HenselLift(p,primaries,prime,deg)
henselFactors := henselInfo.plist
-- compute integral bases for the factors
factorBases : List Result := empty(); degPrime := degree prime
for pp in primes for k in expons for qq in henselFactors repeat
    base :=
        degPp := degree pp
        degPp > 1 and gcd(degPp,degPrime) = 1 =>
            compLocalBasisOverExt(qq,prime,pp,k)
        compLocalBasis(qq,prime)
    factorBases := concat(base,factorBases)
factorBases := reverse_! factorBases
ib := chineseRemainder(henselFactors,factorBases,rank()$F)$IBACHIN(K,R,UP)
index := diagonalProduct(ib.basisInv)
[ib.basis,ib.basisDen,ib.basisInv,disc quo (index * index)]

localIntegralBasis prime ==
p := definingPolynomial()$F; disc := discriminant p
disc := determinant traceMatrix()$F
redDisc := reducedDiscriminant p
ib := padicLocalIntegralBasis(p,disc,redDisc,prime)
[ib.basis,ib.basisDen,ib.basisInv]

listSquaredFactors: R -> List R
listSquaredFactors px ==
    -- returns a list of the factors of px which occur with
    -- exponent > 1
    ans : List R := empty()
    factored := factor(px)$DistinctDegreeFactorize(K,R)
    for f in factors(factored) repeat
        if f.exponent > 1 then ans := concat(f.factor,ans)
    ans

integralBasis() ==
p := definingPolynomial()$F; disc := discriminant p; n := rank()$F
--traceMat := traceMatrix()$F; n := rank()$F
--disc := determinant traceMat -- discriminant of current order
singList := listSquaredFactors disc -- singularities of relative Spec
\[ \text{redDisc} := \text{reducedDiscriminant\ p} \]
\[ \text{runningRb} := \text{runningRbInv} := \text{scalarMatrix\(n,1\)} \]

---

\[ \text{runningRb} = \text{basis matrix of current order} \]
\[ \text{runningRbInv} = \text{inverse basis matrix of current order} \]
---

\[ \text{these are wrt the original basis for F} \]
\[ \text{runningRbden} : R := 1 \]
---

\[ \text{runningRbden} = \text{denominator for current basis matrix} \]
\[ \text{empty?\ singList \Rightarrow [runningRb, runningRbden, runningRbInv]} \]

for prime in singList repeat

\[ \text{lb} := \text{padicLocalIntegralBasis\(p,disc,\text{redDisc},\text{prime}\)} \]
\[ \text{rb} := \text{lb.basis}; \text{rbinv} := \text{lb.basisInv}; \text{rbden} := \text{lb.basisDen} \]
\[ \text{disc} := \text{lb.dscr} \]
\[ \text{mat} := \text{vertConcat}\(\text{rbden} \ast \text{runningRb}, \text{runningRbden} \ast \text{rb}\) \]
\[ \text{runningRbden} := \text{runningRbden} \ast \text{rbden} \]
\[ \text{runningRb} := \text{squareTop\ rowEchelon(mat, runningRbden)} \]
\[ \text{runningRbInv} := \text{UpTriBddDenomInv(runningRb, runningRbden)} \]

\[ [\text{runningRb, runningRbden, runningRbInv}] \]

---

— PWFFINTB.dotabb —

"PWFFINTB" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PWFFINTB"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"PWFFINTB" -> "MONOGEN"

---

package YSTREAM ParadoxicalCombinatorsForStreams

— ParadoxicalCombinatorsForStreams.input —

\)
\)set break resume
\)sys rm -f ParadoxicalCombinatorsForStreams.output
\)spool ParadoxicalCombinatorsForStreams.output
\)set message test on
\)set message auto off
\)clear all

--S 1 of 1
\)show ParadoxicalCombinatorsForStreams

--R

--R ParadoxicalCombinatorsForStreams(A: Type) is a package constructor
---R Abbreviation for ParadoxicalCombinatorsForStreams is YSTREAM
---R This constructor is not exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for YSTREAM
---R
---R----------------------------------- Operations --------------------------------
---R Y : (Stream(A) -> Stream(A)) -> Stream(A)
---R Y : ((List(Stream(A)) -> List(Stream(A))),Integer) -> List(Stream(A))
---R
---E 1

)spool
)lisp (bye)

---

--- ParadoxicalCombinatorsForStreams.help ---

====================================================================
ParadoxicalCombinatorsForStreams examples
====================================================================

Computation of fixed points of mappings on streams

See Also:
o )show ParadoxicalCombinatorsForStreams

---

ParadoxicalCombinatorsForStreams (YSTREAM)

Exports:
Y
package YSTREAM ParadoxicalCombinatorsForStreams —

)abbrev package YSTREAM ParadoxicalCombinatorsForStreams
++ Author: Burge, Watt (revised by Williamson)
++ Date Created: 1986
++ Date Last Updated: 21 October 1989
++ Description:
++ Computation of fixed points of mappings on streams

ParadoxicalCombinatorsForStreams(A):Exports == Implementation where
  ++ This package implements fixed-point computations on streams.
  + A : Type
  + ST ==> Stream
  + L ==> List
  + I ==> Integer

Exports ==> with
  Y: (ST A -> ST A) -> ST A
    ++ Y(f) computes a fixed point of the function f.
  Y: (L ST A -> L ST A,I) -> L ST A
    ++ Y(g,n) computes a fixed point of the function g, where g takes
    ++ a list of n streams and returns a list of n streams.

Implementation ==> add

Y f ==
  y : ST A := CONS(0$I,0$I)$Lisp
  j := f y
  RPLACA(y,frst j)$Lisp
  RPLACD(y,rst j)$Lisp
  y

Y(g,n) ==
  x : L ST A := [CONS(0$I,0$I)$Lisp for i in 1..n]
  j := g x
  for xi in x for ji in j repeat
    RPLACA(xi,frst ji)$Lisp
    RPLACD(xi,rst ji)$Lisp
  x

— YSTREAM.dotabb —

"YSTREAM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=YSTREAM"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"YSTREAM" -> "TYPE"
package PLEQN ParametricLinearEquations

-- ParametricLinearEquations.input --

)set break resume
)sys rm -f ParametricLinearEquations.output
)spool ParametricLinearEquations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricLinearEquations

--R ParametricLinearEquations(R: Join(EuclideanDomain,CharacteristicZero),Var: Join(OrderedSet,ConvertibleTo(Symbol)),Expon: OrderedAbelianMonoidSup,GR: PolynomialCategory(R,Expon,Var)) is a package constructor

--R Abbreviation for ParametricLinearEquations is PLEQN

--R This constructor is not exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PLEQN

--R----------------------------- Operations --------------------------------

--R factorset : GR -> List(GR) inconsistent? : List(GR) -> Boolean
--R pr2dmp : Polynomial(R) -> GR sqfree : GR -> GR
--R B1solve : Record(mat: Matrix(Fraction(Polynomial(R))),vec: List(Fraction(Polynomial(R))),rank: NonNegativeInteger) -> Record(partsol: Vector(Fraction(Polynomial(R))),basis: List(Vector(Fraction(Polynomial(R))))))
--R ParCond : (Matrix(GR),NonNegativeInteger) -> List(Record(det: GR,rows: List(Integer),cols: List(Integer)))
--R ParCondList : (Matrix(GR),NonNegativeInteger) -> List(Record(rank: NonNegativeInteger,eqns: List(Record(det: GR,rows: List(Integer),cols: List(Integer))),fgb: List(GR)))
--R bsolve : (Matrix(GR),List(Fraction(Polynomial(R))),NonNegativeInteger,String,Integer) -> Record(rgl: List(List(Integer)),rgsz: Integer))
--R dmp2rfi : GR -> Fraction(Polynomial(R))
--R dmp2rfi : Matrix(GR) -> Matrix(Fraction(Polynomial(R)))
--R dmp2rfi : List(GR) -> List(Fraction(Polynomial(R)))
--R hasoln : (List(GR),List(GR)) -> Record(sysok: Boolean,z0: List(GR),n0: List(GR))
--R inconsistent? : List(Polynomial(R)) -> Boolean
--R maxrank : List(Record(rank: NonNegativeInteger,eqns: List(Record(det: GR,rows: List(Integer),cols: List(Integer))),fgb: List(GR))) -> NonNegativeInteger
--R minrank : List(Record(rank: NonNegativeInteger,eqns: List(Record(det: GR,rows: List(Integer),cols: List(Integer))),fgb: List(GR))) -> NonNegativeInteger
--R minset : List(List(GR)) -> List(List(GR))
--R nextSublist : (Integer,Integer) -> List(List(Integer))
--R overset? : (List(GR),List(List(GR))) -> Boolean
--R psolve : (Matrix(GR),List(GR)) -> List(Record(eqzro: List(GR),neqzro: List(GR),wcond: List(Polynomial(R))))
--R psolve : (Matrix(GR),List(Symbol)) -> List(Record(eqzro: List(GR),neqzro: List(GR),wcond: List(Polynomial(R))))
--R psolve : Matrix(GR) -> List(Record(eqzro: List(GR),neqzro: List(GR),wcond: List(Polynomial(R))))
--R psolve : (Matrix(GR),List(GR),PositiveInteger) -> List(Record(eqzro: List(GR),neqzro: List(GR),wcond: List(Polynomial(R))))
--R psolve : (Matrix(GR),List(Symbol),PositiveInteger) -> List(Record(eqzro: List(GR),neqzro: List(GR),wcond: List(Polynomial(R))))
--R psolve : (Matrix(GR),PositiveInteger) -> List(Record(eqzro: List(GR),neqzro: List(GR),wcond: List(Polynomial(R))))
--R psolve : (Matrix(GR),List(GR),String) -> Integer
This package completely solves a parametric linear system of equations by decomposing the set of all parametric values for which the linear system is consistent into a union of quasi-algebraic sets (which need not be irredundant, but most of the time is). Each quasi-algebraic set is described by a list of polynomials that vanish on the set, and a list of polynomials that vanish at no point of the set. For each quasi-algebraic set, the solution of the linear system is given, as a particular solution and a basis of the homogeneous system.

The parametric linear system should be given in matrix form, with a coefficient matrix and a right hand side vector. The entries of the coefficient matrix and right hand side vector should be polynomials in the parametric variables, over a Euclidean domain of characteristic zero.

If the system is homogeneous, the right hand side need not be given. The right hand side can also be replaced by an indeterminate vector, in which case, the conditions required for consistency will also be given.

The package has other facilities for saving results to external files, as well as solving the system for a specified minimum rank. Altogether there are 12 mode maps for psolve, as explained below.

See Also:
o )show ParametricLinearEquations
ParametricLinearEquations (PLEQN)

Exports:
bsolve B1solve dmp2rfi dmp2rfi dmp2rfi
tfactorset hasoln inconsistent? maxrank minrank
minset nextSublist overset? ParCond ParCondList
pr2dmp psolve rdregime redmat redpps
regime se2rfi sqfree wrregime

-- modified to conform with new runtime system 06/04/90
-- updated with comments for MB, 02/16/94
-- also cleaned up some unnecessary arguments in regime routine
--
-- MB: In order to allow the rhs to be indeterminate, while working
-- mainly with the parametric variables on the lhs (less number of
-- variables), certain conversions of internal representation from
-- GR to Polynomial R and Fraction Polynomial R are done. At the time
-- of implementation, I thought that this may be more efficient. I
-- have not done any comparison since that requires rewriting the
-- package. My own application needs to call this package quite often,
-- and most computations involves only polynomials in the parametric
-- variables.

-- The 12 modes of psolve are probably not all necessary. Again, I
-- was thinking that if there are many regimes and many ranks, then
-- the output is quite big, and it may be nice to be able to save it
-- and read the results in later to continue computing rather than
-- recomputing. Because of the combinatorial nature of the algorithm
-- (computing all subdeterminants!), it does not take a very big matrix
-- to get into many regimes. But I now have second thoughts of this
design, since most of the time, the results are just intermediate,
passed to be further processed. On the other hand, there is probably
no penalty in leaving the options as is.

— package PLEQN ParametricLinearEquations —

)}

parametric linear system should be given in matrix form, with
+ a coefficient matrix and a right hand side vector. The entries
+ of the coefficient matrix and right hand side vector should be
+ polynomials in the parametric variables, over a Euclidean domain
+ of characteristic zero.

++ If the system is homogeneous, the right hand side need not be given.
++ The right hand side can also be replaced by an indeterminate vector,
++ in which case, the conditions required for consistency will also be
++ given.
++ The package has other facilities for saving results to external
++ files, as well as solving the system for a specified minimum rank.
++ Altogether there are 12 mode maps for psolve, as explained below.

ParametricLinearEquations(R,Var,Expon,GR):
  Declaration == Definition where

  R : Join(EuclideanDomain, CharacteristicZero)
  Var : Join(OrderedSet,ConvertibleTo (Symbol))
  Expon : OrderedAbelianMonoidSup
  GR : PolynomialCategory(R,Expon,Var)
  F == Fraction R
  FILE ==> FileCategory
  FNAME ==> FileName
  GB ==> EuclideanGroebnerBasisPackage
  -- GBINTERN ==> GroebnerInternalPackage
  I ==> Integer
  L ==> List
M ==> Matrix
NNI ==> NonNegativeInteger
OUT ==> OutputForm
P ==> Polynomial
PI ==> PositiveInteger
SEG ==> Segment
SM ==> SquareMatrix
S ==> String
V ==> Vector
mf ==> MultivariateFactorize(Var,Expon,R,GR)
rp ==> GB(R,Expon,Var,GR)
gb ==> GB(R,Expon,Var,GR)
PR ==> P R
GF ==> Fraction PR
plift ==> PolynomialCategoryLifting(Expon,Var,R,GR,GF)
Inputmode ==> Integer
groebner ==> euclideanGroebner
redPol ==> euclideanNormalForm

-- MB: The following macros are data structures to store mostly
-- intermediate results
-- Rec stores a subdeterminant with corresponding row and column indices
-- Fgb is a Groebner basis for the ideal generated by the subdeterminants
-- of a given rank.
-- Linsys specifies a linearly independent system of a given system
-- assuming a given rank, using given row and column indices
-- Linsoln stores the solution to the parametric linear system as a basis
-- and a particular solution (for a given regime)
-- Rec2 stores the rank, and a list of subdeterminants of that rank,
-- and a Groebner basis for the ideal they generate.
-- Rec3 stores a regime and the corresponding solution; the regime is
-- given by a list of equations (eqzro) and one inequation (neqzro)
-- describing the quasi-algebraic set which is the regime; the
-- additional consistency conditions due to the rhs is given by wcond.
-- Ranksolns stores a list of regimes and their solutions, and the number
-- of regimes all together.
-- Rec8 (temporary) stores a quasi-algebraic set with an indication
-- whether it is empty (sysok = false) or not (sysok = true).

-- I think psolve should be renamed parametricSolve, or even
-- parametricLinearSolve. On the other hand, may be just solve will do.
-- Please feel free to change it to conform with system conventions.
-- Most psolve routines return a list of regimes and solutions,
-- except those that output to file when the number of regimes is
-- returned instead.
-- This version has been tested on the pc version 1.608 March 13, 1992

Rec   ==> Record(det:GR,rows:L I,cols:L I)
Eqns ==> L Rec
Fgb   ==> L GR -- groebner basis
Linsoln ==> Record(partsol:V GF, basis:L V GF)
Linsys ==> Record(mat:M GF, vec:L GF, rank:NNI, rows:L I, cols:L I)
Rec2 ==> Record(rank:NNI, eqns:Eqns, fgb:Fgb)
RankConds ==> L Rec2
Rec3 ==> Record(eqzro:L GR, neqzro:L GR, wcond:L PR, bsoln:Linsoln)
Ranksolns ==> Record(rgl:L Rec3, rgsz:I)
Rec8 ==> Record(sysok:Boolean, z0:L GR, n0:L GR)

Declaration == with
  psolve: (M GR, L GR) -> L Rec3
  ++ psolve(c, w) solves c z = w for all possible ranks
  ++ of the matrix c and given right hand side vector w
  -- this is mode 1
  psolve: (M GR, L Symbol) -> L Rec3
  ++ psolve(c, w) solves c z = w for all possible ranks
  ++ of the matrix c and indeterminate right hand side w
  -- this is mode 2
  psolve: M GR -> L Rec3
  ++ psolve(c) solves the homogeneous linear system
  ++ c z = 0 for all possible ranks of the matrix c
  -- this is mode 3
  psolve: (M GR, L GR, PI) -> L Rec3
  ++ psolve(c, w, k) solves c z = w for all possible ranks >= k
  ++ of the matrix c and given right hand side vector w
  -- this is mode 4
  psolve: (M GR, L Symbol, PI) -> L Rec3
  ++ psolve(c, w, k) solves c z = w for all possible ranks >= k
  ++ of the matrix c and indeterminate right hand side w
  -- this is mode 5
  psolve: (M GR, L GR, S) -> I
  ++ psolve(c, w, s) solves c z = w for all possible ranks
  ++ of the matrix c and given right hand side vector w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 7
  psolve: (M GR, L Symbol, S) -> I
  ++ psolve(c, w, s) solves c z = w for all possible ranks
  ++ of the matrix c and indeterminate right hand side w,
  ++ writes the results to a file named s, and returns the
  ++ number of regimes
  -- this is mode 8
  psolve: (M GR, S) -> I
  ++ psolve(c, s) solves c z = 0 for all possible ranks
  ++ of the matrix c and given right hand side vector w,
  ++ writes the results to a file named s, and returns the
++ number of regimes
-- this is mode 9
psolve: (M GR, L GR, PI, S) -> I
++ psolve(c,w,k,s) solves c z = w for all possible ranks >= k
++ of the matrix c and given right hand side w,
++ writes the results to a file named s, and returns the
++ number of regimes
-- this is mode 10
psolve: (M GR, L Symbol, PI, S) -> I
++ psolve(c,w,k,s) solves c z = w for all possible ranks >= k
++ of the matrix c and indeterminate right hand side w,
++ writes the results to a file named s, and returns the
++ number of regimes
-- this is mode 11
psolve: (M GR, PI, S) -> I
++ psolve(c,k,s) solves c z = 0 for all possible ranks >= k
++ of the matrix c,
++ writes the results to a file named s, and returns the
++ number of regimes
-- this is mode 12
wrregime : (L Rec3, S) -> I
++ wrregime(l,s) writes a list of regimes to a file named s
++ and returns the number of regimes written
rdregime : S -> L Rec3
++ rdregime(s) reads in a list from a file with name s
++ for internal use only --
++ these are exported so my other packages can use them
bsolve: (M GR, L GF, NNI, S, Inputmode) -> Ranksolns
++ bsolve(c, w, r, s, m) returns a list of regimes and
++ solutions of the system c z = w for ranks at least r;
++ depending on the mode m chosen, it writes the output to
++ a file given by the string s.
dmp2rfi: GR -> GF
++ dmp2rfi(p) converts p to target domain
dmp2rfi: M GR -> M GF
++ dmp2rfi(m) converts m to target domain
dmp2rfi: L GR -> L GF
++ dmp2rfi(l) converts l to target domain
se2rfi: L Symbol -> L GF
++ se2rfi(l) converts l to target domain
pr2dmp: PR -> GR
++ pr2dmp(p) converts p to target domain
hasoln: (Fgb, L GR) -> Rec8
++ hasoln(g, l) tests whether the quasi-algebraic set
++ defined by p = 0 for p in g and q ^= 0 for q in l
++ is empty or not and returns a simplified definition
++ of the quasi-algebraic set
-- this is now done in QALGSET package
ParCondList: (M GR,NNI) -> RankConds
  ++ ParCondList(c,r) computes a list of subdeterminants of each
  ++ rank >= r of the matrix c and returns
  ++ a groebner basis for the
  ++ ideal they generate
redpps: (Linsoln, Fgb) -> Linsoln
  ++ redpps(s,g) returns the simplified form of s after reducing
  ++ modulo a groebner basis g

-- LOCAL FUNCTIONS

B1solve: Linsys -> Linsoln
  ++ B1solve(s) solves the system (s.mat) z = s.vec
  ++ for the variables given by the column indices of s.cols
  ++ in terms of the other variables and the right hand side s.vec
  ++ by assuming that the rank is s.rank,
  ++ that the system is consistent, with the linearly
  ++ independent equations indexed by the given row indices s.rows;
  ++ the coefficients in s.mat involving parameters are treated as
  ++ polynomials. B1solve(s) returns a particular solution to the
  ++ system and a basis of the homogeneous system (s.mat) z = 0.
factorset: GR -> L GR
  ++ factorset(p) returns the set of irreducible factors of p.
maxrank: RankConds -> NNI
  ++ maxrank(r) returns the maximum rank in the list r of regimes
minrank: RankConds -> NNI
  ++ minrank(r) returns the minimum rank in the list r of regimes
minset: L L GR -> L L GR
  ++ minset(sl) returns the sublist of sl consisting of the minimal
  ++ lists (with respect to inclusion) in the list sl of lists
nextSublist: (I, I) -> L L I
  ++ nextSublist(n,k) returns a list of k-subsets of {1, ..., n}.
overset?: (L GR, L L GR) -> Boolean
  ++ overset?(s,sl) returns true if s properly a sublist of a member
  ++ of sl; otherwise it returns false
ParCond : (M GR,NNI) -> Eqns
  ++ ParCond(m,k) returns the list of all k by k subdeterminants in
  ++ the matrix m
redmat: (M GR, Fgb) -> M GR
  ++ redmat(m,g) returns a matrix whose entries are those of m
  ++ modulo the ideal generated by the groebner basis g
regime: (Rec,M GR,L GF,L L GR,NNI,NNI,Inputmode) -> Rec3
  ++ regime(y,c, w, p, r, rm, m) returns a regime,
  ++ a list of polynomials specifying the consistency conditions,
  ++ a particular solution and basis representing the general
  ++ solution of the parametric linear system c z = w
  ++ on that regime. The regime returned depends on
++ the subdeterminant y.det and the row and column indices.
++ The solutions are simplified using the assumption that
++ the system has rank r and maximum rank rm. The list p
++ represents a list of list of factors of polynomials in
++ a groebner basis of the ideal generated by higher order
++ subdeterminants, and ius used for the simplification.
++ The mode m
++ distinguishes the cases when the system is homogeneous,
++ or the right hand side is arbitrary, or when there is no
++ new right hand side variables.

sqfree: GR -> GR
++ sqfree(p) returns the product of square free factors of p

inconsistent?: L GR -> Boolean
++ inconsistent?(pl) returns true if the system of equations
++ p = 0 for p in pl is inconsistent. It is assumed
++ that pl is a groebner basis.
-- this is needed because of change to
-- EuclideanGroebnerBasisPackage

inconsistent?: L PR -> Boolean
++ inconsistent?(pl) returns true if the system of equations
++ p = 0 for p in pl is inconsistent. It is assumed
++ that pl is a groebner basis.
-- this is needed because of change to
-- EuclideanGroebnerBasisPackage

Definition == add

inconsistent?(pl:L GR):Boolean ==
  for p in pl repeat
    ground? p => return true
  false

inconsistent?(pl:L PR):Boolean ==
  for p in pl repeat
    ground? p => return true
  false

Bisolve (sys:Linsys):Linsoln ==
i,j,i1,j1:1
rss:L I:=sys.rows
nss:L I:=sys.cols
k:=sys.rank
cmat:M GF:=sys.mat
n:=ncols cmat
frcols:L I:=setDifference$(L I) (expand$(SEG I) (1..n), nss)
w:L GF:=sys.vec
p:V GF:=new(n,0)
pbas:L V GF:=[]
if k ^= 0 then
  augmat:M GF:=zero(k,n+1)
  for i in rss for i1 in 1.. repeat
for j in nss for j1 in 1.. repeat
    augmat(i1,j1):=cmat(i,j)
for j in frcols for j1 in k+1.. repeat
    augmat(i1,j1):=-cmat(i,j)
augmat(i1,n+1):=w.i
augmat:=rowEchelon$(M GF) augmat
for i in nss for i1 in 1.. repeat p.i:=augmat(i1,n+1)
for j in frcols for j1 in k+1.. repeat
    pb:V GF:=new(n,0)
    pb.j:=1
for i in nss for i1 in 1.. repeat
    pb.i:=augmat(i1,j1)
pbas:=cons(pb,pbas)
else
for j in frcols for j1 in k+1.. repeat
    pb:V GF:=new(n,0)
    pb.j:=1
    pbas:=cons(pb,pbas)
[p,pbas]

regime (y, coef, w, psbf, rk, rkmax, mode) ==
i,j:I
-- use the y.det nonzero to simplify the groebner basis
-- of ideal generated by higher order subdeterminants
ydetf:L GR:=factorset y.det
yzero:L GR:=
rk = rkmax => nil$(L GR)
psbf:=[setDifference(x, ydetf) for x in psbf]
groebner$gb [*x for x in psbf]
-- simplify coefficients by modulo ideal
nc:M GF:=dmp2rfi redmat(coef,yzero)
-- solve the system
rss:L I:=y.rows; nss:L I :=y.cols
sys:Linsys:=[nc,w,rk,rss,nss]$Linsys
pps:= B1solve(sys)
pp:=pps.partsol
frows:L I:=setDifference$(L I) (expand$(SEG I) (1..nrows coef),rss)
wdc:L PR:= []
-- case homogeneous rhs
entry? (mode, [3,6,9,12]$(L I)) =>
[yzero, ydetf, wcd, redpps(pps, yzero)]$Rec3
-- case arbitrary rhs, pps not reduced
for i in frows repeat
    weqn:GF:=[nc(i,j)* (pp.j) for j in nss]
    wnum:PR:=num$GF (w.i - weqn)
    wnum = 0 => "trivially satisfied"
ground? wnum => return [yzero, ydetf, [i$PR]$(L PR),pps]$Rec3
    wdc:=cons(wnum,wdc)
entry? (mode, [2,5,8,11]$(L I)) => [yzero, ydetf, wcd, pps]$Rec3
-- case no new rhs variable
if not empty? wcd then
    yzero:=removeDuplicates append(yzero,[pr2dmp pw for pw in wcd])
  test:Rec8:=hasoln (yzero, ydetf)
not test.sysok => [test.z0, test.n0, [1$PR]$$(L PR), pps]$Rec3
  [test.z0, test.n0, [], redpps(pps, test.z0)]$Rec3

bsolve (coeff, w, h, outname, mode) ==
  r:=nrows coeff
  n:=ncols coeff
  r ^= #w => error "number of rows unequal on lhs and rhs"
newfile:FNAME
rksoln:File Rec3
  count:=0
  lrec3:L Rec3:=[]
filemode:Boolean:= entry? (mode, [7,8,9,10,11,12]$$(L I))
if filemode then
  newfile:=new$FNAME ("",outname,"regime")
  rksoln:=open$(File Rec3) newfile
y:Rec
k:NNI
rrcl:RankConds:=
  entry? (mode,[1,2,3,7,8,9]$$(L I)) => ParCondList (coeff,0)
  entry? (mode,[4,5,6,10,11,12]$$(L I)) => ParCondList (coeff,h)
rkmax:=maxrank rrcl
rkmin:=minrank rrcl
  for k in rkmax-rkmin+1..1 by -1 repeat
    rk:=rrcl.k.rank
    pc:Eqns:=rrcl.k.eqns
    psb:Fgb:= (if rk=rkmax then [] else rrcl.(k+1).fgb)
    psbf:L L GR:= [factorset x for x in psb]
    psbf:= minset(psbf)
    for y in pc repeat
      rec3:Rec3:= regime (y, coeff, w, psbf, rk, rkmax, mode)
      inconsistent? rec3.wcond => "incompatible system"
      if filemode then write_!(rksoln, rec3)
      else lrec3:= cons(rec3, lrec3)
      count:=count+1
    if filemode then close_! rksoln
  [lrec3, count]$Ranksolns

factorset y ==
ground? y => []
  [j.factor for j in factors(factor$mf y)]

ParCondList (mat, h) ==
rcl: RankConds:= []
  ps: L GR:=[]
  pc:Eqns:=[]
  npc: Eqns:=[]
  psbf: Fgb:=[]
rc: Rec
done: Boolean := false
r:= nrows mat
n:= ncols mat
maxrk:I:= min(r, n)
k: NNI
for k in min(r, n) .. h by -1 until done repeat
    pc:= ParCond(mat, k)
    npc:= [
    (empty? pc) and (k >= 1) => maxrk:= k - 1
    if ground? pc.1.det -- only one is sufficient (neqzro = {})
    then (npc:= pc; done:= true; ps := [1$GR])
    else
        zro:L GR:= (if k = maxrk then [] else rcl.1.fgb)
        covered: Boolean:= false
        for rc in pc until covered repeat
            p: GR:= redPol$rp (rc.det, zro)
            p = 0 => "incompatible or covered subdeterminant"
            test:= hasoln(zro, [rc.det])
            zroideal:= ideal(zro)
            inRadical? (p, zroideal) => "incompatible or covered"
            test.sysok => "incompatible or covered"
        ^test.sysok => "incompatible or covered"
        -- The next line is WRONG! cannot replace zro by test.z0
        zro:= groebner$gb (cons(*/test.n0, test.z0))
        zro:= groebner$gb (cons(p, zro))
        npc:= cons(rc, npc)
        done:= covered:= inconsistent? zro
        ps:= zro
        pcl: Rec2:= construct(k, npc, ps)
        rcl:= cons(pcl, rcl)
    rcl
redpps(pps, zz) ==
    pv:= pps.partsol
    r:= #pv
    pb:= pps.basis
    n:= #pb + 1
    nummat: M GR:= zero(r, n)
    denmat: M GR:= zero(r, n)
    for i in 1..r repeat
        nummat(i, 1):= pr2dmp numer$GF pv.i
        denmat(i, 1):= pr2dmp denom$GF pv.i
    for j in 2..n repeat
        for i in 1..r repeat
            nummat(i, j):= pr2dmp numer$GF (pb.(j-1)).i
            denmat(i, j):= pr2dmp denom$GF (pb.(j-1)).i
        nummat:= redmat(nummat, zz)
        denmat:= redmat(denmat, zz)
    for i in 1..r repeat
        pv.i:= (dmp2rfi nummat(i, 1))/(dmp2rfi denmat(i, 1))
for \( j \) in 2..\( n \) repeat
\[
pbj: V \text{GF} := \text{new}(r,0)
\]
for \( i \) in 1..\( r \) repeat
\[
pbj.i := \frac{\text{dmp2rfi nummat}(i,j)}{\text{dmp2rfi denmat}(i,j)}
\]
\[\text{pb}.(j-1) := \text{pbj}\]
\[\text{[pv, pb]}\]

\[
\text{dmp2rfi} \text{ (mat:M GR): M GF ==}
\]
\[
r := \text{nrows mat}
\]
\[
n := \text{ncols mat}
\]
\[
nmat := \text{zero}(r,n)
\]
for \( i \) in 1..\( r \) repeat
\[
\text{for } j \text{ in 1..n repeat}
\]
\[
nmat(i,j) := \text{dmp2rfi mat}(i,j)
\]
nmat

\[
\text{dmp2rfi} \text{ (vl: L GR): L GF ==}
\]
\[
[\text{dmp2rfi } v \text{ for } v \text{ in } vl]
\]

\[
\text{psolve} \text{ (mat:M GR, w:L GR): L Rec3 ==}
\]
\[
\text{bsolve(mat, dmp2rfi w, 1, "nofile", 1).rgl}
\]

\[
\text{psolve} \text{ (mat:M GR, w:L Symbol): L Rec3 ==}
\]
\[
\text{bsolve(mat, se2rfi w, 1, "nofile", 2).rgl}
\]

\[
\text{psolve} \text{ (mat:M GR): L Rec3 ==}
\]
\[
\text{bsolve(mat, [0$GF for } i \text{ in } 1..\text{nrows mat}, 1, "nofile", 3).rgl}
\]

\[
\text{psolve} \text{ (mat:M GR, w:L GR, h:PI): L Rec3 ==}
\]
\[
\text{bsolve(mat, dmp2rfi w, h::NNI, "nofile", 4).rgl}
\]

\[
\text{psolve} \text{ (mat:M GR, w:L Symbol, h:PI): L Rec3 ==}
\]
\[
\text{bsolve(mat, se2rfi w, h::NNI, "nofile", 5).rgl}
\]

\[
\text{psolve} \text{ (mat:M GR, h:PI): L Rec3 ==}
\]
\[
\text{bsolve(mat, [0$GF for } i \text{ in } 1..\text{nrows mat}, h::NNI, "nofile", 6).rgl}
\]

\[
\text{psolve} \text{ (mat:M GR, w:L GR, outname:S): I ==}
\]
\[
\text{bsolve(mat, dmp2rfi w, 1, outname, 7).rgsz}
\]

\[
\text{psolve} \text{ (mat:M GR, w:L Symbol, outname:S): I ==}
\]
\[
\text{bsolve(mat, se2rfi w, 1, outname, 8).rgsz}
\]

\[
\text{psolve} \text{ (mat:M GR, outname:S): I ==}
\]
\[
\text{bsolve(mat, [0$GF for } i \text{ in } 1..\text{nrows mat}, 1, outname, 9).rgsz}
\]

\[
\text{nextSublist} \text{(n,k) ==}
\]
\[
n <= 0 \Rightarrow []
\]
\[
k <= 0 \Rightarrow [\text{nil$\{$List Integer\} ]
\]
\[
k > n \Rightarrow []
\]
\[
n = 1 \text{ and } k = 1 \Rightarrow [[1]]
\]
\[
\text{mslist: L L I:[]}]
\]
for ms in \text{nextSublist}(n-1,k-1) repeat
\[
\text{mslist:=cons(append(ms,[n]),mslist)}
\]
append(\text{nextSublist}(n-1,k), mslist)
psolve (mat:M GR, w:L GR, h:PI, outname:S): I ==
   bsolve(mat, dmp2rfi w, h::NNI, outname, 10).rgsz
psolve (mat:M GR, w:L Symbol, h:PI, outname:S): I ==
   bsolve(mat, se2rfi w, h::NNI, outname, 11).rgsz
psolve (mat:M GR, h:PI, outname:S): I ==
   bsolve(mat,[0$GF for i in 1..nrows mat],h::NNI,outname, 12).rgsz

hasoln (zro,nzro) ==
   empty? zro => [true, zro, nzro]
   zro:=groebner$gb zro
   inconsistent? zro => [false, zro, nzro]
   empty? nzro =>$ [true, zro, nzro]
   pnzro:=GR:=redPol$rp (*/nzro, zro)
   nzro = 0 =>$ [false, zro, nzro]
   nzro:=factorset pnzro
   psbf:L L GR:= minset [factorset p for p in zro]
   psbf:= [setDifference(x, nzro) for x in psbf]
   entry? ([], psbf) =>$ [false, zro, nzro]
   zro:=groebner$gb [/*x for x in psbf]
   inconsistent? zro =>$ [false, zro, nzro]
   nzro:=[redPol$rp (p,zro) for p in nzro]
   nzro:=[p for p in nzro | ^(ground? p)]
   [true, zro, nzro]

se2rfi w == [coerce$GF monomial$PR (1$PR, wi, 1) for wi in w]

pr2dmp p ==
   ground? p =>$ (ground p)::GR
   algCoerceInteractive(p,PR,GR)$(Lisp) pretend GR

wrregime (lrec3, outname) ==
   newfile:FNAME:=new$FNAME ("",outname,"regime")
   rksoln: File Rec3:=open$(File Rec3) newfile
   count:I:=0 -- number of distinct regimes
   for rec3 in lrec3 repeat
      write_!(rksoln, rec3)
      count:=count+1
   close_!(rksoln)
   count

dmp2rfi (p:GR):GF ==
   map$plift ((v1:Var):GF +-> (convert v1)@Symbol::GF,
             (r1:R):GF +-> r1::PR::GF, p)

rdregime inname ==
   infilename:=filename$FNAME ("",inname, "regime")
infile: File Rec3:=open$(File Rec3) (infilename, "input")
rksoln:L Rec3:=[]
rec3:Union(Rec3, "failed"):=readIfCan_!$(File Rec3) (infile) while rec3 case Rec3 repeat
    rksoln:=cons(rec3::Rec3,rksoln) -- replace : to :: for AIX
    rec3:=readIfCan_!$(File Rec3) (infile)
close_!(infile)
rksoln

maxrank rcl ==
    empty? rcl => 0
    "max"/[j.rank for j in rcl]

minrank rcl ==
    empty? rcl => 0
    "min"/[j.rank for j in rcl]

minset lset ==
    empty? lset => lset
    [x for x in lset | ^(overset?(x,lset))]

sqfree p == */[j.factor for j in factors(squareFree p)]

ParCond (mat, k) ==
    k = 0 => [[[1, []], []]$Rec]
    j:NNI:=k::NNI
    DetEqn :Eqns := []
    r:I:= nrows(mat)
    n:I:= ncols(mat)
    k > min(r,n) => error "k exceeds maximum possible rank ",
    found:Boolean:=false
    for rss in nextSublist(r, k) until found repeat
        for nss in nextSublist(n, k) until found repeat
            matsub := mat(rss, nss) pretend SM(j, GR)
            detmat := determinant(matsub)
            if detmat ^= 0 then
                found:= (ground? detmat)
                detmat:=sqfree detmat
                neweqn:Rec:=construct(detmat, rss, nss)
                DetEqn:=cons(neweqn, DetEqn)
            found => [first DetEqn]$Eqns
    sort((z1:Rec,z2:Rec):Boolean +-> degree z1.det < degree z2.det, DetEqn)

overset?(p,qlist) ==
    empty? qlist => false
    or/[(brace$(Set GR) q) <$$(Set GR) (brace$(Set GR) p) _
        for q in qlist]
redmat (mat,psb) ==
  i,j:I
  r:=nrows(mat)
  n:=ncols(mat)
  newmat: M GR:=zero(r,n)
  for i in 1..r repeat
    for j in 1..n repeat
      p:GR:=mat(i,j)
      ground? p => newmat(i,j):=p
      newmat(i,j):=redPol$rp (p,psb)
    newmat

---

— PLEQN.dotabb —

"PLEQN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLEQN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PLEQN" -> "PFECAT"

---

package PARPC2 ParametricPlaneCurveFunctions2

— ParametricPlaneCurveFunctions2.input —

)set break resume
)sys rm -f ParametricPlaneCurveFunctions2.output
)spool ParametricPlaneCurveFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricPlaneCurveFunctions2
--R
--R ParametricPlaneCurveFunctions2(CF1: Type,CF2: Type) is a package constructor
--R Abbreviation for ParametricPlaneCurveFunctions2 is PARPC2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARPC2
--R
--R----------------------------------- Operations --------------------------------
This package has no description

See Also:
  o )show ParametricPlaneCurveFunctions2
ParametricPlaneCurveFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricPlaneCurve(CF1)) -> ParametricPlaneCurve(CF2)
++ map(f,x) \ undocumented
== add
  map(f, c) == curve(f coordinate(c,1), f coordinate(c, 2))

---

PARSC2.dotabb

"PARPC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARPC2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"PARPC2" -> "TYPE"

---

package PARSC2 ParametricSpaceCurveFunctions2

--- ParametricSpaceCurveFunctions2.input ---

)set break resume
)sys rm -f ParametricSpaceCurveFunctions2.output
)spool ParametricSpaceCurveFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametricSpaceCurveFunctions2
--R
--R ParametricSpaceCurveFunctions2(CF1: Type,CF2: Type) is a package constructor
--R Abbreviation for ParametricSpaceCurveFunctions2 is PARSC2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARSC2
--R
--R-------------------------------- Operations --------------------------------
--R map : ((CF1 -> CF2),ParametricSpaceCurve(CF1)) -> ParametricSpaceCurve(CF2)
--R
--E 1

)spool
)lisp (bye)
This package has no description

See Also:
  o )show ParametricSpaceCurveFunctions2

ParametricSpaceCurveFunctions2 (PARSC2)

Exports:
  map

)abbrev package PARSC2 ParametricSpaceCurveFunctions2
++ Description:
++ This package has no description

ParametricSpaceCurveFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricSpaceCurve(CF1)) -> ParametricSpaceCurve(CF2)
    ++ map(f,x) undocumented
    == add
    map(f, c) == curve(f coordinate(c,1), f coordinate(c,2), f coordinate(c,3))
package PARSU2 ParametricSurfaceFunctions2

-- ParametricSurfaceFunctions2.input --

)set break resume
)sys rm -f ParametricSurfaceFunctions2.output
)spool ParametricSurfaceFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)shov ParametricSurfaceFunctions2
--R
--R ParametricSurfaceFunctions2(CF1: Type,CF2: Type) is a package constructor
--R Abbreviation for ParametricSurfaceFunctions2 is PARSU2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARSU2
--R
--R-------------------------------------- Operations --------------------------------------
--R map : ((CF1 -> CF2),ParametricSurface(CF1)) -> ParametricSurface(CF2)
--R
--E 1

)spool
)lisp (bye)

-- ParametricSurfaceFunctions2.help --

====================================================================
ParametricSurfaceFunctions2 examples
====================================================================
This package has no description

See Also:
o )show ParametricSurfaceFunctions2

---

**ParametricSurfaceFunctions2 (PARSU2)**

![Diagram of ParametricSurfaceFunctions2 (PARSU2)]

Exports:
map

---

)abbrev package PARSU2 ParametricSurfaceFunctions2
++ Description:
++ This package has no description

ParametricSurfaceFunctions2(CF1: Type, CF2:Type): with
  map: (CF1 -> CF2, ParametricSurface(CF1)) -> ParametricSurface(CF2)
  ++ map(f,x) undocumented
  == add
  map(f, c) == surface(f coordinate(c,1), f coordinate(c,2), f coordinate(c,3))

---

"PARSU2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARSU2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
package PARAMP ParametrizationPackage

--- ParametrizationPackage.input ---

)set break resume
)sys rm -f ParametrizationPackage.output
)spool ParametrizationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ParametrizationPackage

--R ParametrizationPackage(\text{\text{K: Field}}, \text{symb: List(Symbol)}, \text{PolyRing: PolynomialCategory\text{(\text{K,E,OrderedVariableList(symb)}),}} \text{E: \text{... ProjectiveSpaceCategory\text{(\text{K})},PCS: LocalPowerSeriesCategory\text{(\text{K})},Plc: PlacesCategory\text{(\text{K,PCS})})}}\text{is a package constructor}

--R Abbreviation for ParametrizationPackage is PARAMP

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARAMP

--R

--R------------------------------- Operations --------------------------------

--R parametrize : (PolyRing,Plc) -> PCS

--R parametrize : (PolyRing,List(PCS)) -> PCS

--R parametrize : (PolyRing,PolyRing,Plc) -> PCS

--R parametrize : (PolyRing,Plc,Integer) -> PCS

--R

--E 1

)spool
)lisp (bye)

--- ParametrizationPackage.help ---

====================================================================
ParametrizationPackage examples
====================================================================

The following is part of the PAFF package

See Also:
**ParametrizationPackage (PARAMP)**

Exports:
parametrize

```plaintext
package PARAMP ParametrizationPackage

parametrize: (PolyRing, List(PCS)) -> PCS
```

---

**DESCRIPTION:**

The following is part of the PAFF package

**PARAMP**

**DIRPCAT**

**PFECAT**
parametrize: (PolyRing, Plc) -> PCS
++ parametrize(f, pl) returns the local parametrization of the
++ polynomial function f at the place pl. Note that local
++ parametrization of the place must have first been computed and set.
++ For simple point on a curve, this done with \spad{pointToPlace}.
++ The local parametrization places corresponding to a leaf in a
++ desingularization tree are computed at the moment of
++ their "creation". (See package \spad{DesingTreePackage}).

parametrize: (PolyRing, PolyRing, Plc) -> PCS
++ parametrize(f, g, pl) returns the local parametrization of the
++ rational function f/g at the place pl. Note that local
++ parametrization of the place must have first been computed and set.
++ For simple point on a curve, this done with \spad{pointToPlace}.
++ The local parametrization places corresponding to a leaf in a
++ desingularization tree are computed at the moment of
++ their "creation". (See package \spad{DesingTreePackage}).

parametrize: (PolyRing, Plc, Integer) -> PCS
++ parametrize(f, pl, n) returns t^n * parametrize(f, p).

Impl ==> add

import PCS
import PolyRing

-- the following returns the parametrization in term of
-- the precomputed local parametrization
-- of the point pt. Note if pl is a place and pl = pt::Plc then
-- parametrize(f, pt) <> parametrize(pl) unless pt is a simple point
parametrize(f: PolyRing, localPar: List(PCS)) ==
  zero?(f) => 0
  lc:K := leadingCoefficient(f)
  ld:E := degree f
  ldp:List NonNegativeInteger := parts(ld)
  if empty?(localPar) then error _
    "the parametrization of the place or leaf has not been done yet!"
  monoPar:PCS := reduce("*", [ s**e for s in localPar for e in ldp])
  lc* monoPar + parametrize(reductum(f), localPar)

parametrize(f: PolyRing, pt: Plc) ==
  zero?(f) => 0
  localPar:List PCS := localParam pt
  parametrize(f, localPar)

parametrize(f: PolyRing, g: PolyRing, pt: Plc) ==
  sf := parametrize(f, pt)
  sg := parametrize(g, pt)
  sf * inv sg
parametrize(f:PolyRing, pt: Plc, n: Integer) ==
s:=parametrize(f, pt)
shift(s, n)

-- PARAMP.dotabb --
"PARAMP" [color="#FF4488", href="bookvol10.4.pdf#nameddest=PARAMP"]
"DIRPCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=DIRPCAT"]
"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]
"PARAMP" -> "DIRPCAT"
"PARAMP" -> "PFECAT"

package PFRPAC PartialFractionPackage

-- PartialFractionPackage.input --

)set break resume
)spool PartialFractionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 5
a:=x+1/(y+1)
--R
--R x y + x + 1
--R (1) -----------
--R y + 1
--R

Type: Fraction(Polynomial(Integer))
--E 1

--S 2 of 5
partialFraction(a,y)$PFRPAC(INT)
--R
--R 1
--R (2) x + ------
--R y + 1

--R Type: PartialFraction(UnivariatePolynomial(y,Fraction(Polynomial(Integer))))
--E 2

--S 3 of 5
b:=y+1/(x+1)
--R
--R (x + 1)y + 1
--R (3) ---------
--R x + 1
--R Type: Fraction(Polynomial(Integer))
--E 3

--S 4 of 5
partialFraction(b,x)$PFRPAC(INT)
--R
--R 1
--R (4) y + ----- 
--R x + 1
--R Type: PartialFraction(UnivariatePolynomial(x,Fraction(Polynomial(Integer))))
--E 4

--S 5 of 5
)show PartialFractionPackage
--R
--R PartialFractionPackage(R: Join(EuclideanDomain,CharacteristicZero)) is a package constructor
--R Abbreviation for PartialFractionPackage is PFRPAC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFRPAC
--R
--R----------------------------------- Operations -----------------------------------
--R partialFraction : (Fraction(Polynomial(R)),Symbol) -> Any
--R partialFraction : (Polynomial(R),Factored(Polynomial(R)),Symbol) -> Any
--R
--E 5

)spool
)lisp (bye)
a := x + 1/(y+1)

it will end up with the result in Fraction Polynomial Integer.

\[
\frac{x y + x + 1}{y + 1}
\]

We might want it in a partial fraction form. This can be done with:

```
partialFraction(a, y)$PFRPAC(INT)
```

```
1
x + ------
y + 1
```

We can do the same thing with a different variable:

b := y + 1/(x+1)

\[
\frac{(x + 1)y + 1}{x + 1}
\]

```
partialFraction(b, x)$PFRPAC(INT)
```

```
1
y + ------
x + 1
```

See Also:
- \)show PartialFractionPackage
PartialFractionPackage (PFRPAC)

Exports:
partialFraction

--- package PFRPAC PartialFractionPackage ---

)abbrev package PFRPAC PartialFractionPackage
++ Author: Barry M. Trager
++ Date Created: 1992
++ Description:
++ The package \spadtype{PartialFractionPackage} gives an easier
++ to use interface the domain \spadtype{PartialFraction}.
++ The user gives a fraction of polynomials, and a variable and
++ the package converts it to the proper datatype for the
++ \spadtype{PartialFraction} domain.

PartialFractionPackage(R): Cat == Capsule where
-- R : UniqueFactorizationDomain -- not yet supported
R : Join(EuclideanDomain, CharacteristicZero)
FPR ==> Fraction Polynomial R
INDE ==> IndexedExponents Symbol
PR ==> Polynomial R
SUP ==> SparseUnivariatePolynomial
Cat == with
  partialFraction: (FPR, Symbol) -> Any
  ++ partialFraction(rf, var) returns the partial fraction decomposition
  ++ of the rational function rf with respect to the variable var.
  ++
  ++X a:=x+1/(y+1)
  ++X partialFraction(a,y)$PFRPAC(INT)
  partialFraction: (PR, Factored PR, Symbol) -> Any
  ++ partialFraction(num, facdenom, var) returns the partial fraction
  ++ decomposition of the rational function whose numerator is num and
  ++ whose factored denominator is facdenom with respect to the
  ++ variable var.
Capsule == add
partialFraction(rf, v) ==
  df := factor(denom rf)$MultivariateFactorize(Symbol, INDE,R,PR)
  partialFraction(numer rf, df, v)

makeSup(p:Polynomial R, v:Symbol) : SparseUnivariatePolynomial FPR ==
  up := univariate(p,v)
  map((z1:PR):FPR -> z1::FPR,up)
  $UnivariatePolynomialCategoryFunctions2(PR, SUP PR, FPR, SUP FPR)

partialFraction(p, facq, v) ==
  up := UnivariatePolynomial(v, Fraction Polynomial R)
  fup := Factored up
  ffact := [makeSup(u.factor,v) pretend up,u.exponent]
     for u in factors facq
  fcont:=makeSup(unit facq,v) pretend up
  nflist:fup := fcont*(*/[primeFactor(ff.irr,ff.pow) for ff in ffact])
  pfup:=partialFraction(makeSup(p,v) pretend up,nflist)$PartialFraction(up)
  coerce(pfup)$AnyFunctions1(PartialFraction up)

package PARTPERM PartitionsAndPermutations

package PARTPERM PartitionsAndPermutations

<table>
<thead>
<tr>
<th>PartitionsAndPermutations.input</th>
</tr>
</thead>
</table>

)set break resume
)sys rm -f PartitionsAndPermutations.output
)spool PartitionsAndPermutations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PartitionsAndPermutations
--R
--R PartitionsAndPermutations is a package constructor
--R Abbreviation for PartitionsAndPermutations is PARTPERM
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PARTPERM
--R
--R------------------------------- Operations --------------------------------
--R conjugate : List(Integer) -> List(Integer)
--R conjugates : Stream(List(Integer)) -> Stream(List(Integer))
--R partitions : (Integer,Integer,Integer) -> Stream(List(Integer))
--R partitions : Integer -> Stream(List(Integer))
--R partitions : (Integer,Integer) -> Stream(List(Integer))
--R permutations : Integer -> Stream(List(Integer))
--R sequences : (List(Integer),List(Integer)) -> Stream(List(Integer))
--R sequences : List(Integer) -> Stream(List(Integer))
--R shuffle : (List(Integer),List(Integer)) -> Stream(List(Integer))
--R shufflein : (List(Integer),Stream(List(Integer))) -> Stream(List(Integer))
--R
--E 1

)spool
)lisp (bye)

——

— PartitionsAndPermutations.help —

====================================================================
PartitionsAndPermutations examples
====================================================================

PartitionsAndPermutations contains functions for generating streams of
total integer partitions, and streams of sequences of integers composed from
a multi-set.

See Also:
o )show PartitionsAndPermutations
PartitionsAndPermutations (PARTPERM)

Exports:
   conjugate conjugates partitions permutations sequences
   shuffle shufflein

~~~ package PARTPERM PartitionsAndPermutations ~~~

)abbrev package PARTPERM PartitionsAndPermutations
++ Author: William H. Burge
++ Date Created: 29 October 1987
++ Date Last Updated: 3 April 1991
++ Description:
++ PartitionsAndPermutations contains functions for generating streams of
++ integer partitions, and streams of sequences of integers
++ composed from a multi-set.

PartitionsAndPermutations:Exports == Implementation where
   I  == Integer
   L  == List
   ST == Stream
   ST1 == StreamFunctions1
   ST2 == StreamFunctions2
   ST3 == StreamFunctions3

Exports ==> with

   partitions: (I,I,I) -> ST L I
      "\textsf{partitions}(p,l,n)" is the stream of partitions
      of \emph{n} whose number of parts is no greater than \emph{p}
      and whose largest part is no greater than \emph{l}.
   partitions: I    -> ST L I
      "\textsf{partitions}(n)" is the stream of all partitions of \emph{n}.
   partitions: (I,I) -> ST L I
      "\textsf{partitions}(p,l)" is the stream of all
      partitions whose number of
      parts and largest part are no greater than \emph{p} and \emph{l}. 
conjugate: L I -> L I
   \spad{conjugate(pt)} is the conjugate of the partition pt.
conjugates: ST L I -> ST L I
   \spad{conjugates(lp)} is the stream of conjugates of a stream
   of partitions lp.
shuffle: (L I,L I) -> ST L I
   \spad{shuffle(l1,l2)} forms the stream of all shuffles of l1
   and l2, i.e. all sequences that can be formed from
   merging l1 and l2.
shufflein: (L I,ST L I) -> ST L I
   \spad{shufflein(l,st)} maps shuffle(l,u) on to all
   members u of st, concatenating the results.
sequences: (L I,L I) -> ST L I
   \spad{sequences(l1,l2)} is the stream of all sequences that
   can be composed from the multiset defined from
   two lists of integers l1 and l2.
   For example, the pair \spad{([1,2,4],[2,3,5])} represents
   multi-set with 1 \spad{2}'s, 2 \spad{3}'s, and 4 \spad{5}'s.
sequences: L I -> ST L I
   \spad{sequences(l1)} is the set of
   all sequences formed from
   \spad{l1} 0's, \spad{1} 1's, \spad{2} 2's,..., \spad{n} n's.
permutations: I -> ST L I
   \spad{permutations(n)} is the stream of permutations
   formed from \spad{1,2,3,...,n}.

Implementation == add

partitions(M,N,n) ==
   zero? n => concat(empty()$L(I),empty()$(ST L I))
   zero? M or zero? N or n < 0 => empty()
   c := map((l1:List(I)):List(I)+->concat(N,l1),partitions(M - 1,N,n - N))
   concat(c,partitions(M,N - 1,n))
partitions n == partitions(n,n,n)

partitions(M,N)==
   aaa : L ST L I := [partitions(M,N,i) for i in 0..M*N]
   concat(aaa :: ST ST L I)$ST1(L I)

-- nogreq(n,l) is the number of elements of l that are greater or
-- equal to n
nogreq: (I,L I) -> I
nogreq(n,x) == +/[1 for i in x | i >= n]

conjugate x ==
   empty? x => empty()
   [nogreq(i,x) for i in 1..first x]

conjugates z == map(conjugate,z)
shuffle(x,y) ==
   empty? x => concat(y,empty())$ST L I
   empty? y => concat(x,empty())$ST L I
   concat(map((l1:List(I)):List(I)+->concat(first x,l1),shuffle(rest x,y)),_
   map((l2:List(I)):List(I)+->concat(first y,l2),shuffle(x,rest y)))

shufflein(x,yy) ==
   concat(map((l1:List(I)):ST(L I)+->shuffle(x,l1),yy)_
   $ST2(L I,ST L I))$ST1(L I)

-- rpt(n,m) is the list of n m's
rpt: (I,I) -> L I
rpt(n,m) == [m for i in 1..n]

-- zrpt(x,y) where x is [x0,x1,x2...] and y is [y0,y1,y2...]
-- is the stream [rpt(x0,y0),rpt(x1,y1),...]
zrpt: (L I,L I) -> ST L I
zrpt(x,y) == map(rpt,x :: ST I,y :: ST I)$ST3(I,I,L I)

sequences(x,y) ==
   reduce(concat(empty()$L(I),empty()$(ST L I)),_
   shufflein,zrpt(x,y))$ST2(L I,ST L I)

sequences x == sequences(x,[i for i in 0..#x-1])

permutations n == sequences(rpt(n,1),[i for i in 1..n])

— PARTPERM.dotabb —

"PARTPERM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PARTPERM"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PARTPERM" -> "FLAGG"

package PATTERN1 PatternFunctions1

— PatternFunctions1.input —

)set break resume
)sys rm -f PatternFunctions1.output
PatternFunctions1 help

Utilities for handling patterns

See Also:
  o )show PatternFunctions1
PatternFunctions1 (PATTERN1)

Exports:
badValues addBadValue predicate satisfy? suchThat

--- package PATTERN1 PatternFunctions1 ---

)abbrev package PATTERN1 PatternFunctions1
++ Author: Manuel Bronstein
++ Date Created: 28 Nov 1989
++ Date Last Updated: 5 Jul 1990
++ Description:
++ Utilities for handling patterns

PatternFunctions1(R:SetCategory, D:Type): with
suchThat : (Pattern R, D -> Boolean) -> Pattern R
  ++ suchThat(p, f) makes a copy of p and adds the predicate
  ++ f to the copy, which is returned.
suchThat : (Pattern R, List(D -> Boolean)) -> Pattern R
  ++ \spad{suchThat(p, [f1,...,fn])} makes a copy of p and adds the
  ++ predicate f1 and ... and fn to the copy, which is returned.
suchThat : (Pattern R, List Symbol, List D -> Boolean) -> Pattern R
  ++ \spad{suchThat(p, [a1,...,an], f)} returns a copy of p with
  ++ the top-level predicate set to \spad{f(a1,...,an)}.
predicate : Pattern R -> (D -> Boolean)
  ++ predicate(p) returns the predicate attached to p, the
  ++ constant function true if p has no predicates attached to it.
satisfy? : (D, Pattern R) -> Boolean
  ++ satisfy?(v, p) returns \spad{f(v)} where f is the predicate
  ++ attached to p.
satisfy? : (List D, Pattern R) -> Boolean
  ++ \spad{satisfy?([v1,...,vn], p)} returns \spad{f(v1,...,vn)}
  ++ where f is the
  ++ top-level predicate attached to p.
addBadValue : (Pattern R, D) -> Pattern R
  ++ addBadValue(p, v) adds v to the list of "bad values" for p;
  ++ p is not allowed to match any of its "bad values".
badValues : Pattern R -> List D  
++ badValues(p) returns the list of "bad values" for p;  
++ p is not allowed to match any of its "bad values".

== add
A1D ==> AnyFunctions1(D)
A1 ==> AnyFunctions1(D -> Boolean)
A1L ==> AnyFunctions1(List D -> Boolean)

applyAll: (List Any, D) -> Boolean
st : (Pattern R, List Any) -> Pattern R

st(p, l) == withPredicates(p, concat(predicates p, l))
predicate p == (d1:D):Boolean +-> applyAll(predicates p, d1)
addBadValue(p, v) == addBadValue(p, coerce(v)$A1D)
badValues p == [retract(v)$A1D for v in getBadValues p]
suchThat(p, l, f) == setTopPredicate(copy p, l, coerce(f)$A1L)
satisfy?(d:D, p:Pattern R) == st(p, [coerce(f)$A1])
satisfy?(l:List D, p:Pattern R) ==
  empty?((rec := topPredicate p).var) => true
  retract(rec.pred)$A1L l

applyAll(1, d) ==
  for f in l repeat
    not(retract(f)$A1 d) => return false
  true

suchThat(p:Pattern R, l:List(D -> Boolean)) ==
  st(p, [coerce(f)$A1 for f in l])
package PATTERN2 PatternFunctions2

--- PatternFunctions2.input ---

)set break resume
)sys rm -f PatternFunctions2.output
)spool PatternFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternFunctions2
--R
--R PatternFunctions2(R: SetCategory,S: SetCategory) is a package constructor
--R Abbreviation for PatternFunctions2 is PATTERN2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PATTERN2
--R
--R-------------------------------- Operations --------------------------------
--R map : ((R -> S),Pattern(R)) -> Pattern(S)
--R
--E 1

)spool
)lisp (bye)

-----

--- PatternFunctions2.help ---

====================================================================
PatternFunctions2 examples
====================================================================

Lifts maps to patterns

See Also:
o )show PatternFunctions2

-----
PatternFunctions2 (PATTERN2)

Exports:
map

--- package PATTERN2 PatternFunctions2 ---

>abbrev package PATTERN2 PatternFunctions2
++ Author: Manuel Bronstein
++ Date Created: 28 Nov 1989
++ Date Last Updated: 12 Jan 1990
++ Description:
++ Lifts maps to patterns

PatternFunctions2(R:SetCategory, S:SetCategory): with
  map: (R -> S, Pattern R) -> Pattern S
  ++ map(f, p) applies f to all the leaves of p and
  ++ returns the result as a pattern over S.
  == add
  map(f, p) ==
    (r := (retractIfCan p)@Union(R, "failed")) case R =>
      f(r::R)::Pattern(S)
    (u := isOp p) case Record(op:BasicOperator, arg:List Pattern R) =>
      ur := u::Record(op:BasicOperator, arg:List Pattern R)
      (ur.op) [map(f, x) for x in ur.arg]
    (v := isQuotient p) case Record(num:Pattern R, den:Pattern R) =>
      vr := v::Record(num:Pattern R, den:Pattern R)
      map(f, vr.num) / map(f, vr.den)
    (l := isPlus p) case List(Pattern R) =>
      reduce("+", [map(f, x) for x in l::List(Pattern R)])
    (l := isTimes p) case List(Pattern R) =>
      reduce("*", [map(f, x) for x in l::List(Pattern R)])
    (x := isPower p) case
      Record(val:Pattern R, exponent: Pattern R) =>
      xr := x::Record(val:Pattern R, exponent: Pattern R)
      map(f, xr.val) ** map(f, xr.exponent)
    (w := isExpt p) case
Record(val:Pattern R, exponent: NonNegativeInteger) =>
wr := w::Record(val:Pattern R, exponent: NonNegativeInteger)
map(f, wr.val) ** wr.exponent
sy := retract(p)@Symbol
setPredicates(sy::Pattern(S), copy predicates p)

---

— PATTERN2.dotabb —

"PATTERN2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATTERN2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PATTERN2" -> "FLAGG"

---

package PATMATCH PatternMatch

— PatternMatch.input —

)set break resume
)sys rm -f PatternMatch.output
)spool PatternMatch.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatch
--R
--R PatternMatch(Base: SetCategory,Subject: PatternMatchable(Base),Pat: ConvertibleTo(Pattern(Base))) is a
--R Abbreviation for PatternMatch is PATMATCH
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PATMATCH
--R
--R----------------------------------- Operations -----------------------------------
--R is? : (Subject,Pat) -> Boolean      is? : (List(Subject),Pat) -> Boolean
--R Is : (List(Subject),Pat) -> PatternMatchListResult(Base,Subject,List(Subject))
--R Is : (Subject,Pat) -> List(Equation(Subject)) if Subject has RETRACT(SYMBOL)
--R Is : (Subject,Pat) -> List(Equation(Polynomial(Subject))) if Subject has RING and not(has(Subject,RetractableTo(Symbol))) and not
--R
--E 1
PatternMatch (PATMATCH)

Exports:
  is?  Is

--- package PATMATCH PatternMatch ---

)abbrev package PATMATCH PatternMatch
++ Top-level pattern matching functions
++ Author: Manuel Bronstein
++ Date Created: 3 Dec 1989
++ Date Last Updated: 29 Jun 1990
++ Description:
  ++ This package provides the top-level pattern matching functions.
PatternMatch(Base, Subject, Pat): Exports == Implementation where
Base : SetCategory
Subject: PatternMatchable Base
Pat : ConvertibleTo Pattern Base

Exports ==> with
is?: (Subject, Pat) -> Boolean
++ is?(expr, pat) tests if the expression expr matches
++ the pattern pat.
is?: (List Subject, Pat) -> Boolean
++ is?([e1,...,en], pat) tests if the list of
++ expressions \(\text{spad}[[e1,...,en]]\) matches
++ the pattern pat.
Is : (List Subject, Pat) ->
++ PatternMatchListAggregate(Base, Subject, List Subject)
++ Is([e1,...,en], pat) matches the pattern pat on the list of
++ expressions \(\text{spad}[[e1,...,en]]\) and returns the result.
if Subject has RetractableTo(Symbol) then
Is: (Subject, Pat) -> List Equation Subject
++ Is(expr, pat) matches the pattern pat on the expression
++ expr and returns a list of matches \(\text{spad}[[v1 = e1,...,vn = en]]\);
++ returns an empty list if either expr is exactly equal to
++ pat or if pat does not match expr.
else
if Subject has Ring then
Is: (Subject, Pat) -> List Equation Polynomial Subject
++ Is(expr, pat) matches the pattern pat on the expression
++ expr and returns a list of matches \(\text{spad}[[v1 = e1,...,vn = en]]\);
++ returns an empty list if either expr is exactly equal to
++ pat or if pat does not match expr.
else
Is: (Subject, Pat) -> PatternMatchResult(Base, Subject)
++ Is(expr, pat) matches the pattern pat on the expression
++ expr and returns a match of the form \(\text{spad}[[v1 = e1,...,vn = en]]\);
++ returns an empty match if expr is exactly equal to pat.
++ returns a \(\text{spadfun}[/failed/]\) match if pat does not match expr.

Implementation ==> add
import PatternMatchListAggregate(Base, Subject, List Subject)

ist: (Subject, Pat) -> PatternMatchResult(Base, Subject)

ist(s, p) == patternMatch(s, convert p, new())
is?(s: Subject, p:Pat) == not failed? ist(s, p)
is?(s:List Subject, p:Pat) == not failed? Is(s, p)
Is(s:List Subject, p:Pat) == patternMatch(s, convert p, new())

if Subject has RetractableTo(Symbol) then
Is(s:Subject, p:Pat):List(Equation Subject) ==
failed?(r := ist(s, p)) => empty()
[rec.key::Subject = rec.entry for rec in destruct r]

else
  if Subject has Ring then
    Is(s:Subject, p:Pat):List(Equation Polynomial Subject) ==
    failed?(r := ist(s, p)) => empty()
    [rec.key::Polynomial(Subject) =$Equation(Polynomial Subject)
     rec.entry::Polynomial(Subject) for rec in destruct r]
  else
    Is(s:Subject,p:Pat):PatternMatchResult(Base,Subject) == ist(s,p)

— PATTERNMATCH.dotabb —

"PATTERNMATCH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATTERNMATCH"]
"PATMAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PATMAB"]
"RETRACT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RETRACT"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"PATTERNMATCH" -> "PATMAB"
"PATTERNMATCH" -> "RETRACT"
"PATTERNMATCH" -> "LMODULE"
"PATTERNMATCH" -> "SGROUP"

package PMASS PatternMatchAssertions

— PatternMatchAssertions.input —

)set break resume
)sys rm -f PatternMatchAssertions.output
)spool PatternMatchAssertions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchAssertions
--R
--R PatternMatchAssertions is a package constructor
---R Abbreviation for PatternMatchAssertions is PMASS
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMASS
---R
---R------------------------------- Operations --------------------------------
---R assert : (Symbol,String) -> Expression(Integer)
---R constant : Symbol -> Expression(Integer)
---R multiple : Symbol -> Expression(Integer)
---R optional : Symbol -> Expression(Integer)
---R
---E 1

)spool
)lisp (bye)

-----

--- PatternMatchAssertions.help ---

====================================================================
PatternMatchAssertions examples
====================================================================

Attaching assertions to symbols for pattern matching.

See Also:
o )show PatternMatchAssertions

-----

PatternMatchAssertions (PMASS)

Exports:
assert constant multiple optional
package PMASS PatternMatchAssertions

)abbrev package PMASS PatternMatchAssertions
++ Author: Manuel Bronstein
++ Date Created: 21 Mar 1989
++ Date Last Updated: 23 May 1990
++ Description:
++ Attaching assertions to symbols for pattern matching.

PatternMatchAssertions(): Exports == Implementation where
  FE == Expression Integer

Exports ==>
  assert : (Symbol, String) -> FE
    ++ assert(x, s) makes the assertion s about x.
  constant: Symbol -> FE
    ++ constant(x) tells the pattern matcher that x should
    ++ match only the symbol 'x and no other quantity.
  optional: Symbol -> FE
    ++ optional(x) tells the pattern matcher that x can match
    ++ an identity (0 in a sum, 1 in a product or exponentiation).
  multiple: Symbol -> FE
    ++ multiple(x) tells the pattern matcher that x should
    ++ preferably match a multi-term quantity in a sum or product.
    ++ For matching on lists, multiple(x) tells the pattern matcher
    ++ that x should match a list instead of an element of a list.

Implementation ==> add
  import FunctionSpaceAssertions(Integer, FE)

  constant x == constant(x::FE)
  multiple x == multiple(x::FE)
  optional x == optional(x::FE)
  assert(x, s) == assert(x::FE, s)
package PMFS PatternMatchFunctionSpace

--- PatternMatchFunctionSpace.input ---

)set break resume
)sys rm -f PatternMatchFunctionSpace.output
)spool PatternMatchFunctionSpace.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchFunctionSpace
--R
--R PatternMatchFunctionSpace(S: SetCategory,R: Join(IntegralDomain,OrderedSet,PatternMatchable(S)),F: Join(FunctionSpace(R),ConvertibleTo(Pattern(S)),PatternMatchable(S),RetractableTo(Kernel($)))) is a package constructor
--R Abbreviation for PatternMatchFunctionSpace is PMFS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMFS
--R
--R---------------------------------------- Operations ----------------------------------------
--R patternMatch : (F,Pattern(S),PatternMatchResult(S,F)) -> PatternMatchResult(S,F)
--R
--E 1

)spool
)lisp (bye)

---

--- PatternMatchFunctionSpace.help ---

====================================================================
PatternMatchFunctionSpace examples
====================================================================

This package provides pattern matching functions on function spaces.

See Also:
• )show PatternMatchFunctionSpace

---
PatternMatchFunctionSpace (PMFS)

Exports:
patternMatch

— package PMFS PatternMatchFunctionSpace —

)abbrev package PMFS PatternMatchFunctionSpace
++ Author: Manuel Bronstein
++ Date Created: 15 Mar 1990
++ Date Last Updated: 20 June 1991
++ Description:
++ This package provides pattern matching functions on function spaces.

PatternMatchFunctionSpace(S, R, F): Exports== Implementation where
S: SetCategory
R: Join(IntegralDomain, OrderedSet, PatternMatchable S)
F: Join(FunctionSpace R, ConvertibleTo Pattern S, PatternMatchable S,
  RetractableTo Kernel %) -- that one is redundant but won't
  -- compile without it

N  ==> NonNegativeInteger
K  ==> Kernel F
PAT ==> Pattern S
PRS ==> PatternMatchResult(S, F)
RCP ==> Record(val:PAT, exponent:N)
RCX ==> Record(var:K, exponent:Integer)

Exports ==> with
  patternMatch: (F, PAT, PRS) -> PRS
    ++ patternMatch(expr, pat, res) matches the pattern pat to the
    ++ expression expr; res contains the variables of pat which
    ++ are already matched and their matches.

Implementation ==> add
  import PatternMatchKernel(S, F)
  import PatternMatchTools(S, R, F)
import PatternMatchPushDown(S, R, F)

patternMatch(x, p, l) ==
generic? p => addMatch(p, x, l)
(r := retractIfCan(x)@Union(R, "failed")) case R =>
patternMatch(r::R, p, l)
(v := retractIfCan(x)@Union(K, "failed")) case K =>
patternMatch(v::K, p, l)
(q := isQuotient p) case Record(num:PAT, den:PAT) =>
uq := q::Record(num:PAT, den:PAT)
failed?(l := patternMatch(numer(x)::F, uq.num, l)) => 1
patternMatch(denom(x)::F, uq.den, l)
(u := isPlus p) case List(PAT) =>
  (lx := isPlus x) case List(F) =>
patternMatch(lx::List(F), u::List(PAT), l1 => +/l1, 1, patternMatch)
  (u := optpair(u::List(PAT))) case List(PAT) =>
failed?(l := addMatch(first(u::List(PAT)), 0, 1)) => failed()
patternMatch(x, second(u::List(PAT)), l)
failed()
(u := isTimes p) case List(PAT) =>
  (lx := isTimes x) case List(F) =>
patternMatchTimes(lx::List(F), u::List(PAT), l, patternMatch)
  (u := optpair(u::List(PAT))) case List(PAT) =>
failed?(l := addMatch(first(u::List(PAT)), 1, 1)) => failed()
patternMatch(x, second(u::List(PAT)), l)
failed()
(uu := isPower p) case Record(val:PAT, exponent:PAT) =>
uur := uu::Record(val:PAT, exponent: PAT)
(ex := isExpt x) case RCX =>
failed?(l := patternMatch((ex::RCX).exponent::Integer::F, uur.exponent, l)) => failed()
patternMatch((ex::RCX).var, uur.val, l)
optional?(uur.exponent) =>
failed?(l := addMatch(uur.exponent, 1, 1)) => failed()
patternMatch(x, uur.val, l)
failed()
((ep := isExpt p) case RCP) and ((ex := isExpt x) case RCX) and
  ((ex::RCX).exponent = ((ep::RCP).exponent)::Integer) =>
patternMatch((ex::RCX).var, (ep::RCP).val, l)
failed()
package PMINS PatternMatchIntegerNumberSystem

— PatternMatchIntegerNumberSystem.input —

)set break resume
/sys rm -f PatternMatchIntegerNumberSystem.output
/spool PatternMatchIntegerNumberSystem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchIntegerNumberSystem

--R
--R PatternMatchIntegerNumberSystem(I: IntegerNumberSystem) is a package constructor
--R Abbreviation for PatternMatchIntegerNumberSystem is PMINS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMINS
--R
--R-------------------------------- Operations --------------------------------
--R patternMatch : (I,Pattern(Integer),PatternMatchResult(Integer,I)) -> PatternMatchResult(Integer,I)
--R
--E 1

)spool
)lisp (bye)

— PatternMatchIntegerNumberSystem.help —

====================================================================
PatternMatchIntegerNumberSystem examples
====================================================================

This package provides pattern matching functions on integers.

See Also:
  o )show PatternMatchIntegerNumberSystem
PatternMatchIntegerNumberSystem (PMINS)

Exports:
patternMatch

— package PMINS PatternMatchIntegerNumberSystem —

)abbrev package PMINS PatternMatchIntegerNumberSystem
++ Author: Manuel Bronstein
++ Date Created: 29 Nov 1989
++ Date Last Updated: 22 Mar 1990
++ Description:
++ This package provides pattern matching functions on integers.

PatternMatchIntegerNumberSystem(I:IntegerNumberSystem): with
  patternMatch: (I, Pattern Integer, PatternMatchResult(Integer, I)) ->
    PatternMatchResult(Integer, I)
  ++ patternMatch(n, pat, res) matches the pattern pat to the
  ++ integer n; res contains the variables of pat which
  ++ are already matched and their matches.
  == add
  import IntegerRoots(I)

PAT ==> Pattern Integer
PMR ==> PatternMatchResult(Integer, I)

patternMatchInner : (I, PAT, PMR) -> PMR
patternMatchRestricted: (I, PAT, PMR, I) -> PMR
patternMatchSumProd :
  (I, List PAT, PMR, (I, I) -> Union(I, "failed"), I) -> PMR

patternMatch(x, p, l) ==
  generic? p => addMatch(p, x, l)
  patternMatchInner(x, p, l)

patternMatchRestricted(x, p, l, y) ==
  generic? p => addMatchRestricted(p, x, l, y)
patternMatchInner(x, p, l) ==

patternMatchSumProd(x, lp, l, invOp, ident) ==
  #lp = 2 =>
  p2 := last lp
  if ((r := retractIfCan(p1 := first lp)@Union(Integer,"failed")) case "failed") then
    (p1 := p2; p2 := first lp)
  (r := retractIfCan(p1)@Union(Integer, "failed")) case "failed" =>
    failed()

  (y := invOp(x, r::Integer::I)) case "failed" =>
    failed()
  patternMatchRestricted(y::I, p2, l, ident)
  failed()

patternMatchInner(x, p, l) ==
  constant? p =>
    (r := retractIfCan(p)@Union(Integer, "failed")) case Integer =>
      convert(x)@Integer = r::Integer => l
      failed()
    failed()
  (u := isExpt p) case Record(val:PAT,exponent:NonNegativeInteger) =>
    ur := u::Record(val:PAT, exponent:NonNegativeInteger)
    (v := perfectNthRoot(x, ur.exponent)) case "failed" =>
      patternMatchRestricted(v::I, ur.val, l, 1)
    failed()
  (uu := isPower p) case Record(val:PAT, exponent:PAT) =>
    uur := uu::Record(val:PAT, exponent: PAT)
    pr := perfectNthRoot x
    failed?(l := patternMatchRestricted(pr.exponent::Integer::I, urr.exponent, 1, 1)) => failed()
    patternMatchRestricted(pr.base, uur.val, l, 1)
  (w := isTimes p) case List(PAT) =>
    patternMatchSumProd(x, w::List(PAT), l, (i1:I,i2:I):Union(I,"failed") +-> i1 exquo i2, 1)
  (w := isPlus p) case List(PAT) =>
    patternMatchSumProd(x,w::List(PAT),l, (i1:I,i2:I):Union(I,"failed") +-> (i1-i2)::Union(I,"failed"),0)
  (uv := isQuotient p) case Record(num:PAT, den:PAT) =>
    uvr := uv::Record(num:PAT, den:PAT)
    (r := retractIfCan(uvr.num)@Union(Integer,"failed")) case Integer
      and (v := r::Integer::I exquo x) case I =>
        patternMatchRestricted(v::I, uvr.den, l, 1)
    (r := retractIfCan(uvr.den)@Union(Integer,"failed")) case Integer
      => patternMatch(r::Integer * x, uvr.num, l)
    failed()
    failed()

— PMINS.dotabb —
package INTPM PatternMatchIntegration

-- PatternMatchIntegration.input --

)set break resume
)sys rm -f PatternMatchIntegration.output
)spool PatternMatchIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchIntegration

--R
--R PatternMatchIntegration(R: Join(OrderedSet, RetractableTo(Integer), GcdDomain, LinearlyExplicitRingOver(Integer)), F: Join(AlgebraicallyClosedField, TranscendentalFunctionCategory, FunctionSpace(R))) is a package constructor

--R Abbreviation for PatternMatchIntegration is INTPM

--R This constructor is not exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTPM

)--R

)--R----------------------------- Operations ------------------------------

--R pmComplexintegrate : (F, Symbol) -> Union(Record(special: F, integrand: F), "failed") if F has LFCAT and R has KONVERT(PATTERN(INT)) and R has PATMAB(INT)

--R pmintegrate : (F, Symbol) -> Union(Record(special: F, integrand: F), "failed") if F has LFCAT and R has KONVERT(PATTERN(INT)) and R has PATMAB(INT)

--R pmintegrate : (F, Symbol, OrderedCompletion(F), OrderedCompletion(F)) -> Union(F, "failed") if F has SPFCAT and R has KONVERT(PATTERN(INT)) and R has PATMAB(INT)

--R splitConstant : (F, Symbol) -> Record(const: F, nconst: F)

)--R

)--E 1

)spool
)lisp (bye)

-- PatternMatchIntegration.help --

================================================================================

PatternMatchIntegration examples

================================================================================
PatternMatchIntegration provides functions that use the pattern matcher to find some indefinite and definite integrals involving special functions and found in the literature.

See Also:
- )show PatternMatchIntegration

---

PatternMatchIntegration (INTPM)

Exports:
- pmComplexIntegrate
- pmIntegrate
- pmIntegrate
- splitConstant

— package INTPM PatternMatchIntegration —

)abbrev package INTPM PatternMatchIntegration
++ Author: Manuel Bronstein
++ Date Created: 5 May 1992
++ Date Last Updated: 27 September 1995
++ Description:
++ \texttt{PatternMatchIntegration} provides functions that use
++ the pattern matcher to find some indefinite and definite integrals
++ involving special functions and found in the litterature.

PatternMatchIntegration(R, F): Exports == Implementation where
  R : Join(OrderedSet, RetractableTo Integer, GcdDomain,
           LinearlyExplicitRingOver Integer)
  F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory,
           FunctionSpace R)

  \( N \implies \text{NonNegativeInteger} \)
Z ==> Integer
SY ==> Symbol
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
SUP ==> SparseUnivariatePolynomial F
PAT ==> Pattern Z
RES ==> PatternMatchResult(Z, F)
OFE ==> OrderedCompletion F
REC ==> Record(which: Z, exponent: F, coeff: F)
ANS ==> Record(special:F, integrand:F)
NONE ==> 0
EI ==> 1
ERF ==> 2
SI ==> 3
CI ==> 4
GAM2 ==> 5
CI0 ==> 6
Exports ==> with
  splitConstant: (F, SY) -> Record(const:F, nconst:F)
  if R has ConvertibleTo Pattern Integer and
  if F has LiouvillianFunctionCategory then
    pmComplexintegrate: (F, SY) -> Union(ANS, "failed")
    ++ pmComplexintegrate(f, x) returns either "failed" or
    ++ \spad{\integrate(f, x) = g + \integrate(h, x)}.
    ++ It only looks for special complex integrals that pmintegrate
    ++ does not return.
  pmintegrate: (F, SY) -> Union(ANS, "failed")
  ++ pmintegrate(f, x) returns either "failed" or \spad{\integrate(f, x) = g + \integrate(h, x)}.
  if F has SpecialFunctionCategory then
    pmintegrate: (F, SY, OFE, OFE) -> Union(F, "failed")
    ++ pmintegrate(f, x = a..b) returns the integral of
    ++ \spad{\integrate(f(x)dx) from a to b}
    ++ if it can be found by the built-in pattern matching rules.
Implementation ==> add
  import PatternMatch(Z, F, F)
  import ElementaryFunctionSign(R, F)
  import FunctionSpaceAssertions(R, F)
  import TrigonometricManipulations(R, F)
  import FunctionSpaceAttachPredicates(R, F, F)

mkalist : RES -> AssociationList(SY, F)

pm := new()$SY
 CHAPTER 17.  CHAPTER P

pmw := new pm
pmm := new pm
pms := new pm
pmc := new pm
pma := new pm
pmb := new pm

c := optional(pmc::F)

w := suchThat(optional(pmw::F),
  (x1:F):Boolean +-> empty? variables x1)

s := suchThat(optional(pms::F),
  (x1:F):Boolean +-> empty? variables x1 and real? x1)

m := suchThat(optional(pmm::F),
  (x1:F):Boolean+->(retractIfCan(x1)@Union(Z,"failed") case Z) and x1 >= 0)

spi := sqrt(pi()$F)

half := 1::F / 2::F

mkalist res == construct destruct res

splitConstant(f, x) ==
  not member?(x, variables f) => [f, 1]
  (retractIfCan(f)@Union(K, "failed")) case K => [1, f]
  (u := isTimes f) case List(F) =>
    cc := nc := 1$F
    for g in u::List(F) repeat
      rec := splitConstant(g, x)
      cc := cc * rec.const
      nc := nc * rec.nconst
    [cc, nc]
  (u := isPlus f) case List(F) =>
    rec := splitConstant(first(u::List(F)), x)
    cc := rec.const
    nc := rec.nconst
    for g in rest(u::List(F)) repeat
      rec := splitConstant(g, x)
      if rec.nconst = nc then cc := cc + rec.const
      else if rec.nconst = -nc then cc := cc - rec.const
      else return [1, f]
    [cc, nc]
  if (v := isPower f) case Record(val:F, exponent:Z) then
    vv := v::Record(val:F, exponent:Z)
    (vv.exponent ^= 1) =>
      rec := splitConstant(vv.val, x)
      return [rec.const ** vv.exponent, rec.nconst ** vv.exponent]
  error "splitConstant: should not happen"

if R has ConvertibleTo Pattern Integer and
   R has PatternMatchable Integer then
if F has LiouvillianFunctionCategory then
import ElementaryFunctionSign(R, F)

insqrt  : F -> F
matchei  : (F, SY) -> REC
matcherfei : (F, SY, Boolean) -> REC
matchsici : (F, SY) -> REC
matchli  : (F, SY) -> List F
matchli0 : (F, K, SY) -> List F
matchdilog : (F, SY) -> List F
matchdilog0: (F, K, SY, P, F) -> List F
goodlilog? : (K, P) -> Boolean
gooddilog? : (K, P, F) -> Boolean

-- gooddilog?(k, p) == is?(k, "log":SY) and one? minimumDegree(p, k)
gooddilog?(k, p) == is?(k, "log":SY) and (minimumDegree(p, k) = 1)

gooddilog?(k, p, q) ==
-- is?(k, "log":SY) and one? degree(p, k) and zero? degree(q, k)
is?(k, "log":SY) and (degree(p, k) = 1) and zero? degree(q, k)

-- matches the integral to a result of the form d * erf(u) or d * ei(u)
-- returns [case, u, d]
matcherfei(f, x, comp?) ==
res0 := new()$RES
pat := c * exp(pma::F)
failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
  comp? => [NONE, 0,0]
matchei(f,x)
l := mkalist res
da := differentiate(a := l.pma, x)
d := a * (cc := l.pmc) / da
zero? differentiate(d, x) => [EI, a, d]
comp? or ((u := sign a) case Z) and (u::Z) < 0) =>
d := cc * (sa := insqrt(- a)) / da
zero? differentiate(d, x) => [ERF, sa, - d * spi]
[NONE, 0,0]
[NONE, 0,0]

-- matches the integral to a result of the form d * ei(k * log u)
-- returns [case, k * log u, d]
matchei(f, x) ==
res0 := new()$RES
a := pma::F
pat := c * a**w / log a
failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
  [NONE, 0,0]
l := mkalist res
da := differentiate(a := l.pma, x)
d := (cc := l.pmc) / da
zero? differentiate(d, x) => [EI, (1 + l.pmw) * log a, d] [NONE, 0, 0]

-- matches the integral to a result of the form d * dilog(u) + int(v),
-- returns [u,d,v] or []
matchdilog(f, x) ==
  n := numer f
df := (d := denom f)::F
for k in select_!
  (x1:K):Boolean +-> gooddilog?(x1,n,d),variables n)$List(K) repeat
  not empty?(l := matchdilog0(f, k, x, n, df)) => return l
empty()

-- matches the integral to a result of the form d * dilog(a) + int(v)
-- where k = log(a)
-- returns [a,d,v] or []
matchdilog0(f, k, x, p, q) ==
  zero?(da := differentiate(a := first argument k, x)) => empty()
a1 := 1 - a
d := coefficient(univariate(p, k), 1)::F * a1 / (q * da)
zero? differentiate(d, x) => [a, d, f - d * da * (k::F) / a1]
empty()

-- matches the integral to a result of the form d * li(u) + int(v),
-- returns [u,d,v] or []
matchli(f, x) ==
d := denom f
for k in select_!
  (x1:K):Boolean +-> goodlilog?(x1,d),variables d)$List(K) repeat
  not empty?(l := matchli0(f, k, x)) => return l
empty()

-- matches the integral to a result of the form d * li(a) + int(v)
-- where k = log(a)
-- returns [a,d,v] or []
matchli0(f, k, x) ==
g := (lg := k::F) * f
zero?(da := differentiate(a := first argument k, x)) => empty()
zero? differentiate(d := g / da, x) => [a, d, 0]
ug := univariate(g, k)
(u:=retractIfCan(ug)@Union(SUP,"failed")) case "failed" => empty()
degree(p := u::SUP) > 1 => empty()
zero? differentiate(d := coefficient(p, 0) / da, x) =>
  [a, d, leadingCoefficient p]
empty()

-- matches the integral to a result of the form d * Si(u) or d * Ci(u)
-- returns [case, u, d]
matchesici(f, x) ==
res0 := new()$RES
b := pmb::F
t := tan(a := pma::F)
patsi := c * t / (patden := b + b * t**2)
patci := (c - c * t**2) / patden
patci0 := c / patden
ci0?:Boolean
(ci? := failed?(res := patternMatch(f, convert(patsi)@PAT, res0))
and (ci0? := failed?(res := patternMatch(f, convert(patci)@PAT, res0)))
and failed?(res := patternMatch(f, convert(patci0)@PAT, res0))) =>
[NONE, 0, 0]
l := mkalist res
(b := l.pmb) ^= 2 * (a := l.pma) => [NONE, 0, 0]
db := differentiate(b, x)
d := (cc := l.pmc) / db
zero? differentiate(d, x) =>
  ci? =>
  ci0? => [CI0, b, d / (2::F)]
  [CI, b, d]
  [SI, b, d / (2::F)]
  [NONE, 0, 0]

-- returns a simplified sqrt(y)
insqrt y ==
  rec := froot(y, 2)$PolynomialRoots(IndexedExponents K, K, R, P, F)
--
  one?(rec.exponent) => rec.coef * rec.radicand
  (rec.exponent) = 1 => rec.coef * rec.radicand
  rec.exponent ^=2 => error "insqrt: hould not happen"
  rec.coef * sqrt(rec.radicand)

pmintegrate(f, x) ==
  (rc := splitConstant(f, x)).const ^= 1 =>
  (u := pmintegrate(rc.nconst, x)) case "failed" => "failed"
  rec := u::ANS
  [rc.const * rec.special, rc.const * rec.integrand]
  not empty?(l := matchli(f, x)) => [second l * li first l, third l]
  not empty?(l := matchdilog(f, x)) =>
    [second l * dilog first l, third l]
  cse := (rec := matcherfei(f, x, false)).which
  cse = EI => [rec.coef * Ei(rec.exponent), 0]
  cse = ERF => [rec.coef * erf(rec.exponent), 0]
  cse := (rec := matchsici(f, x)).which
  cse = SI => [rec.coef * Si(rec.exponent), 0]
  cse = CI => [rec.coef * Ci(rec.exponent), 0]
  cse = CI0 => [rec.coef * Ci(rec.exponent)
     + rec.coef * log(rec.exponent), 0]
  "failed"

pmComplexintegrate(f, x) ==
  (rc := splitConstant(f, x)).const ^= 1 =>
  (u := pmintegrate(rc.nconst, x)) case "failed" => "failed"
rec := u::ANS
    [rc.const * rec.special, rc.const * rec.integrand]
cse := (rec := matcherfei(f, x, true)).which
cse = ERF => [rec.coeff * erf(rec.exponent), 0]
"failed"

if F has SpecialFunctionCategory then
  match1 : (F, SY, F, F) -> List F
  formula1 : (F, SY, F, F) -> Union(F, "failed")

-- tries only formula (1) of the Geddes & al, AAECC 1 (1990) paper
formula1(f, x, t, cc) ==
  empty?(l := match1(f, x, t, cc)) => "failed"
mw := first l
zero?(ms := third l) or ((sgs := sign ms) case "failed") => "failed"
    ((sgz := sign(z := (mw + 1) / ms)) case "failed") or (sgz::Z < 0)
    => "failed"
mmi := retract(mm := second l)@Z
sgs * (last l) * ms**(- mmi - 1) *
eval(differentiate(Gamma(x::F), x, mmi::N), [kernel(x)@K, [z]])
-- returns [w, m, s, c] or []
-- matches only formula (1) of the Geddes & al, AAECC 1 (1990) paper
match1(f, x, t, cc) ==
  res0 := new()$RES
  pat := cc * log(t)**m * exp(-t**s)
  not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
    l := mkalist res
    [0, l.pmm, l.pms, l.pmc]
  pat := cc * t**w * exp(-t**s)
  not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
    l := mkalist res
    [l.pmw, 0, l.pms, l.pmc]
  pat := cc / t**w * exp(-t**s)
  not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
    l := mkalist res
    [- l.pmw, 0, l.pms, l.pmc]
  pat := cc * t**w * log(t)**m * exp(-t**s)
  not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
    l := mkalist res
    [l.pmw, l.pmm, l.pms, l.pmc]
  pat := cc / t**w * log(t)**m * exp(-t**s)
  not failed?(res := patternMatch(f, convert(pat)@PAT, res0)) =>
    l := mkalist res
    [- l.pmw, l.pmm, l.pms, l.pmc]
  empty()

pmintegrate(f, x, a, b) ==
  --
  zero? a and one? whatInfinity b =>
  zero? a and ((whatInfinity b) = 1) =>
formula1(f, x, constant(x::F),
    suchThat(c, (x1:F)::Boolean +-> freeOf?(x1, x)))
"failed"

---

| INTPM.dotabb |

"INTPM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPM"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"INTPM" -> "ACF"
"INTPM" -> "FS"

---

package PMKERNEL PatternMatchKernel

--- PatternMatchKernel.input ---

)set break resume
)sys rm -f PatternMatchKernel.output
)spool PatternMatchKernel.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchKernel
--R
--R PatternMatchKernel(S: SetCategory,E: Join(OrderedSet,RetractableTo(Kernel($)),ConvertibleTo(Pattern(S)),PatternMatchable(S))) is a package constructor
--R Abbreviation for PatternMatchKernel is PMKERNEL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMKERNEL
--R
--R-------------------------------- Operations --------------------------------
--R patternMatch : (Kernel(E),Pattern(S),PatternMatchResult(S,E)) -> PatternMatchResult(S,E)
--R
--E 1

)spool
)lisp (bye)

---
This package provides pattern matching functions on kernels.

See Also:
- \texttt{\textbackslash show PatternMatchKernel}

---

**PatternMatchKernel (PMKERNEL)**

Exports:
- \texttt{patternMatch}

---

\texttt{\textbackslash abbrev package PMKERNEL PatternMatchKernel}
\textlt{Author: Manuel Bronstein}\n\textlt{Date Created: 12 Jan 1990}\n\textlt{Date Last Updated: 4 May 1992}\n\textlt{Description:}\n\textlt{This package provides pattern matching functions on kernels.}\n
\texttt{PatternMatchKernel(S, E): Exports == Implementation where}\n\texttt{S: SetCategory}\n\texttt{E: Join(OrderedSet, RetractableTo Kernel \%, ConvertibleTo Pattern S, PatternMatchable S)}
PAT ==> Pattern S
PRS ==> PatternMatchResult(S, E)
POWER ==> "^power":::Symbol
NTHRT ==> "nthRoot":::Symbol

Exports ==> with
  patternMatch: (Kernel E, PAT, PRS) -> PRS
   ++ patternMatch(f(e1,...,en), pat, res) matches the pattern pat
   ++ to spad{f(e1,...,en)}; res contains the variables of pat which
   ++ are already matched and their matches.

Implementation ==> add
  patternMatch : (Kernel E, List PAT, PRS) -> PRS
  patternMatchInner: (Kernel E, PAT, PRS) -> Union(PRS, "failed")

  -- matches the ordered lists ls and lp.
  patternMatchArg(ls, lp, 1) ==
    #ls ^= #lp => failed()
    for p in lp for s in ls repeat
      generic? p and failed?(l := addMatch(p, s, l)) => return failed()
    for p in lp for s in ls repeat
      not(generic? p) and failed?(l := patternMatch(s, p, l)) =>
        return failed()
    l

  patternMatchInner(s, p, l) ==
    generic? p => addMatch(p, s::E, l)
    (u := isOp p) case Record(op:BasicOperator, arg: List PAT) =>
      ur := u::Record(op:BasicOperator, arg: List PAT)
      ur.op = operator s => patternMatchArg(argument s, ur.arg, l)
      failed()
    constant? p =>
      ((v := retractIfCan(p)@Union(Symbol, "failed")) case Symbol)
      and ((w := symbolIfCan s) case Symbol) and
      (v::Symbol = w::Symbol) => l
      failed()
      "failed"

  if E has Monoid then
    patternMatchMonoid: (Kernel E, PAT, PRS) -> Union(PRS, "failed")
    patternMatchOpt : (E, List PAT, PRS, E) -> PRS

    patternMatchOpt(x, lp, 1, id) ==
      (u := optpair lp) case List(PAT) =>
        failed?(l := addMatch(first(u::List(PAT)), id, l)) => failed()
        patternMatch(x, second(u::List(PAT)), 1)
        failed()

    patternMatchMonoid(s, p, 1) ==
      (u := patternMatchInner(s, p, 1)) case PRS => u::PRS
(v := isPower p) case Record(val: PAT, exponent: PAT) =>
  vr := v::Record(val: PAT, exponent: PAT)
is?(op := operator s, POWER) =>
  patternMatchArg(argument s, [vr.val, vr.exponent], l)
is?(op, NTHRT) and ((r := recip(second(arg := argument s))) case E) =>
  patternMatchArg([first arg, r::E], [vr.val, vr.exponent], l)
optional?(vr.exponent) =>
  failed?(l := addMatch(vr.exponent, 1, l)) => failed()
patternMatch(s::E, vr.val, l)
failed()

(w := isTimes p) case List(PAT) =>
  patternMatchOpt(s::E, w::List(PAT), l, 1)
"failed"

if E has AbelianMonoid then
  patternMatch(s, p, l) ==
  (u := patternMatchMonoid(s, p, l)) case PRS => u::PRS
  (w := isPlus p) case List(PAT) =>
    patternMatchOpt(s::E, w::List(PAT), l, 0)
  failed()
else
  patternMatch(s, p, l) ==
  (u := patternMatchMonoid(s, p, l)) case PRS => u::PRS
  failed()
else
  patternMatch(s, p, l) ==
  (u := patternMatchInner(s, p, l)) case PRS => u::PRS
  failed()

—— PMKERNEL.dotabb ——

"PMKERNEL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMKERNEL"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"PMKERNEL" -> "ALIST"

——

package PMLSAGG PatternMatchListAggregate

— PatternMatchListAggregate.input —
---S 1 of 1
)show PatternMatchListAggregate

--R PatternMatchListAggregate(S: SetCategory,R: PatternMatchable(S),L: ListAggregate(R)) is a package constructor
--R Abbreviation for PatternMatchListAggregate is PMLSAGG
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMLSAGG

--R--------------------------------- Operations --------------------------------
--R patternMatch : (L,Pattern(S),PatternMatchListResult(S,R,L)) -> PatternMatchListResult(S,R,L)
--R
--E 1

)spool
)lisp (bye)

—— PatternMatchListAggregate.help ——

====================================================================
PatternMatchListAggregate examples
====================================================================

This package provides pattern matching functions on lists.

See Also:
  o )show PatternMatchListAggregate
PatternMatchListAggregate (PMLSAGG)

Exports:
patternMatch

--- package PMLSAGG PatternMatchListAggregate ---

)abbrev package PMLSAGG PatternMatchListAggregate
++ Author: Manuel Bronstein
++ Date Created: 4 Dec 1989
++ Date Last Updated: 29 Jun 1990
++ Description:
++ This package provides pattern matching functions on lists.

PatternMatchListAggregate(S, R, L):Exports == Implementation where
S: SetCategory
R: PatternMatchable S
L: ListAggregate R

PLR ==> PatternMatchListResult(S, R, L)

Exports ==> with
  patternMatch: (L, Pattern S, PLR) -> PLR
  ++ patternMatch(l, pat, res) matches the pattern pat to the
  ++ list l; res contains the variables of pat which
  ++ are already matched and their matches.

Implementation ==> add
  match: (L, List Pattern S, PLR, Boolean) -> PLR

  patternMatch(l, p, r) ==
  (u := isList p) case "failed" => failed()
  match(l, u::List Pattern S, r, true)

  match(l, lp, r, new?) ==
  empty? lp =>
  empty? l => r
failed()
multiple?(p0 := first lp) =>
  empty? rest lp =>
    if not new? then l := reverse_! l
    makeResult(atoms r, addMatchRestricted(p0,1,lists r,empty()))
  new? => match(reverse l, reverse lp, r, false)
  error "Only one multiple pattern allowed in list"
  empty? 1 => failed()
failed?(r := makeResult(patternMatch(first l,p0,atoms r),lists r)) => failed()
match(rest l, rest lp, r, new?)

——

PMLSAGG.dotabb

"PMLSAGG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMLSAGG"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMLSAGG" -> "FLAGG"

——

package PMPLCAT PatternMatchPolynomialCategory

— PatternMatchPolynomialCategory.input —

)set break resume
/sys rm -f PatternMatchPolynomialCategory.output
/spool PatternMatchPolynomialCategory.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchPolynomialCategory
--R
--R PatternMatchPolynomialCategory(S: SetCategory,E: OrderedAbelianMonoidSup,V: OrderedSet,R: Join(Ring,OrderedSet,PatternMatchable(S)),P: Join(PolynomialCategory(R,E,V),ConvertibleTo(Pattern(S)))) is a package constructor
--R Abbreviation for PatternMatchPolynomialCategory is PMPLCAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMPLCAT
--R
--R--------------------------------- Operations ---------------------------------
--R patternMatch : (P,Pattern(S),PatternMatchResult(S,P),((V,Pattern(S),PatternMatchResult(S,P)) -> PatternMatchResult(S,P)) if V has PATMAB(S)
This package provides pattern matching functions on polynomials.

See Also:
- `)show PatternMatchPolynomialCategory`

Exports:
- `patternMatch`

--R
--E 1

)spool
)lisp (bye)
++ Description:
++ This package provides pattern matching functions on polynomials.

PatternMatchPolynomialCategory(S,E,V,R,P):Exports== Implementation where
  S: SetCategory
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: Join(Ring, OrderedSet, PatternMatchable S)
  P: Join(PolynomialCategory(R, E, V), ConvertibleTo Pattern S)

N ==> NonNegativeInteger
PAT ==> Pattern S
PRS ==> PatternMatchResult(S, P)
RCP ==> Record(val:PAT, exponent:N)
RCX ==> Record(var:V, exponent:N)

Exports ==> with
  patternMatch: (P, PAT, PRS, (V, PAT, PRS) -> PRS) -> PRS
  ++ patternMatch(p, pat, res, vmatch) matches the pattern pat to
  ++ the polynomial p. res contains the variables of pat which
  ++ are already matched and their matches; vmatch is the matching
  ++ function to use on the variables.
  -- This can be more efficient than pushing down when the variables
  -- are recursive over P (e.g. kernels)
  if V has PatternMatchable S then
  patternMatch: (P, PAT, PRS) -> PRS
  ++ patternMatch(p, pat, res) matches the pattern pat to
  ++ the polynomial p; res contains the variables of pat which
  ++ are already matched and their matches.

Implementation ==> add
import PatternMatchTools(S, R, P)
import PatternMatchPushDown(S, R, P)

if V has PatternMatchable S then
  patternMatch(x, p, l) ==
    patternMatch(x, p, l, patternMatch$PatternMatchPushDown(S,V,P))

patternMatch(x, p, l, vmatch) ==
  generic? p => addMatch(p, x, 1)
  (r := retractIfCan(x)@Union(R, "failed")) case R =>
    patternMatch(r::R, p, l)
  (v := retractIfCan(x)@Union(V, "failed")) case V =>
    vmatch(v::V, p, l)
  (u := :isPlus p) case List(PAT) =>
    (lx := :isPlus x) case List(P) =>
      patternMatch(lx::List(P), u::List(PAT),
        (11:List(P))::P --+ */11, 1,)
    (u := optpair(u::List(PAT))) case List(PAT) =>

failed?(l := addMatch(first(u::List(PAT)), 0, l)) => failed()
patternMatch(x, second(u::List(PAT)), l, vmatch)
failed()
(u := isTimes p) case List(PAT) =>
  (lx := isTimes x) case List(P) =>
    patternMatchTimes(lx::List(P), u::List(PAT), l,
    (p1:P, p2: PAT, p3:PRS) :PRS +-> patternMatch(p1, p2, p3, vmatch))
(u := optpair(u::List(PAT))) case List(PAT) =>
    failed?(l := addMatch(first(u::List(PAT)), 1, l)) => failed()
    patternMatch(x, second(u::List(PAT)), l, vmatch)
    failed()
(uu := isPower p) case Record(val: PAT, exponent: PAT) =>
    uur := uu::Record(val: PAT, exponent: PAT)
    (ex := isExpt x) case RCX =>
      failed?(l := patternMatch((ex::RCX).exponent::Integer::P,
      uur.exponent, l, vmatch)) => failed()
      vmatch((ex::RCX).var, uur.val, l)
      optional?(uur.exponent) =>
        failed?(l := addMatch(uur.exponent, 1, l)) => failed()
        patternMatch(x, uur.val, l, vmatch)
        failed()
((ep := isExpt p) case RCP) and ((ex := isExpt x) case RCX) and
  (ex::RCX).exponent = (ep::RCP).exponent =>
    vmatch((ex::RCX).var, (ep::RCP).val, l)
failed()
---R PatternMatchPushDown(S: SetCategory,A: PatternMatchable(S),B: Join(SetCategory,RetractableTo(A))) is a package constructor
---R Abbreviation for PatternMatchPushDown is PMDOWN
---R This constructor is not exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMDOWN
---R
---R----------------------------------- Operations ----------------------------------
---R fixPredicate : (B -> Boolean) -> (A -> Boolean)
---R patternMatch : (A,Pattern(S),PatternMatchResult(S,B)) -> PatternMatchResult(S,B)
---R
---E 1

)spool
)lisp (bye)

---------

— PatternMatchPushDown.help —

====================================================================
PatternMatchPushDown examples
====================================================================

This packages provides tools for matching recursively in type towers.

See Also:
o )show PatternMatchPushDown
PatternMatchPushDown (PMDOWN)

Exports:

fixPredicate  patternMatch

— package PMDOWN PatternMatchPushDown —

)abbrev package PMDOWN PatternMatchPushDown
++ Author: Manuel Bronstein
++ Date Created: 1 Dec 1989
++ Date Last Updated: 16 August 1995
++ Description:
++ This package provides tools for matching recursively in type towers.

PatternMatchPushDown(S, A, B): Exports == Implementation where
  S: SetCategory
  A: PatternMatchable S
  B: Join(SetCategory, RetractableTo A)

  PAT ==> Pattern S
  PRA ==> PatternMatchResult(S, A)
  PRB ==> PatternMatchResult(S, B)
  REC ==> Record(pat:PAT, res:PRA)

Exports ==

  fixPredicate: (B -> Boolean) -> (A -> Boolean)
  ++ fixPredicate(f) returns g defined by g(a) = f(a::B);
  patternMatch: (A, PAT, PRB) -> PRB
  ++ patternMatch(expr, pat, res) matches the pattern pat to the
  ++ expression expr; res contains the variables of pat which
  ++ are already matched and their matches.
  ++ Note that this function handles type towers by changing the predicates
  ++ and calling the matching function provided by \spad{A}.

Implementation ==

import PatternMatchResultFunctions2(S, A, B)
fixPred : Any -> Union(Any, "failed")
inA : (PAT, PRB) -> Union(List A, "failed")
fixPredicates: (PAT, PRB, PRA) -> Union(REC, "failed")
fixList: (List PAT -> PAT, List PAT, PRB, PRA) -> Union(REC,"failed")

fixPredicate f == (a1:A):Boolean +-> f(a1::B)

patternMatch(a, p, l) ==
  (u := fixPredicates(p, l, new())) case "failed" => failed()
  union(l, map((a1:A):B +->a1::B,
    patternMatch(a, (u::REC).pat, (u::REC).res)))

inA(p, l) ==
  (u := getMatch(p, l)) case "failed" => empty()
  (r := retractIfCan(u::B)@Union(A, "failed")) case A => [r::A]
  "failed"

fixList(fn, l, lb, la) ==
  ll:List(PAT) := empty()
  for x in l repeat
    (f := fixPredicates(x, lb, la)) case "failed" => return "failed"
    ll := concat((f::REC).pat, ll)
    la := (f::REC).res
  [fn ll, la]

fixPred f ==
  (u:= retractIfCan(f)$AnyFunctions1(B -> Boolean)) case "failed" =>
  g := fixPredicate(u::(B -> Boolean))
  coerce(g)$AnyFunctions1(A -> Boolean)

fixPredicates(p, lb, la) ==
  (r:=retractIfCan(p)@Union(S,"failed")) case S or quoted? p =>[p,la]
  (u := isOp p) case Record(op:BasicOperator, arg:List PAT) =>
    ur := u::Record(op:BasicOperator, arg:List PAT)
    fixList((l1:List(PAT)):PAT+-> (ur.op) l1, ur.arg, lb, la)
  (u := isPlus p) case List(PAT) =>
    fixList((l1:List(PAT)):PAT +-> reduce("+", l1), la::List(PAT), lb, la)
  (u := isTimes p) case List(PAT) =>
    fixList((l1:List(PAT)):PAT +-> reduce("*", l1), la::List(PAT), lb, la)
  (u := isQuotient p) case Record(num:PAT, den:PAT) =>
    vr := u::Record(num:PAT, den:PAT)
    (fn := fixPredicates(vr.num, lb, la)) case "failed" => "failed"
    la := (fn::REC).res
    (fd := fixPredicates(vr.den, lb, la)) case "failed" => "failed"
    [(fn::REC).pat / (fd::REC).pat, (fd::REC).res]
  (u := isExpt p) case Record(val:PAT, exponent:NonNegativeInteger) =>
    wr := u::Record(val:PAT, exponent: NonNegativeInteger)
    (f := fixPredicates(wr.val, lb, la)) case "failed" => "failed"
    [(f::REC).pat ** wr.exponent, (f::REC).res]
CHAPTER 17. CHAPTER P

(uu := isPower p) case Record(val: PAT, exponent: PAT) =>
  uur := uu::Record(val: PAT, exponent: PAT)
(fv := fixPredicates(uur.val, lb, la)) case "failed" => "failed"
  la := (fv::REC).res
(fe := fixPredicates(uur.exponent, lb, la)) case "failed" =>
  "failed"
[(fv::REC).pat ** (fe::REC).pat, (fe::REC).res]
generic? p =>
  (ua := inA(p, lb)) case "failed" => "failed"
  lp := [if (h := fixPred g) case Any then h::Any else
    return "failed" for g in predicates p]#List(Any)
  q := setPredicates(patternVariable(retract p, constant? p,
    optional? p, multiple? p), lp)
  [q, (empty?(ua::List A) => la; insertMatch(q,first(ua::List A), la))]
error "Should not happen"

— PMDOWN.dotabb —

"PMDOWN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PMDOWN"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"PMDOWN" -> "FLAGG"

— package PMQFCAT PatternMatchQuotientFieldCategory —

package PMQFCAT PatternMatchQuotientFieldCategory

— PatternMatchQuotientFieldCategory.input —

)set break resume
)sys rm -f PatternMatchQuotientFieldCategory.output
)spool PatternMatchQuotientFieldCategory.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PatternMatchQuotientFieldCategory
--R
--R PatternMatchQuotientFieldCategory(S: SetCategory,R: Join(IntegralDomain,PatternMatchable(S),ConvertibleTo(Pattern(S))),Q: QuotientFieldCategory(R)) is a package constructor
--R Abbreviation for PatternMatchQuotientFieldCategory is PMQFCAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMQFCAT
This package provides pattern matching functions on quotients.

See Also:
- \texttt{)show PatternMatchQuotientFieldCategory}
PatternMatchQuotientFieldCategory(S,R,Q):Exports == Implementation where
  S: SetCategory
  R: Join(IntegralDomain, PatternMatchable S, ConvertibleTo Pattern S)
  Q: QuotientFieldCategory R

  PAT ==> Pattern S
  PRQ ==> PatternMatchResult(S, Q)

Exports ==>
  with
    patternMatch: (Q, PAT, PRQ) -> PRQ
    ++ patternMatch(a/b, pat, res) matches the pattern pat to the
    ++ quotient a/b; res contains the variables of pat which
    ++ are already matched and their matches.

Implementation ==>
  add
    import PatternMatchPushDown(S, R, Q)

    patternMatch(x, p, l) ==
      generic? p => addMatch(p, x, l)
      (r := retractIfCan x)@Union(R, "failed") case R =>
        patternMatch(r::R, p, l)
      (u := isQuotient p) case Record(num:PAT, den:PAT) =>
        ur := u::Record(num:PAT, den:PAT)
        failed?(l := patternMatch(numer x, ur.num, l)) => l
        patternMatch(denom x, ur.den, l)
        failed()
package PATRES2 PatternMatchResultFunctions2

— PatternMatchResultFunctions2.input —

)set break resume
)sys rm -f PatternMatchResultFunctions2.output
)spool PatternMatchResultFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchResultFunctions2

--R PatternMatchResultFunctions2(R: SetCategory,A: SetCategory,B: SetCategory) is a package constructor
--R Abbreviation for PatternMatchResultFunctions2 is PATRES2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PATRES2
--R
--R------------------------------- Operations --------------------------------
--R map : ((A -> B),PatternMatchResult(R,A)) -> PatternMatchResult(R,B)
--R
--E 1

)spool
)lisp (bye)

———

— PatternMatchResultFunctions2.help ——

====================================================================
PatternMatchResultFunctions2 examples
====================================================================

Lifts maps to pattern matching results.

See Also:
o )show PatternMatchResultFunctions2

———
PatternMatchResultFunctions2 (PATRES2)

Exports:
map

— package PATRES2 PatternMatchResultFunctions2 —

)abbrev package PATRES2 PatternMatchResultFunctions2
+ Author: Manuel Bronstein
+ Date Created: 1 Dec 1989
+ Date Last Updated: 14 Dec 1989
+ Description:
  + Lifts maps to pattern matching results.

PatternMatchResultFunctions2(R, A, B): Exports == Implementation where
  R: SetCategory
  A: SetCategory
  B: SetCategory

Exports ==> with
  map: (A -> B, PatternMatchResult(R, A)) -> PatternMatchResult(R, B)
  ++ map(f, [(v1,a1),...(vn,an)]) returns the matching result
  ++ [(v1,f(a1)),...(vn,f(an))].

Implementation ==> add
  map(f, r) ==
  failed? r => failed()
  construct [[rec.key, f(rec.entry)] for rec in destruct r]

— PATRES2.dotabb —

"PATRES2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PATRES2"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]

---
package PMSYM PatternMatchSymbol

--- PatternMatchSymbol.input ---

)set break resume
)sys rm -f PatternMatchSymbol.output
)spool PatternMatchSymbol.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PatternMatchSymbol
--R
--R PatternMatchSymbol(S: SetCategory) is a package constructor
--R Abbreviation for PatternMatchSymbol is PMSYM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMSYM
--R
--R----------------------------------- Operations -----------------------------------
--R patternMatch : (Symbol,Pattern(S),PatternMatchResult(S,Symbol)) -> PatternMatchResult(S,Symbol)
--R
--E 1

)spool
)lisp (bye)

---

--- PatternMatchSymbol.help ---

====================================================================
PatternMatchSymbol examples
====================================================================

This package provides pattern matching functions on symbols.

See Also:
  o )show PatternMatchSymbol
PatternMatchSymbol (PMSYM)

Exports:
patternMatch

— package PMSYM PatternMatchSymbol —

)abbrev package PMSYM PatternMatchSymbol
++ Author: Manuel Bronstein
++ Date Created: 9 Jan 1990
++ Date Last Updated: 20 June 1991
++ Description:
++ This package provides pattern matching functions on symbols.

PatternMatchSymbol(S:SetCategory): with
  patternMatch: (Symbol, Pattern S, PatternMatchResult(S, Symbol)) ->
    PatternMatchResult(S, Symbol)
    ++ patternMatch(expr, pat, res) matches the pattern pat to the
    ++ expression expr; res contains the variables of pat which
    ++ are already matched and their matches (necessary for recursion).
  == add
  import TopLevelPatternMatchControl

patternMatch(s, p, l) ==
  generic? p => addMatch(p, s, l)
  constant? p =>
    ((u := retractIfCan(p)@Union(Symbol, "failed")) case Symbol)
    and (u::Symbol) = s => l
    failed()
  failed()
package PMTOOLS PatternMatchTools

--- PatternMatchTools.input ---

)set break resume
)sys rm -f PatternMatchTools.output
)spool PatternMatchTools.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show PatternMatchTools
--R
--R PatternMatchTools(S: SetCategory,R: Join(Ring,OrderedSet),P: Join(Ring,ConvertibleTo(Pattern(S)),RetractableTo(R))) is a package constructor
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PMTOOLS
--R
--R----------------------------------- Operations -----------------------------------
--R patternMatch : (List(P),List(Pattern(S)),(List(P) -> P),PatternMatchResult(S,P),((P,Pattern(S),PatternMatchResult(S,P)) -> PatternMatchResult(S,P))
--R patternMatchTimes : (List(P),List(Pattern(S)),PatternMatchResult(S,P),((P,Pattern(S),PatternMatchResult(S,P)) -> PatternMatchResult(S,P))
--R
--E 1

)spool
)lisp (bye)

--- PatternMatchTools.help ---

====================================================================
PatternMatchTools examples
This package provides tools for the pattern matcher.

See Also:
- )show PatternMatchTools

---

PatternMatchTools (PMTOOLS)

Exports:
- patternMatch
- patternMatchTimes

---

)abbrev package PMTOOLS PatternMatchTools
++ Author: Manuel Bronstein
++ Date Created: 13 Mar 1990
++ Date Last Updated: 4 February 1992
++ Description:
++ This package provides tools for the pattern matcher.

PatternMatchTools(S, R, P): Exports == Implementation where
  S: SetCategory
  R: Join(Ring, OrderedSet)
  P: Join(Ring, ConvertibleTo Pattern S, RetractableTo R)

PAT ==> Pattern S
PRS ==> PatternMatchResult(S, P)
REC ==> Record(res:PRS, s:List P)
RC ==> Record(pat:List PAT, s:List P)
Exports ==> with

\texttt{patternMatch: (List P, List PAT, List P -> P, PRS,}
\texttt{(P, PAT, PRS) -> PRS) -> PRS}

\texttt{++ patternMatch(lsubj, lpat, op, res, match) matches the list}
\texttt{++ of patterns lpat to the list of subjects lsubj, allowing for}
\texttt{++ commutativity; op is the operator such that op(lpat) should}
\texttt{++ match op(lsubj) at the end, r contains the previous matches,}
\texttt{++ and match is a pattern-matching function on P.}

\texttt{patternMatchTimes: (List P, List PAT, PRS,}
\texttt{(P, PAT, PRS) -> PRS) -> PRS}

\texttt{++ patternMatchTimes(lsubj, lpat, res, match) matches the}
\texttt{++ product of patterns \texttt{\(\text{reduce}(\ast, lpat)\)}}
\texttt{++ to the product of subjects \texttt{\(\text{reduce}(\ast, lsubj)\)}};
\texttt{++ r contains the previous matches}
\texttt{++ and match is a pattern-matching function on P.}

Implementation ==> add

\texttt{import PatternFunctions1(S, P)}

\texttt{preprocessList: (PAT, List P, PRS) -> Union(List P, "failed")}

\texttt{selBestGen : List PAT -> List PAT}

\texttt{negConstant : List P -> Union(P, "failed")}

\texttt{findMatch : (PAT, List P, PRS, P, (P, PAT, PRS) -> PRS) -> REC}

\texttt{tryToMatch : (List PAT, REC, P, (P, PAT, PRS) -> PRS) ->}
\texttt{Union(REC, "failed")}

\texttt{filterMatchedPatterns: (List PAT, List P, PRS) -> Union(REC, "failed")}

\texttt{mn1 := convert(-1::P)@Pattern(S)}

\texttt{negConstant l ==}
\texttt{for x in l repeat}
\texttt{((r := retractIfCan(x)@Union(R, "failed")) case R) and}
\texttt{(r::R < 0) => return x}
\texttt{"failed"}

-- tries to match the list of patterns lp to the list of subjects rc.s
-- with rc.res being the list of existing matches.
-- updates rc with the new result and subjects still to match
\texttt{tryToMatch(lp, rc, ident, pmatch) ==}
\texttt{rec:REC := [l := rc.res, ls := rc.s]}
\texttt{for p in lp repeat}
\texttt{rec := findMatch(p, ls, l, ident, pmatch)}
\texttt{failed?(l := rec.res) => return "failed"}
\texttt{ls := rec.s}
\texttt{rec}

-- handles -1 in the pattern list.
\texttt{patternMatchTimes(ls, lp, l, pmatch) ==}
\texttt{member(mn1, lp) =>}
\texttt{(u := negConstant ls) case "failed" => failed()}
if (u::P ^= -1::P) then ls := concat(-u::P, ls)

patternMatch(remove(u::P, ls), remove(mn1, lp),
  (l1:List(P)):P +-> */l1, 1, pmatch)

patternMatch(ls, lp, (l1:List(P)):P +-> */l1, 1, pmatch)

-- finds a match for p in ls, try not to match to a "bad" value
findMatch(p, ls, 1, ident, pmatch) ==
  bad:List(P) :=
    generic? p => setIntersection(badValues p, ls)
    empty()
  l1:PRS := failed()
  for x in setDifference(ls, bad)
    while (t := x; failed?(l1 := pmatch(x, p, l))) repeat 0
    failed? l1 =>
    for x in bad
      while (t := x; failed?(l1 := pmatch(x, p, l))) repeat 0
    failed? l1 => [addMatchRestricted(p, ident, 1, ident), ls]
    [l1, remove(t, ls)]

-- filters out pattern if it's generic and already matched.
preprocessList(pattern, ls, l) ==
  generic? pattern =>
  (u := getMatch(pattern, l)) case P =>
    member?(u::P, ls) => [u::P]
    "failed"
    empty()
  empty()  

-- take out already matched generic patterns
filterMatchedPatterns(lp, ls, l) ==
  for p in lp repeat
    (rc := preprocessList(p, ls, l)) case "failed" => return "failed"
    if not empty?(rc::List(P)) then
      lp := remove(p, lp)
      ls := remove(first(rc::List(P)), ls)
    [lp, ls]

-- select a generic pattern with no predicate if possible
selBestGen l ==
  ans := empty()$List(PAT)
  for p in l | generic? p repeat
    ans := [p]
    not hasPredicate? p => return ans
  ans

-- matches unordered lists ls and lp
patternMatch(ls, lp, op, 1, pmatch) ==
  ident := op empty()
  (rc := filterMatchedPatterns(lp, ls, l)) case "failed" => return failed()
lp := (rc::RC).pat
ls := (rc::RC).s
empty? lp => 1
#(lpm := select(optional?, lp)) > 1 =>
  error "More than one optional pattern in sum/product"
(#ls + #lpm) < #lp => failed()
if (not empty? lpm) and (#ls + 1 = #lp) then
  lp := remove(first lpm, lp)
  failed?(l := addMatch(first lpm, ident, l)) => return l
#(lpm := select(multiple?, lp)) > 1 =>
  error "More than one expandable pattern in sum/product"
#ls > #lp and empty? lpm and empty?(lpm := selBestGen lp) =>
  failed()
if not empty? lpm then lp := remove(first lpm, lp)
-- this is the order in which we try to match predicates
-- 11 = constant patterns (i.e. 'x, or sin('x))
l1 := select(constant?, lp)
-- 12 = patterns with a predicate attached to them
l2 := select((p1:PAT):Boolean+->hasPredicate? p1 and not constant? p1,lp)
-- 13 = non-generic patterns without predicates
l3 := sort_!((z1:PAT,z2:PAT):Boolean+->depth(z1) > depth(z2),
  select((p2:PAT):Boolean+->not(hasPredicate? p2
  or generic? p2 or constant? p2),lp))
-- 14 = generic patterns with predicates
l4 := select((p1:PAT):Boolean+->generic? p1 and
  not(hasPredicate? p1 or constant? p1), lp)
rec:=REC := [l, ls]
(u := tryToMatch(l1, rec, ident, pmatch)) case "failed" =>
  failed()
(u := tryToMatch(l2, u::REC, ident, pmatch)) case "failed" =>
  failed()
(u := tryToMatch(l3, u::REC, ident, pmatch)) case "failed" =>
  failed()
rec := u::REC
(rc := filterMatchedPatterns(l4,rec.s,rec.res)) case "failed" => failed()
rec := [rec.res, (rc::RC).s]
(u := tryToMatch((rc::RC).pat,rec,ident,pmatch)) case "failed" => failed()
rec := u::REC
l := rec.res
ls := rec.s
empty? lp =>
  empty? ls => 1
  failed()
addMatch(first lpm, op ls, l)

— PMTOOLS.dotabb —
package PERMAN Permanent

— Permanent.input —

)set break resume
)spool Permanent.output
)set message test on
)set message auto off
)clear all

--S 1 of 4
kn n ==
    r : MATRIX INT := new(n,n,1)
    for i in 1..n repeat
        r.i.i := 0
    r
Type: Void

--E 1

--S 2 of 4
permanent(kn(5) :: SQMATRIX(5,INT))
Type: PositiveInteger

--E 2

--S 3 of 4
[permanent(kn(n) :: SQMATRIX(n,INT)) for n in 1..13]
Type: List(NonNegativeInteger)
Permanent examples
====================================================================

The package Permanent provides the function permanent for square matrices. The permanent of a square matrix can be computed in the same way as the determinant by expansion of minors except that for the permanent the sign for each element is 1, rather than being 1 if the row plus column indices is positive and -1 otherwise. This function is much more difficult to compute efficiently than the determinant. An example of the use of permanent is the calculation of the n-th derangement number, defined to be the number of different possibilities for n couples to dance but never with their own spouse.

Consider an n by n matrix with entries 0 on the diagonal and 1 elsewhere. Think of the rows as one-half of each couple (for example, the males) and the columns the other half. The permanent of such a matrix gives the desired derangement number.

```
kn n ==
  r : MATRIX INT := new(n,n,1)
  for i in 1..n repeat
    r.i.i := 0
  r
```
Here are some derangement numbers, which you see grow quite fast.

\[
\text{permanent}(\text{kn}(5) :: \text{SQMATRIX}(5, \text{INT}))
\]

\[
[\text{permanent}(\text{kn}(n) :: \text{SQMATRIX}(n, \text{INT})) \text{ for } n \text{ in 1..13}]
\]

See Also:
- \text{show Permanent}

---

**Permanent (PERMAN)**

Exports:
- permanent

---

)abbrev package PERMAN Permanent
++ Authors: Johannes Grabmeier, Oswald Gschnitzer
++ Date Created: 7 August 1989
++ Date Last Updated: 23 August 1990
++ References:
++ Description:
  ++ Permanent implements the functions permanent, the permanent for square matrices.
Permanent(n : PositiveInteger, R : Ring with commutative("*")):
public == private where
  I ==> Integer
  L ==> List
  V ==> Vector
  SM ==> SquareMatrix(n,R)
  VECTPKG1 ==> VectorPackage1(I)
  NNI ==> NonNegativeInteger
  PI ==> PositiveInteger
  GRAY ==> GrayCode

public ==> with

  permanent: SM -> R
  ++ permanent(x) computes the permanent of a square matrix x.
  ++ The permanent is equivalent to
  ++ the \spadfun{determinant} except that coefficients have
  ++ no change of sign. This function
  ++ is much more difficult to compute than the
  ++ determinant. The formula used is by H.J. Ryser,
  ++ improved by [Nijenhuis and Wilf, Ch. 19].
  ++ Note that permanent(x) choose one of three algorithms, depending
  ++ on the underlying ring R and on n, the number of rows (and
  ++ columns) of x:
  ++ if 2 has an inverse in R we can use the algorithm of
  ++ [Nijenhuis and Wilf, ch.19,p.158]; if 2 has no inverse,
  ++ some modifications are necessary:
  ++ different from 2 (the algorithm works if and only 2 is not a
  ++ zero-divisor of R and characteristic(R) \not= 2,
  ++ but how to check that for any given R?),
  ++ the local function permanent2 is called;
  ++ else, the local function permanent3 is called
  ++ (works for all commutative rings R).

private ==> add

  -- local functions:

  permanent2: SM -> R

  permanent3: SM -> R

x : SM
a,b : R
i,j,k,l : I

permanent3(x) ==
  -- This algorithm is based upon the principle of inclusion-
  -- exclusion. A Gray-code is used to generate the subsets of
-- 1,...,n. This reduces the number of additions needed in
-- every step.
sgn : R := 1
k : R
a := 0$R
vv : V V I := firstSubsetGray(n)$GRAY
-- For the meaning of the elements of vv, see GRAY.
w : V R := new(n,0$R)
j := 1 -- Will be the number of the element changed in subset
while j ^= (n+1) repeat -- we sum over all subsets of (1,...,n)
  sgn := -sgn
  b := sgn
  if vv.1.j = 1 then k := -1
  else k := 1 -- was that element deleted(k=-1) or added(k=1)?
  for i in 1...(n::I) repeat
    w.i := w.i +$R k *$R x(i,j)
    b := b *$R w.i
  a := a +$R b
  vv := nextSubsetGray(vv,n)$GRAY
  j := vv.2.1
if odd?(n) then a := -a
a

permanent(x) ==
-- If 2 has an inverse in R, we can spare half of the calcu-
-- lation needed in "permanent3": This is the algorithm of
-- [Nijenhuis and Wilf, ch.19,p.158]
  n = 1 => x(1,1)
  two : R := (2:I) :: R
  half : Union(R,"failed") := recip(two)
  if (half case "failed") then
    if n < 7 then return permanent3(x)
    else return permanent2(x)
  sgn : R := 1
  a := 0$R
  w : V R := new(n,0$R)
-- w.i will be at first x.i and later lambda.i in
-- [Nijenhuis and Wilf, p.158, (24a) resp.(26)].
  rowi : V R := new(n,0$R)
  for i in 1..n repeat
    rowi := row(x,i) :: V R
    b := 0$R
    for j in 1..n repeat
      b := b + rowi.j
      w.i := rowi(n) - (half*b)$R
  vv : V V I := firstSubsetGray((n-1): PI)$GRAY
-- For the meaning of the elements of vv, see GRAY.
  n :: I
  b := 1
for i in 1..n repeat
b := b * w.i
a := a+b
j := 1 -- Will be the number of the element changed in subset
while j ^= n repeat -- we sum over all subsets of (1,...,n-1)
sgn := -sgn
b := sgn
if vv.1.j = 1 then k := -1
else k := 1 -- was that element deleted(k=-1) or added(k=1)?
for i in 1..n repeat
w.i := w.i +$R k *$R x(i,j)
b := b *$R w.i
a := a +$R b
vv := nextSubsetGray(vv,(n-1) : PI)$GRAY
j := vv.2.1
if not odd?(n) then a := -a
two * a

permanent2(x) ==
c : R := 0
sgn : R := 1
if (not (R has IntegralDomain))
  -- or (characteristic($R = (2:NNI))
  -- compiler refuses to compile the line above !!
  or (sgn + sgn = c)
then return permanent3(x)
-- This is a slight modification of permanent which is
-- necessary if 2 is not zero or a zero-divisor in R, but has
-- no inverse in R.
n = 1 => x(1,1)
two : R := (2:I) :: R
a := 0$R
w : V R := new(n,0$R)
-- w.i will be at first x.i and later lambda.i in
-- [Wijehuis and Wilf, p.158, (24a) resp.(26)].
rowi : V R := new(n,0$R)
for i in 1..n repeat
rowi := row(x,i) :: V R
b := 0$R
for j in 1..n repeat
b := b + rowi.j
w.i := (two*(rowi(n)))$R - b
vv : V V I := firstSubsetGray((n-1) : PI)$GRAY
n :: I
b := 1
for i in 1..n repeat
b := b *$R w.i
a := a +$R b
j := 1 -- Will be the number of the element changed in subset
while j ^= n repeat -- we sum over all subsets of (1,...,n-1)
sgn := -sgn
b := sgn
if vv.1.j = 1 then k := -1
else k := 1 -- was that element deleted(k=-1) or added(k=1)?
c := k * two
for i in 1..n repeat
  w.i := w.i +$R c *$R x(i,j)
b := b *$R w.i
a := a +$R b
vv := nextSubsetGray(vv,(n-1) : PI)$GRAY
j := vv.2.1
if not odd?(n) then a := -a
b := two ** ((n-1):NNI)
(a exquo b) :: R

—— PERMAN.dotabb ——

"PERMAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PERMAN"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"PERMAN" -> "IVECTOR"

——

package PGE PermutationGroupExamples

—— PermutationGroupExamples.input ——

)set break resume
)sys rm -f PermutationGroupExamples.output
)spool PermutationGroupExamples.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PermutationGroupExamples
--R
--R PermutationGroupExamples is a package constructor
--R Abbreviation for PermutationGroupExamples is PGE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PGE
--R
PermutationGroupExamples provides permutation groups for some classes of groups: symmetric, alternating, dihedral, cyclic, direct products of cyclic, which are in fact the finite abelian groups of symmetric groups called Young subgroups. Furthermore, Rubik's group as permutation group of 48 integers and a list of sporadic simple groups derived from the atlas of finite groups.

See Also:
o )show PermutationGroupExamples
PermutationGroupExamples (PGE)

Exports:
abelianGroup alternatingGroup cyclicGroup dihedralGroup janko2
mathieu11 mathieu12 mathieu22 mathieu23 mathieu24
rubiksGroup symmetricGroup youngGroup

— package PGE PermutationGroupExamples —

)abbrev package PGE PermutationGroupExamples
++ Authors: M. Weller, G. Schneider, J. Grabmeier
++ Date Created: 20 February 1990
++ Date Last Updated: 09 June 1990
++ References:
++ J. Conway, R. Curtis, S. Norton, R. Parker, R. Wilson:
++ Description:
++ PermutationGroupExamples provides permutation groups for
++ some classes of groups: symmetric, alternating, dihedral, cyclic,
++ direct products of cyclic, which are in fact the finite abelian groups
++ of symmetric groups called Young subgroups.
++ Furthermore, Rubik’s group as permutation group of 48 integers and a list
++ of sporadic simple groups derived from the atlas of finite groups.

PermutationGroupExamples():public == private where

  L    ===> List
  I    ===> Integer
  PI   ===> PositiveInteger
  NNI  ===> NonNegativeInteger
  PERM ===> Permutation
  PERMGROUP ===> PermutationGroup
public => with

symmetricGroup: PI -> PERMGRP I
  ++ symmetricGroup(n) constructs the symmetric group Sn
  ++ acting on the integers 1,...,n, generators are the
  ++ n-cycle (1,...,n) and the 2-cycle (1,2).
symmetricGroup: L I -> PERMGRP I
  ++ symmetricGroup(li) constructs the symmetric group acting on
  ++ the integers in the list li, generators are the
  ++ cycle given by li and the 2-cycle (li.1,li.2).
  ++ Note that duplicates in the list will be removed.
alternatingGroup: PI -> PERMGRP I
  ++ alternatingGroup(n) constructs the alternating group An
  ++ acting on the integers 1,...,n, generators are in general the
  ++ n-2-cycle (3,...,n) and the 3-cycle (1,2,3)
  ++ if n is odd and the product of the 2-cycle (1,2) with
  ++ n-2-cycle (3,...,n) and the 3-cycle (1,2,3)
  ++ if n is even.
alternatingGroup: L I -> PERMGRP I
  ++ alternatingGroup(li) constructs the alternating group acting
  ++ on the integers in the list li, generators are in general the
  ++ n-2-cycle (li.3,...,li.n) and the 3-cycle
  ++ (li.1,li.2,li.3), if n is odd and
  ++ product of the 2-cycle (li.1,li.2) with
  ++ n-2-cycle (li.3,...,li.n) and the 3-cycle
  ++ (li.1,li.2,li.3), if n is even.
  ++ Note that duplicates in the list will be removed.
abelianGroup: L PI -> PERMGRP I
  ++ abelianGroup([n1,...,nk]) constructs the abelian group that
  ++ is the direct product of cyclic groups with order ni.
cyclicGroup: PI -> PERMGRP I
  ++ cyclicGroup(n) constructs the cyclic group of order n acting
  ++ on the integers 1,...,n.
cyclicGroup: L I -> PERMGRP I
  ++ cyclicGroup([i1,...,ik]) constructs the cyclic group of
  ++ order k acting on the integers i1,...,ik.
  ++ Note that duplicates in the list will be removed.
dihedralGroup: PI -> PERMGRP I
  ++ dihedralGroup(n) constructs the dihedral group of order 2n
  ++ acting on integers 1,...,N.
dihedralGroup: L I -> PERMGRP I
  ++ dihedralGroup([i1,...,ik]) constructs the dihedral group of
  ++ order 2k acting on the integers out of i1,...,ik.
  ++ Note that duplicates in the list will be removed.
mathieu11: L I -> PERMGRP I
  ++ mathieu11(li) constructs the mathieu group acting on the 11
  ++ integers given in the list li.
  ++ Note that duplicates in the list will be removed.
  ++ error, if li has less or more than 11 different entries.
mathieu11: () -> PERMGRP I
++ mathieu11 constructs the mathieu group acting on the
++ integers 1,...,11.
mathieu12: L I -> PERMGRP I
++ mathieu12(li) constructs the mathieu group acting on the 12
++ integers given in the list li.
++ Note that duplicates in the list will be removed
++ Error: if li has less or more than 12 different entries.
mathieu12: () -> PERMGRP I
++ mathieu12 constructs the mathieu group acting on the
++ integers 1,...,12.
mathieu22: L I -> PERMGRP I
++ mathieu22(li) constructs the mathieu group acting on the 22
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 22 different entries.
mathieu22: () -> PERMGRP I
++ mathieu22 constructs the mathieu group acting on the
++ integers 1,...,22.
mathieu23: L I -> PERMGRP I
++ mathieu23(li) constructs the mathieu group acting on the 23
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 23 different entries.
mathieu23: () -> PERMGRP I
++ mathieu23 constructs the mathieu group acting on the
++ integers 1,...,23.
mathieu24: L I -> PERMGRP I
++ mathieu24(li) constructs the mathieu group acting on the 24
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 24 different entries.
mathieu24: () -> PERMGRP I
++ mathieu24 constructs the mathieu group acting on the
++ integers 1,...,24.
janko2: L I -> PERMGRP I
++ janko2(li) constructs the janko group acting on the 100
++ integers given in the list li.
++ Note that duplicates in the list will be removed.
++ Error: if li has less or more than 100 different entries.
janko2: () -> PERMGRP I
++ janko2 constructs the janko group acting on the
++ integers 1,...,100.
rubiksGroup: () -> PERMGRP I
++ rubiksGroup constructs the permutation group representing
++ Rubic’s Cube acting on integers 10*i+j for
++ 1 <= i <= 6, 1 <= j <= 8.
++ The faces of Rubik’s Cube are labelled in the obvious way
++ Front, Right, Up, Down, Left, Back and numbered from 1 to 6
++ in this given ordering, the pieces on each face
++ (except the unmoveable center piece) are clockwise numbered
**PACKAGE PGE PERMUTATIONGROUPEXAMPLES**

++ from 1 to 8 starting with the piece in the upper left corner. The moves of the cube are represented as permutations on these pieces, represented as a two digit integer \(ij\) where \(i\) is the number of the face (1 to 6) and \(j\) is the number of the piece on this face.
++ The remaining ambiguities are resolved by looking at the 6 generators, which represent a 90 degree turns of the faces, or from the following pictorial description.
++ Permutation group representing Rubik’s Cube acting on integers \(10i+j\) for \(1 \leq i \leq 6, 1 \leq j \leq 8\).
++
++ \begin{verbatim}
++ Rubik’s Cube: +-----+ +-- B where: marks Side # :
++ / U /|/
++ / / | F(ront) <-> 1
++ L --> +-----+ R| R(ight) <-> 2
++ | | + U(p) <-> 3
++ | F | / D(own) <-> 4
++ | |/ L(eft) <-> 5
++ +-----+ B(ack) <-> 6
++ ^
++ |
++ D
++
++ The Cube’s surface:
++ +---+ The pieces on each side
++ |567| (except the unmoveable center
++ |4U8| piece) are clockwise numbered
++ |321| from 1 to 8 starting with the
++ +---+ piece in the upper left
++ +---------+ corner (see figure on the
++ [781|123|345] left). The moves of the cube
++ [6L2|8F4|2R6] are represented as
++ [543|765|187] permutations on these pieces.
++ +----------+ Each of the pieces is
++ |123| represented as a two digit
++ |8D4| integer \(ij\) where \(i\) is the
++ |765| \# of the side ( 1 to 6 for
++ +---+ F to B (see table above ))
++ |567| and \(j\) is the \# of the piece.
++ |4B8|
++ |321|
++ +---+
++ \end{verbatim}

++ \begin{verbatim}
++ \end{verbatim}
++ youngGroup: L I -> PERMGRP I
++ youngGroup([n1,...,nk]) constructs the direct product of the symmetric groups \(S_{n1},...,S_{nk}\).
++ youngGroup: Partition -> PERMGRP I
++ youngGroup(lambda) constructs the direct product of the symmetric groups given by the parts of the partition lambda.
private ==> add

-- import the permutation and permutation group domains:

import PERM I
import PERMGRP I

-- import the needed map function:

import ListFunctions2(L L I,PERM I)

-- the internal functions:

llli2gp(l:L L L I):PERMGRP I ==
  --++ Converts an list of permutations each represented by a list
  --++ of cycles ( each of them represented as a list of Integers )
  --++ to the permutation group generated by these permutations.
  (map(cycles,l))::PERMGRP I

li1n(n:I):L I ==
  --++ constructs the list of integers from 1 to n
  [i for i in 1..n]

-- definition of the exported functions:

youngGroup(l:L I):PERMGRP I ==
  gens:= nil()$(L L L I)
  element:= 1
  for n in l | n > 1 repeat
    gens:=cons(list [i for i in element..(element+n-1)], gens)
    if n >= 3 then gens := cons([[element,element+1]],gens)
    element:=element+n
  llll2gp
  #gens = 0 => [[[1]]]
  gens

youngGroup(lambda : Partition):PERMGRP I ==
  youngGroup(convert(lambda)$Partition)

rubiksGroup():PERMGRP I ==
  -- each generator represents a 90 degree turn of the appropriate
  -- side.
  f:L L I:=
    [[11,13,15,17],[12,14,16,18],[51,31,21,41],[53,33,23,43],[52,32,22,42]]
  r:L L I:=
    [[21,23,25,27],[22,24,26,28],[13,37,67,43],[15,31,61,45],[14,38,68,44]]
  u:L L I:=
    [[31,33,35,37],[32,34,36,38],[13,51,63,25],[11,57,61,23],[12,58,62,24]]
  d:L L I:=
    [[41,43,45,47],[42,44,46,48],[17,21,67,55],[15,27,65,53],[16,28,66,54]]
  l:L L I:=

\[
[[51,53,55,57],[52,54,56,58],[11,41,65,35],[17,47,63,33],[18,48,64,34]]
\]

\[
[[61,63,65,67],[62,64,66,68],[45,25,35,55],[47,27,37,57],[46,26,36,56]]
\]

mathieu11(l:L I):PERMGRP I ==

-- permutations derived from the ATLAS
l:=[\[l.1,l.10\],\[l.2,l.8\],\[l.3,l.11\],\[l.5,l.7\]]
llli2gp [a,[[l.1,l.4,l.7,l.6],[l.2,l.11,l.10,l.9]]]

mathieu11():PERMGRP I == mathieu11 li1n 11

mathieu12(l:L I):PERMGRP I ==

-- permutations derived from the ATLAS
l:=[\[l.1,l.2,l.3,l.4,l.5,l.6,l.7,l.8,l.9,l.10,l.11\]]
llli2gp [a,[[l.1,l.6,l.5,l.8,l.3,l.7,l.4,l.2,l.9,l.10],[l.11,l.12]]]

mathieu12():PERMGRP I == mathieu12 li1n 12

mathieu22(l:L I):PERMGRP I ==

-- permutations derived from the ATLAS
l:=[\[l.1,l.2,l.4,l.8,l.16,l.9,l.18,l.13,l.3,l.6,l.12\],
\[l.5,l.10,l.20,l.17,l.11,l.22,l.21,l.19,l.15,l.7,l.14\]]
llli2gp [a,b]

mathieu22():PERMGRP I == mathieu22 li1n 22

mathieu23(l:L I):PERMGRP I ==

-- permutations derived from the ATLAS
l:=[\[l.1,l.2,l.3,l.4,l.5,l.6,l.7,l.8,l.9,l.10,l.11,l.12,l.13,l.14,l.15,l.16,l.17,l.18,l.19,l.20,l.21,l.22,l.23\]]
llli2gp [a,b]

mathieu23():PERMGRP I == mathieu23 li1n 23

mathieu24(l:L I):PERMGRP I ==

-- permutations derived from the ATLAS
l:=removeDuplicates l
#l ^= 24 => error "Exactly 24 integers for mathieu24 needed !"

mathieu24():PERMGRP I == mathieu24 li1n 24

janko2(l:L I):PERMGRP I ==
-- permutations derived from the ATLAS
l:=removeDuplicates l
#l ^= 100 => error "Exactly 100 integers for janko2 needed !"

abelianGroup(l:L PI):PERMGRP I ==
gen:= nil()$(L L L I)
element:= 1
for n in l | n > 1 repeat
gen:=cons( list [i for i in element..(element+n-1) ], gens )
element:=element+n
lli2gp [a,b]

janko2():PERMGRP I == janko2 li1n 100
alternatingGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #1 = 0 =>
    error "Cannot construct alternating group on empty set"
  #1 < 3 => llli2gp [[l.1]]
  #1 = 3 => llli2gp [[l.1,l.1,2,l.1,3]]
  tmp:= [l.i for i in 3..(#l)]
  gens:L L L I:=[[tmp],[[l.1,l.1,2,l.1,3]]]
  odd?(#1) => llli2gp gens
  gens.1 := cons([l.1,l.1,2],gens.1)
  llli2gp gens

alternatingGroup(n:PI):PERMGRP I == alternatingGroup li1n n

symmetricGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #1 = 0 => error "Cannot construct symmetric group on empty set !"
  #1 < 3 => llli2gp [[l]]
  llli2gp [[l],[[l.1,l.1,2]]]

symmetricGroup(n:PI):PERMGRP I == symmetricGroup li1n n

cyclicGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #1 = 0 => error "Cannot construct cyclic group on empty set"
  llli2gp [[l]]

cyclicGroup(n:PI):PERMGRP I == cyclicGroup li1n n

dihedralGroup(l:L I):PERMGRP I ==
  l:=removeDuplicates l
  #1 < 3 => error "in dihedralGroup: Minimum of 3 elements needed !"
  tmp := [[l.i, l.(#l-i+1) ] for i in 1..(#l quo 2)]
  llli2gp [ [ l ], tmp ]

dihedralGroup(n:PI):PERMGRP I ==
  n = 1 => symmetricGroup (2::PI)
  n = 2 => llli2gp [[[1,2],[[3,4]]]
  dihedralGroup li1n n

— PGE.dotabb —

"PGE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PGE"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
package PICOERCE PiCoercions

— PiCoercions.input —

)set break resume
)sys rm -f PiCoercions.output
)spool PiCoercions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PiCoercions
--R
--R PiCoercions(R: Join(OrderedSet,IntegralDomain)) is a package constructor
--R Abbreviation for PiCoercions is PICOERCE
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PICOERCE
--R
--R----------------------------------- Operations -----------------------------------
--R coerce : Pi -> Expression(R)
--R
--E 1

)spool
)lisp (bye)

— PiCoercions.help —

PiCoercions examples

Provides a coercion from the symbolic fractions in %pi with integer coefficients to any Expression type.

See Also:
c )show PiCoercions
PiCoercions (PICOERCE)

Exports:
coerce

— package PICOERCE PiCoercions —

)abbrev package PICOERCE PiCoercions
++ Author: Manuel Bronstein
++ Date Created: 21 Feb 1990
++ Date Last Updated: 21 Feb 1990
++ Description:
++ Provides a coercion from the symbolic fractions in %pi with
++ integer coefficients to any Expression type.

PiCoercions(R:Join(OrderedSet, IntegralDomain)):: with
coerce: Pi -> Expression R
++ coerce(f) returns f as an Expression(R).
== add
p2e: SparseUnivariatePolynomial Integer -> Expression R

coerce(x:Pi):Expression(R) ==
f := convert(x)$Fraction(SparseUnivariatePolynomial Integer)
p2e(numer f) / p2e(denom f)

p2e p ==
map((x1:Integer):Expression(R) +-> x1::Expression(R), p)
    $SparseUnivariatePolynomialFunctions2(Integer, Expression R)
    (pi())$Expression(R)
package PLOT1 PlotFunctions1

-- PlotFunctions1.input --

)set break resume
)sys rm -f PlotFunctions1.output
)spool PlotFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PlotFunctions1
--R
--R PlotFunctions1(S: ConvertibleTo(InputForm)) is a package constructor
--R Abbreviation for PlotFunctions1 is PLOT1
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PLOT1
--R
--R----------------------------------- Operations -----------------------------------
--R plotPolar : (S,Symbol) -> Plot
--R plot : (S,Symbol,Segment(DoubleFloat)) -> Plot
--R plot : (S,S,Symbol,Segment(DoubleFloat)) -> Plot
--R plotPolar : (S,Symbol,Segment(DoubleFloat)) -> Plot
--R
--E 1

)spool
)lisp (bye)

---

-- PlotFunctions1.help --

====================================================================
PlotFunctions1 examples
PlotFunctions1 provides facilities for plotting curves where functions SF \( \rightarrow \) SF are specified by giving an expression.

See Also:

\texttt{)}show PlotFunctions1

---

### PlotFunctions1 (PLOT1)

**Exports:**

- plotPolar
- plot

---

)abbrev package PLOT1 PlotFunctions1
++ Authors: R.T.M. Bronstein, C.J. Williamson
++ Date Created: Jan 1989
++ Date Last Updated: 4 Mar 1990
++ Description:
++ PlotFunctions1 provides facilities for plotting curves
++ where functions SF \( \rightarrow \) SF are specified by giving an expression.

PlotFunctions1(S:ConvertibleTo InputForm): with

- plot : (S, Symbol, Segment DoubleFloat) \( \rightarrow \) Plot
  ++ plot(fcn,x,seg) plots the graph of \( y = f(x) \) on a interval
- plot : (S, S, Symbol, Segment DoubleFloat) \( \rightarrow \) Plot
  ++ plot(f,g,t,seg) plots the graph of \( x = f(t), y = g(t) \)
  ++ as \( t \) ranges over an interval.
- plotPolar : (S, Symbol, Segment DoubleFloat) \( \rightarrow \) Plot
  ++ plotPolar(f,theta,seg) plots the graph of \( r = f(\theta) \) as
++ theta ranges over an interval
plotPolar : (S, Symbol) -> Plot
++ plotPolar(f,theta) plots the graph of \spad{r = f(theta)} as
++ theta ranges from 0 to 2 pi
== add
import MakeFloatCompiledFunction(S)

plot(f, x, xRange) == plot(makeFloatFunction(f, x), xRange)
plotPolar(f,theta) == plotPolar(makeFloatFunction(f,theta))
plot(f1, f2, t, tRange) ==
  plot(makeFloatFunction(f1, t), makeFloatFunction(f2, t), tRange)
plotPolar(f,theta,thetaRange) ==
  plotPolar(makeFloatFunction(f,theta),thetaRange)

package PLOTTOOL PlotTools

— PlotTools.input —

)set break resume
)sys rm -f PlotTools.output
)spool PlotTools.output
)set message test on
)set message auto off
)clear all

—S 1 of 1
)show PlotTools
—R
—R PlotTools is a package constructor
—R Abbreviation for PlotTools is PLOTTOOL
—R This constructor is not exposed in this frame.
—R Issue )edit bookvol10.4.pamphlet to see algebra source code for PLOTTOOL
—R
—R-------------------------------- Operations --------------------------------
--R calcRanges : List(List(Point(DoubleFloat))) -> List(Segment(DoubleFloat))
--R
--E 1

)spool
)lisp (bye)

---

— PlotTools.help —

====================================================================
PlotTools examples
====================================================================

This package exports plotting tools

See Also:
o )show PlotTools

---

PlotTools (PLOTTOOL)

Exports:
calcRanges

— package PLOTTOOL PlotTools —

)abbrev package PLOTTOOL PlotTools
++ Description:
++ This package exports plotting tools
PlotTools(): Exports == Implementation where
  L ==> List
  -- Pt ==> TwoDimensionalPoint
  SEG ==> Segment
  SF ==> DoubleFloat
  Pt ==> Point(SF)
  PLOT ==> Plot
  DROP ==> DrawOption
  S ==> String
  VIEW2D ==> TwoDimensionalViewport

Exports ==> with
  calcRanges: L L Pt -> L SEG SF
  ++ calcRanges(l) \undocumented

Implementation ==> add
  import GraphicsDefaults
  import PLOT
  import TwoDimensionalPlotClipping
  import DrawOptionFunctions0
  import ViewportPackage
  import POINT
  import PointPackage(SF)

  --%Local functions
  xRange0: L Pt -> SEG SF
  xRange: L L Pt -> SEG SF
  yRange0: L Pt -> SEG SF
  yRange: L L Pt -> SEG SF
  drawToScaleRanges: (SEG SF,SEG SF) -> L SEG SF

drawToScaleRanges(xVals,yVals) ==
  xDiff := (xHi := hi xVals) - (xLo := lo xVals)
  yDiff := (yHi := hi yVals) - (yLo := lo yVals)
  pad := abs(yDiff - xDiff)/2
  yDiff > xDiff => [segment(xLo - pad,xHi + pad),yVals]
  [xVals,segment(yLo - pad,yHi + pad)]

select : (L Pt,Pt -> SF,(SF,SF) -> SF) -> SF
select(l,f,g) ==
  m := f first l
  for p in rest l repeat m := g(m,f p)
  m

xRange0(list:L Pt) == select(list,xCoord,min) .. select(list,xCoord,max)
yRange0(list:L Pt) == select(list,yCoord,min) .. select(list,yCoord,max)

select2: (L L Pt,L Pt -> SF,(SF,SF) -> SF) -> SF
select2(l,f,g) ==
m := f first l
for p in rest l repeat m := g(m,f p)
m

xRange(list:L L Pt) ==
  select2(list,(u1:L(Pt)):SF +-> lo(xRange0(u1)),min) _
  .. select2(list,(v1:L(Pt)):SF +-> hi(xRange0(v1)),max)

yRange(list:L L Pt) ==
  select2(list,(u1:L(Pt)):SF +-> lo(yRange0(u1)),min) _
  .. select2(list,(v1:L(Pt)):SF +-> hi(yRange0(v1)),max)

--%Exported Functions
calcRanges(llp) ==
drawToScale() => drawToScaleRanges(xRange llp, yRange llp)
[xRange llp, yRange llp]

———

— PLOTTOOL.dotabb —

"PLOTTOOL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PLOTTOOL"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"PLOTTOOL" -> "FIELD"
"PLOTTOOL" -> "RADCAT"

———

PRJALGPK ProjectiveAlgebraicSetPackage

package PRJALGPK ProjectiveAlgebraicSetPackage

—— ProjectiveAlgebraicSetPackage.input ——

)set break resume
)sys rm -f ProjectiveAlgebraicSetPackage.output
)spool ProjectiveAlgebraicSetPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ProjectiveAlgebraicSetPackage
--R
CHAPTER 17. CHAPTER P

--R ProjectiveAlgebraicSetPackage(K: Field, symb: List(Symbol), PolyRing: PolynomialCategory(K,E,OrderedVariableList(symb)), E: DirectProductCategory(#(symb), NonNegativeInteger), ProjPt: ProjectiveSpaceCategory(K)) is a package constructor
--R Abbreviation for ProjectiveAlgebraicSetPackage is PRJALGPK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PRJALGPK
--R
--R----------------------------------- Operations -----------------------------------
--R algebraicSet : List(PolyRing) -> List(ProjPt)
--R rationalPoints : (PolyRing, PositiveInteger) -> List(ProjPt)
--R singularPoints : PolyRing -> List(ProjPt)
--R singularPointsWithRestriction : (PolyRing, List(PolyRing)) -> List(ProjPt)
--R
--E 1

)spool
)lisp (bye)

——

ProjectiveAlgebraicSetPackage.help —

====================================================================
ProjectiveAlgebraicSetPackage examples
====================================================================

The following is part of the PAFF package

See Also:
  o )show ProjectiveAlgebraicSetPackage

——

ProjectiveAlgebraicSetPackage (PRJALGPK)

PRJALGPK

AFALGGRO
Exports:
algebraicSet rationalPoints singularPoints singularPointsWithRestriction

— package PRJALGPK ProjectiveAlgebraicSetPackage —

)abbrev package PRJALGPK ProjectiveAlgebraicSetPackage
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
ProjectiveAlgebraicSetPackage(K,symb,PolyRing,E,ProjPt):_
Exports == Implementation where
K : Field
symb: List(Symbol)
OV ==> OrderedVariableList(symb)
E : DirectProductCategory(#symb,NonNegativeInteger)
PolyRing : PolynomialCategory(K,E,OV)
ProjPt : ProjectiveSpaceCategory(K)
PCS : LocalPowerSeriesCategory(K)

OF ==> OutputForm
PI ==> PositiveInteger
RFP ==> RootsFindingPackage
SUP ==> SparseUnivariatePolynomial
PPFC1 ==> PolynomialPackageForCurve(K,PolyRing,E,#symb,ProjPt)
SPWRES ==> AffineAlgebraicSetComputeWithResultant(K,symb,PolyRing,E,ProjPt)
SPWGRO ==> AffineAlgebraicSetComputeWithGroebnerBasis(K,symb,PolyRing,E,ProjPt)

Exports ==> with

  singularPointsWithRestriction: (PolyRing,List(PolyRing)) -> List(ProjPt)
  ++ return the singular points that anhilate

  singularPoints: PolyRing -> List(ProjPt)
  ++ singularPoints retourne les points singulier

  algebraicSet: List(PolyRing) -> List(ProjPt)
  ++ algebraicSet returns the algebraic set if finite (dimension 0).

  rationalPoints: (PolyRing,PI) -> List(ProjPt)
  ++ \texttt{ \textbackslash{}axiom\{rationalPoints(f,d)\}} returns all points on the curve \texttt{ \textbackslash{}axiom\{f\}}
  ++ in the extension of the ground field of degree \texttt{ \textbackslash{}axiom\{d\}}.
  ++ For \texttt{ \textbackslash{}axiom\{d > 1\}} this only works if \texttt{ \textbackslash{}axiom\{K\}} is a
  ++ \texttt{ \textbackslash{}axiom\{LocallyAlgebraicallyClosedField\}}

Implementation ==> add

import PPFC1
import PolyRing
import ProjPt

listVar: List(OV) := [index(i::PI)$OV for i in 1..#symb]
opolyToX10 : PolyRing -> SUP(K)

--fonctions de resolution de sys. alg. de dim 0
singuarPoints(crb)==
  F:=crb
  Fx:= differentiate(F,index(1)$OV)
  Fy:= differentiate(F,index(2)$OV)
  Fz:= differentiate(F,index(3)$OV)
  idealT: List PolyRing := [F,Fx,Fy,Fz]
  idealToX10: List SUP(K) := [polyToX10 pol for pol in idealT]
  recOfZerosX10: = distinguishedCommonRootsOf(idealToX10,1)$RFP(K)
  listOfExtDeg: List Integer := [recOfZerosX10.extDegree]
  degExt:=lcm listOfExtDeg
  zero?(degExt) =>
    error("------- Infinite number of points ------")
  one?(degExt) =>
    print("You need an extension of degree")::OF
    print(degExt::OF)
    error("--------------Have a nice day--------------")
  listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
  templ:= affineSingularPoints(crb)$SPWRES
  if templ case "failed" then
    print("failed with resultant")::OF)
    print("The singular points will be computed using grobner basis")::OF)
    templ := affineSingularPoints(crb)$SPWGR0
  templ case "Infinite" =>
    error("------- Infinite number of points ------")
  templ case Integer =>
    print("You need an extension of degree")::OF)
    print(tempL ::OF)
    error("--------------Have a nice day--------------")
  listPtsIdl2: List(ProjPt)
  if templ case List(ProjPt) then
    listPtsIdl2:= (tempL :: List(ProjPt))
  else
    error" From ProjectiveAlgebraicSetPackage: this should not happen"
    listPtsIdl := concat( listPtsId1 , listPtsIdl2)
  if pointInIdeal?(idealT,projectivePoint([1,0,0]))$PPFC1 then
    listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
  listPtsIdl

algebraicSet(idealT:List(PolyRing)) ==
  idealToX10: List SUP(K) := [polyToX10 pol for pol in idealT]
  recOfZerosX10: = distinguishedCommonRootsOf(idealToX10,1)$RFP(K)
  listOfExtDeg: List Integer := [recOfZerosX10.extDegree]
  degExt:=lcm listOfExtDeg
zero?(degExt) =>
  error("-------- Infinite number of points ------")
"one?(degExt) =>
  print(("You need an extension of degree")::OF)
  print(degExt::OF)
  error("---------- Have a nice day----------")
listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
tempL:= affineAlgSet( idealT )$SPWRES
if tempL case "failed" then
  print("failed with resultant"::OF)
  print("The finte alg. set will be computed using grobner basis"::OF)
  tempL := affineAlgSet( idealT )$SPWGRO
  tempL case "Infinite" =>
    error("-------- Infinite number of points ------")
  tempL case Integer =>
    print(("You need an extension of degree")::OF)
    print(tempL ::OF)
    error("---------- Have a nice day----------")
listPtsIdl2:List(ProjPt)
if tempL case List(ProjPt) then
  listPtsIdl2:= ( tempL :: List(ProjPt) )
else
  error" From ProjectiveAlgebraicSetPackage: this should not hapen"
listPtsIdl := concat( listPtsIdl , listPtsIdl2)
if pointInIdeal?(idealT,projectivePoint([1,0,0]))$PPFC1 then
  listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
listPtsIdl

if K has FiniteFieldCategory then
  rationalPoints(crv:PolyRing,extdegree:PI):List(ProjPt) ==
    --The code of this is almost the same as for algebraicSet
    --We could just construct the ideal and call algebraicSet
    --Should we do that? This might be a bit faster.
    listPtsIdl:List(ProjPt):= empty()
    x:= monomial(1,1)$SUP(K)
    if K has PseudoAlgebraicClosureOfFiniteFieldCategory then
      setTower!(1$K)$K
      q:= size()$K
      px:= x**(q**extdegree) - x
      crvX10:= polyToX10 crv
      recOfZerosX10:= distinguishedCommonRootsOf([crvX10,px],1$K)$RFP(K)
      listPtsIdl:= [projectivePoint([a,1,0]) for a in recOfZerosX10.zeros]
      --now we got all of the projective points where z = 0 and y ^= 0
      ratXY1 : List ProjPt:= affineRationalPoints( crv, extdegree )$SPWGRO
      listPtsIdl:= concat(ratXY1,listPtsIdl)
      if pointInIdeal?([crv],projectivePoint([1,0,0]))$PPFC1 then
        listPtsIdl:=cons(projectivePoint([1,0,0]),listPtsIdl)
    listPtsIdl
polyToX10(pol) == 
  zero?(pol) => 0 
  dd := degree pol 
  lc := leadingCoefficient pol 
  pp := parts dd 
  lp := last pp 
  ~zero?(lp) => polyToX10 reductum pol 
  e1 := pp.1 
  monomial(lc, e1)^sup(K) + polyToX10 reductum pol

singularPointsWithRestriction(F, lstPol) ==
  Fx := differentiate(F, index(1)$OV)
  Fy := differentiate(F, index(2)$OV)
  Fz := differentiate(F, index(3)$OV)
  idealSingulier := List(PolyRing) := concat([F, Fx, Fy, Fz], lstPol)
  algebraicSet(idealSingulier)

package PTFUNC2 PointFunctions2

  -- PointFunctions2.input

)set break resume
)sys rm -f PointFunctions2.output
)spool PointFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointFunctions2
--R
--R PointFunctions2(R1: Ring, R2: Ring) is a package constructor
--R Abbreviation for PointFunctions2 is PTFUNC2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PTFUNC2
--R
--R------------------------------------- Operations -------------------------------------
--R map : ((R1 -> R2),Point(R1)) -> Point(R2)
--R
--E 1

)spool
)lisp (bye)

——

--- PointFunctions2.help ---

====================================================================
PointFunctions2 examples
====================================================================

This package has no description

See Also:
 o )show PointFunctions2

——

PointFunctions2 (PTFUNC2)

Exports:
 map

— package PTFUNC2 PointFunctions2 —

)abbrev package PTFUNC2 PointFunctions2
++ Description:
++ This package has no description

PointFunctions2(R1: Ring, R2: Ring): Exports == Implementation where

Exports == with
  map : ((R1 -> R2), Point(R1)) -> Point(R2)
  ++ map(f, p) undocumented

Implementation ==> add
  import Point(R1)
  import Point(R2)

  map(mapping, p) ==
    point([mapping p.(i::PositiveInteger) for i in minIndex(p)..maxIndex(p)])$Point(R2)

---

--- PTFUNC2.dotabb ---

"PTFUNC2", [color="#FF4488", href="bookvol10.4.pdf#nameddest=PTFUNC2"]
"PID", [color="#4488FF", href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP", [color="#4488FF", href="bookvol10.2.pdf#nameddest=OAGROUP"]
"PTFUNC2" -> "PID"
"PTFUNC2" -> "OAGROUP"

---

package PTPACK PointPackage

--- PointPackage.input ---

)set break resume
)sys rm -f PointPackage.output
)spool PointPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointPackage
--R
--R PointPackage(R: Ring) is a package constructor
--R Abbreviation for PointPackage is PTPACK
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PTPACK
--R
--R---------------------------------------------------------- Operations --------------------------------
--R color : Point(R) -> R     hue : Point(R) -> R
--R phiCoord : Point(R) -> R   rCoord : Point(R) -> R
--R shade : Point(R) -> R     thetaCoord : Point(R) -> R
--R xCoord : Point(R) -> R    yCoord : Point(R) -> R
--R zCoord : Point(R) -> R
--R
--E 1

)spool
)lisp (bye)

——

— PointPackage.help —

====================================================================
PointPackage examples
====================================================================

This package has no description

See Also:
o )show PointPackage

——

PointPackage (PTPACK)
Exports:
color hue phiCoord rCoord shade
thetaCoord xCoord yCoord zCoord

— package PTPACK PointPackage —

)abbrev package PTPACK PointPackage
++ Description:
++ This package has no description

PointPackage(R:Ring):Exports == Implementation where

POINT ==> Point(R)
I ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
L ==> List
B ==> Boolean

Exports == with
xCoord : POINT -> R 
  ++ xCoord(pt) returns the first element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used. This function is defined for the
  ++ convenience of the user dealing with a Cartesian
  ++ coordinate system.
yCoord : POINT -> R
  ++ yCoord(pt) returns the second element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used. This function is defined for the
  ++ convenience of the user dealing with a Cartesian
  ++ coordinate system.
zCoord : POINT -> R
  ++ zCoord(pt) returns the third element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used. This function is defined for the
  ++ convenience of the user dealing with a Cartesian
  ++ or a cylindrical coordinate system.
rCoord : POINT -> R
  ++ rCoord(pt) returns the first element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used. This function is defined for the
  ++ convenience of the user dealing with a spherical
  ++ or a cylindrical coordinate system.
thetaCoord : POINT -> R
  ++ thetaCoord(pt) returns the second element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used. This function is defined for the
  ++ convenience of the user dealing with a spherical
  ++ or a cylindrical coordinate system.
phiCoord : POINT -> R
  ++ phiCoord(pt) returns the third element of the point, pt,
  ++ although no assumptions are made as to the coordinate
  ++ system being used. This function is defined for the
  ++ convenience of the user dealing with a spherical
  ++ coordinate system.

color : POINT -> R
  ++ color(pt) returns the fourth element of the point, pt,
  ++ although no assumptions are made with regards as to
  ++ how the components of higher dimensional points are
  ++ interpreted. This function is defined for the
  ++ convenience of the user using specifically, color
  ++ to express a fourth dimension.

hue : POINT -> R
  ++ hue(pt) returns the third element of the two dimensional point, pt,
  ++ although no assumptions are made with regards as to how the
  ++ components of higher dimensional points are interpreted. This
  ++ function is defined for the convenience of the user using
  ++ specifically, hue to express a third dimension.

shade : POINT -> R
  ++ shade(pt) returns the fourth element of the two dimensional
  ++ point, pt, although no assumptions are made with regards as to
  ++ how the components of higher dimensional points are interpreted.
  ++ This function is defined for the convenience of the user using
  ++ specifically, shade to express a fourth dimension.

-- 2D and 3D extraction of data
Implementation ==> add

xCoord p == elt(p,1)
yCoord p == elt(p,2)
zCoord p == elt(p,3)
rCoord p == elt(p,1)
thetaCoord p == elt(p,2)
phiCoord p == elt(p,3)
color p ==
  #p > 3 => p.4
  p.3
hue p == elt(p,3)
  -- 4D points in 2D using extra dimensions for palette information
shade p == elt(p,4)
  -- 4D points in 2D using extra dimensions for palette information

---

-- PTPACK.dotabb --

"PTPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PTPACK"]
package PFO PointsOfFiniteOrder

— PointsOfFiniteOrder.input —

)set break resume
)sys rm -f PointsOfFiniteOrder.output
)spool PointsOfFiniteOrder.output
)set message test on
)set message auto off
)clear all

<table>
<thead>
<tr>
<th>PointsOfFiniteOrder examples</th>
</tr>
</thead>
</table>

This package provides function for testing whether a divisor on a curve is a torsion divisor.
See Also:
  o )show PointsOfFiniteOrder

PointsOfFiniteOrder (PFO)

Exports:
  order torsion? torsionIfCan

— package PFO PointsOfFiniteOrder —

)abbrev package PFO PointsOfFiniteOrder
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 22 July 1998
++ Description:
  ++ This package provides function for testing whether a divisor on a
  ++ curve is a torsion divisor.

PointsOfFiniteOrder(R0, F, UP, UPUP, R): Exports == Implementation where
  R0 : Join(OrderedSet, IntegralDomain, RetractableTo Integer)
  F : FunctionSpace R0
  UP : UnivariatePolynomialCategory F
  UPUP : UnivariatePolynomialCategory Fraction UP
  R : FunctionFieldCategory(F, UP, UPUP)

PI ==> PositiveInteger
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Integer
UPF ==> SparseUnivariatePolynomial F
UPQ ==> SparseUnivariatePolynomial Q
QF ==> Fraction UP
UPUPQ ==> SparseUnivariatePolynomial Fraction UPQ
UP2 ==> SparseUnivariatePolynomial UPQ
UP3 ==> SparseUnivariatePolynomial UP2
FD ==> FiniteDivisor(F, UP, UPUP, R)
K ==> Kernel F
REC ==> Record(ncurve:UP3, disc:Z, dfpoly:UPQ)
RC0 ==> Record(ncurve:UPUPQ, disc:Z)
ID ==> FractionalIdeal(UP, QF, UPUP, R)
SMP ==> SparseMultivariatePolynomial(R0,K)
ALGOP ==> "%alg"

Exports ==> with
order : FD -> Union(N, "failed")
++ order(f) \undocumented
 torsion? : FD -> Boolean
++ torsion?(f) \undocumented
torsionIfCan : FD -> Union(Record(order:N, function:R), "failed")
++ torsionIfCan(f) \undocumented

Implementation ==> add
import IntegerPrimesPackage(Z)
import PointsOfFiniteOrderTools(UPQ, UPUPQ)
import UnivariatePolynomialCommonDenominator(Z, Q, UPQ)

cmult: List SMP -> SMP
raise : (UPQ, K) -> F
raise2 : (UP2, K) -> UP
qmod : F -> Q
fmod : UPF -> UPQ
rmod : UP -> UPQ
pmod : UPUP -> UPUPQ
kqmod : (F, K) -> UPQ
krmod : (UP, K) -> UP2
kpmod : (UPUP, K) -> UP3
selectIntegers: K -> REC
selIntegers: () -> RC0
possibleOrder : FD -> N
ratcurve : (FD, RC0) -> N
algcurve : (FD, REC, K) -> N
kbad3Num : (UP3, UPQ) -> Z
kbadBadNum : (UP2, UPQ) -> Z
kgetGoodPrime : (REC, UPQ, UP3, UP2,UP2) -> Record(prime:PI,poly:UPQ)
goodRed : (REC, UPQ, UP3, UP2, UP2, PI) -> Union(UPQ, "failed")
good? : (UPQ, UP3, UP2, UP2, PI, UPQ) -> Boolean
klist : UP -> List K
aklist : R -> List K
alglist : FD -> List K
notIrr? : UPQ -> Boolean
rat : (UPUP, FD, PI) -> N
toQ1 : (UP2, UPQ) -> UP
Q2F : Q -> F
Q2UPUP : UPUPQ -> UPUP

q := FunctionSpaceReduce(R0, F)

torsion? d == order(d) case N
Q2F x == numer(x)::F / denom(x)::F
qmod x == bringDown(x)$q
kqmod(x,k) == bringDown(x, k)$q
fmod p == map(qmod, p)$SparseUnivariatePolynomialFunctions2(F, Q)
Q2UPUP p == map(Q2F, p)$MultipleMap(Q, UPQ, UPUPQ, F, UP, UPUP)

UPQ2F(p:UPQ, k:K):F ==
    map(Q2F, p)$UnivariatePolynomialCategoryFunctions2(Q, UPQ, F, UP) (k::F)

UP22UP(p:UP2, k:K):UP ==
    map((p1:UPQ):F +-> UPQ2F(p1, k)::QF,p)$UnivariatePolynomialCategoryFunctions2(UP2, UPQ, UPUP)

UP32UPUP(p:UP3, k:K):UPUP ==
    map((p1:UP2):QF +-> UP22UP(p1,k)::QF,p)$UnivariatePolynomialCategoryFunctions2(UP2, UPUP)

if R0 has GcdDomain then
    cmult(l:List SMP):SMP == lcm l
else
    cmult(l:List SMP):SMP == */l

doubleDisc(f:UP3):Z ==
    d := discriminant f
    g := gcd(d, differentiate d)
    d := (d exquo g)::UP2
    zero?(e := discriminant d) => 0
    gcd [retract(c)@Z for c in coefficients e]

commonDen(p:UP):SMP ==
    l1:List F := coefficients p
    l2:List SMP := [denom c for c in l1]
cmult l2

polyred(f:UPUP):UPUP ==
    cmult([commonDen(retract(c)@UP) for c in coefficients f]):F::UP::QF * f

aklist f ==
    (r := retractIfCan(f)@Union(QF, "failed")) case "failed" =>
    "setUnion"/[klist(retract(c)@UP) for c in coefficients lift f]
    klist(retract(r::QF)@UP)

alglist d ==
    n := numer(i := ideal d)
    select_!((k1:K):Boolean +-> has?(operator k1, ALGOP),
        setUnion(klist denom i,
            "setUnion"/[aklist qelt(n,i) for i in minIndex n .. maxIndex n]))

krmod(p,k) ==
    map(z1 +-> kqmod(z1, k),
        p)$UnivariatePolynomialCategoryFunctions2(F, UP, UPQ, UP2)
    rmod p ==
    map(qmod, p)$UnivariatePolynomialCategoryFunctions2(F, UP, Q, UPQ)
    raise(p, k) ==
    (map(Q2F, p)$SparseUnivariatePolynomialFunctions2(Q, F)) (k::F)
    raise2(p, k) ==
    map(z1 +-> raise(z1, k),
        p)$UnivariatePolynomialCategoryFunctions2(UPQ, UP2, F, UP)

algcurve(d, rc, k) ==
    mn := minIndex(n := numer(i := minimize ideal d))
    h := kmod(lift(hh := n(mn + 1)), k)
    b2 := primitivePart
        raise2(b := krmod(retract(retract(n.mn)@QF)@UP, k), k)
    s := kqmod(resultant(primitivePart separate(raise2(krmod( 
        retract(norm hh)@UP, k), k), b2).primePart, b2), k)
    pr := kgetGoodPrime(rc, s, h, b, dd := krmod(denom i, k))
    p := pr.prime
    pp := UP32UPUP(rc.ncurve, k)
    mm := pr.poly
    gf := InnerPrimeField p
    m := map((z1:Q):gf +-> retract(z1)@Z :: gf,mm)
        $SparseUnivariatePolynomialFunctions2(Q, gf)
    -- one? degree m =>
        (degree m = 1) =>
            alpha := - coefficient(m, 0) / leadingCoefficient m
            order(d, pp,
                (z1:F):gf +-> (map((q1:Q):gf +-> numer(q1)::gf / denom(q1)::gf,
                kqmod(z1,k))$SparseUnivariatePolynomialFunctions2(Q,gf))(alpha)
-- d1 := toQ1(dd, mm)
-- rat(pp, divisor ideal([(toQ1(b, mm) / d1)::QF::R,
   -- inv(d1::QF) * toQ2(h, mm)])$ID, p)
order(d, pp,
   (z1:F):sae +-> reduce(map((q1:Q):gf +-> numer(q1)::gf / denom(q1)::gf,
      kqmod(z1, k))$SparseUnivariatePolynomialFunctions2(Q, gf))$sae
)$ReducedDivisor(F, UP, UPUP, R, sae)

-- returns the potential order of d, 0 if d is of infinite order
ratcurve(d, rc) ==
   mn := minIndex(nm := numer(i := minimize ideal d))
   h := pmod lift(hh := nm(mn + 1))
   b := rmod(retract(retract(nm.mn)@QF)@UP)
   s := separate(rmod(retract(norm hh)@UP), b).primePart
   bd := badNum rmod denom i
   r := resultant(s, b)
   bad := lcm [rc.disc, numer r, denom r, bd.den*bd.gcdnum, badNum h]$List(Z)
   pp := Q2UPUP(rc.ncurve)
   n := rat(pp, d, p := getGoodPrime bad)
   -- if n > 1 then it is cheaper to compute the order modulo a second prime,
   -- since computing n * d could be very expensive
   -- one? n => n
   n = 1 => n
   m := rat(pp, d, getGoodPrime(p * bad))
   n = m => n
   0

-- returns the order of d mod p
rat(pp, d, p) ==
gf := InnerPrimeField p
order(d, pp,
   (q1:F):gf +-> (qq := qmod q1; numer(qq)::gf / denom(qq)::gf)
)$ReducedDivisor(F, UP, UPUP, R, gf)

-- returns the potential order of d, 0 if d is of infinite order
possibleOrder d ==
   -- zero?(genus()) or one?(#(numer ideal d)) => 1
   zero?(genus()) or (#(numer ideal d) = 1) => 1
   empty?(la := alglist d) => error "PFO::possibleOrder: more than 1 algebraic constant"
   algcurve(d, selectIntegers first la, first la)
   selIntegers():RC0 ==
      f := definingPolynomial()$R
      while zero?(d := doubleDisc(r := polyred pmod f)) repeat newReduce()$q
      [r, d]
selectIntegers(k:K) : REC ==
g := polyred(f := definingPolynomial()$R)
p := minPoly k
while zero?(d := doubleDisc(r := kpmod(g, k))) or (notIrr? fmod p)
    repeat newReduc()$q
[r, d, splitDenominator(fmod p).num]

toQ1(p, d) ==
    map((p1:UPQ):F +-> Q2F(retract(p1 rem d)@Q), p)$UnivariatePolynomialCategoryFunctions2(UPQ, UP2, F, UP)

toQ2(p, d) ==
    reduce map((p1:UP2):QF +-> toQ1(p1, d)::QF, p)$UnivariatePolynomialCategoryFunctions2(UP2, UP3, QF, UPUP)

kpmod(p, k) ==
    map((p1:QF):UP2 +-> krmod(retract(p1)@UP, k), p)$UnivariatePolynomialCategoryFunctions2(QF, UPUP, UP2, UP3)

order d ==
    zero?(n := possibleOrder(d := reduce d)) => "failed"
    principal? reduce(n::Z * d) => n
    "failed"

kgetGoodPrime(rec, res, h, b, d) ==
p:PI := 3
while (u := goodRed(rec, res, h, b, d, p)) case "failed" repeat
    p := nextPrime(p::Z)::PI
[p, u::UPQ]

goodRed(rec, res, h, b, d, p) ==
    zero?(rec.disc rem p) => "failed"
    gf := InnerPrimeField p
    l := [f.factor for f in factors
        factor(map((z1:Q):gf +-> retract(z1)@Z :: gf, rec.dfpoly)$SparseUnivariatePolynomialCategoryFunctions2(Q, gf))$DistinctDegreeFactorize(gf, SparseUnivariatePolynomial gf) | one?(f.exponent)]
    SparseUnivariatePolynomial gf) | (f.exponent = 1)]
    empty? l => "failed"
    mdg := first l
    for ff in rest l repeat
        if degree(ff) < degree(mdg) then mdg := ff
    md := map((z1:gf):Q +-> convert(z1)@Z :: Q, mdg)$SparseUnivariatePolynomialCategoryFunctions2(gf, Q)
    good?(res, h, b, d, p, md) => md
    "failed"
good?(res, h, b, d, p, m) ==
    bd := badNum(res rem m)
not (zero? (bd.den rem p) or zero? (bd.gcdnum rem p) or 
zero? (kbadBadNum(b, m) rem p) or zero? (kbadBadNum(d, m) rem p) 
or zero? (kbad3Num(h, m) rem p))

package PFOQ PointsOfFiniteOrderRational

not (zero? (bd.den rem p) or zero? (bd.gcdnum rem p) or 
zero? (kbadBadNum(b, m) rem p) or zero? (kbadBadNum(d, m) rem p) 
or zero? (kbad3Num(h, m) rem p))

---

package PFOQ PointsOfFiniteOrderRational

--- PointsOfFiniteOrderRational.input ---

)set break resume
)sys rm -f PointsOfFiniteOrderRational.output
)spool PointsOfFiniteOrderRational.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PointsOfFiniteOrderRational
--R
--R PointsOfFiniteOrderRational(UP: UnivariatePolynomialCategory(Fraction(Integer)),UPUP: UnivariatePolynomialCategory)
--R Abbreviation for PointsOfFiniteOrderRational is PFOQ
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFOQ
--R
--R-------------------------------- Operations --------------------------------
--R order : FiniteDivisor(Fraction(Integer),UP,UPUP,R) -> Union(NonNegativeInteger,"failed")
--R torsion? : FiniteDivisor(Fraction(Integer),UP,UPUP,R) -> Boolean
--R torsionIfCan : FiniteDivisor(Fraction(Integer),UP,UPUP,R) -> Union(Record(order: NonNegativeInteger,func...
--R
--E 1

)spool
)lisp (bye)
This package provides function for testing whether a divisor on a curve is a torsion divisor.

See Also:
- )show PointsOfFiniteOrderRational

Exports:
order torsion? torsionIfCan

package PFOQ PointsOfFiniteOrderRational

PointsOfFiniteOrderRational(UP, UPUP, R): Exports == Implementation where

UP : UnivariatePolynomialCategory Fraction Integer
UPUP : UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(Fraction Integer, UP, UPUP)

PI ==> PositiveInteger
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Integer
FD ==> FiniteDivisor(Q, UP, UPUP, R)

Exports ==> with
order : FD -> Union(N, "failed")
++ order(f) undocumented
 torsion? : FD -> Boolean
++ torsion?(f) undocumented
torsionIfCan: FD -> Union(Record(order:N, function:R), "failed")
++ torsionIfCan(f) undocumented

Implementation ==> add
import PointsOfFiniteOrderTools(UP, UPUP)

possibleOrder: FD -> N
ratcurve : (FD, UPUP, Z) -> N
rat : (UPUP, FD, PI) -> N

torsion? d == order(d) case N
-- returns the potential order of d, 0 if d is of infinite order
ratcurve(d, modulus, disc) ==
  mn := minIndex(nm := numer(i := ideal d))
  h := lift(hh := nm(mn + 1))
  s := separate(retract(norm hh)@UP,
      b := retract(retract(nm.mn)@Fraction(UP))@UP).primePart
  bd := badNum denom i
  r := resultant(s, b)
  bad := lcm [disc, numer r, denom r, bd.den * bd.gcdnum, badNum h]$List(Z)
  n := rat(modulus, d, p := getGoodPrime bad)
-- if n > 1 then it is cheaper to compute the order modulo a second prime,
-- since computing n * d could be very expensive
-- one? n => n
  (n = 1) => n
  m := rat(modulus, d, getGoodPrime(p * bad))
  n = m => n
  0

rat(pp, d, p) ==
gf := InnerPrimeField p
order(d, pp,
  (z1:Q):gf +->
    numer(z1)::gf / denom(z1)::gf)$ReducedDivisor(Q, UP, UPUP, R, gf)
CHAPTER 17. CHAPTER P

-- returns the potential order of d, 0 if d is of infinite order
possibleOrder d ==
  zero?(genus()) or one?(#(numer ideal d)) => 1
  zero?(genus()) or (#(numer ideal d) = 1) => 1
  r := polyred definingPolynomial()$R
  ratcurve(d, r, doubleDisc r)

order d ==
  zero?(n := possibleOrder(d := reduce d)) => "failed"
  principal? reduce(n::Z * d) => n
  "failed"

torsionIfCan d ==
  zero?(n := possibleOrder(d := reduce d)) => "failed"
  (g := generator reduce(n::Z * d)) case "failed" => "failed"
  [n, g::R]

package PFOTOOLS PointsOfFiniteOrderTools

-- PointsOfFiniteOrderTools.input --

)set break resume
)sys rm -f PointsOfFiniteOrderTools.output
)spool PointsOfFiniteOrderTools.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PointsOfFiniteOrderTools
--R
--R PointsOfFiniteOrderTools(UP: UnivariatePolynomialCategory(Fraction(Integer)),UPUP: UnivariatePolynomialCategory(Fraction(Fraction(Integer)))) is a package constructor
--R Abbreviation for PointsOfFiniteOrderTools is PFOTOOLS
--R This constructor is not exposed in this frame.
PointsOfFiniteOrderTools (PFOTOOLS)

Exports:
badNum  doubleDisc  getGoodPrime  mix  polyred
package PFOTOOLS PointsOfFiniteOrderTools

)abbrev package PFOTOOLS PointsOfFiniteOrderTools
++ Author: Manuel Bronstein
++ Date Created: 25 Aug 1988
++ Date Last Updated: 11 Jul 1990
++ Description:
++ Utilities for PFOQ and PFO

PointsOfFiniteOrderTools(UP, UPUP): Exports == Implementation where

UP : UnivariatePolynomialCategory Fraction Integer
UPUP : UnivariatePolynomialCategory Fraction UP

PI ==> PositiveInteger
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Integer

Exports ==> with

goodPrime : Z -> PI
  ++ goodPrime n returns the smallest prime not dividing n

badNum : UP -> Record(den:Z, gcdnum:Z)
  ++ badNum(p) undocumented

badNum : UPUP -> Z
  ++ badNum(u) undocumented

mix : List Record(den:Z, gcdnum:Z) -> Z
  ++ mix(l) undocumented

doubleDisc : UPUP -> Z
  ++ doubleDisc(u) undocumented

polyred : UPUP -> UPUP
  ++ polyred(u) undocumented

Implementation ==> add

import IntegerPrimesPackage(Z)
import UnivariatePolynomialCommonDenominator(Z, Q, UP)

mix l == lcm(lcm [p.den for p in l], gcd [p.gcdnum for p in l])
badNum(p:UPUP) == mix [badNum(retract(c)@UP) for c in coefficients p]
polyred r ==
  lcm [commonDenominator(retract(c)@UP) for c in coefficients r] * r

badNum(p:UP) ==
  cd := splitDenominator p
  [cd.den, gcd [retract(c)@Z for c in coefficients(cd.num)]]

goodPrime n ==
  p:PI := 3
  while zero?(n rem p) repeat


```
p := nextPrime(p::Z)::PI

doubleDisc r ==
d := retract(discriminant r)@UP
    retract(discriminant((d exquo gcd(d, differentiate d)):UP))@Z

package PLPKCRV PolynomialPackageForCurve

| PolynomialPackageForCurve.input |

)set break resume
)sys rm -f PolynomialPackageForCurve.output
)spool PolynomialPackageForCurve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialPackageForCurve

--R
--R PolynomialPackageForCurve(K: Field,PolyRing: FiniteAbelianMonoidRing(K,E),E: DirectProductCategory(dim,NonNegativeInteger),dim: NonNegativeInteger,ProjPt: ProjectiveSpaceCategory(K)) is a package constructor
--R Abbreviation for PolynomialPackageForCurve is PLPKCRV
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PLPKCRV
--R
--R-------------------------------------------------------------- Operations -----------------------------------
--R eval : (PolyRing,ProjPt) -> K
--R minimalForm : (PolyRing,ProjPt) -> PolyRing
--R minimalForm : (PolyRing,ProjPt,Integer) -> PolyRing
--R multiplicity : (PolyRing,ProjPt) -> NonNegativeInteger
--R multiplicity : (PolyRing,ProjPt,Integer) -> NonNegativeInteger
--R pointInIdeal? : (List(PolyRing),ProjPt) -> Boolean
--R translateToOrigin : (PolyRing,ProjPt,Integer) -> PolyRing

```

---

"PFOTOOLS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFOTOOLS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFOTOOLS" -> "PFECAT"

---
PolynomialPackageForCurve (PLPKCRV)

Exports:
   eval minimalForm multiplicity pointInIdeal? translateToOrigin

--- package PLPKCRV PolynomialPackageForCurve ---

)abbrev package PLPKCRV PolynomialPackageForCurve
++ Author: Gaetan Hache
++ Date Created: 17 nov 1992
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ The following is part of the PAFF package
PolynomialPackageForCurve(K,PolyRing,E,dim,ProjPt):Exp == Impl where
   K:Field
   dim:NonNegativeInteger
   E : DirectProductCategory(dim,NonNegativeInteger)
   PolyRing: FiniteAbelianMonoidRing(K,E) -- PolynomialCategory(K,E,OV)
   ProjPt : ProjectiveSpaceCategory(K)

PackPoly ==> PackageForPoly(K,PolyRing,E,dim)

Exp ==> with
   pointInIdeal?: (List(PolyRing),ProjPt) -> Boolean
      ++ pointInIdeal? test if the given point is in the algebraic set
      ++ defined by the given list of polynomials.

   eval: (PolyRing,ProjPt) -> K
      ++ eval returns the value at given point.

   translateToOrigin: (PolyRing,ProjPt,Integer) -> PolyRing
      ++ translateToOrigin translate the polynomial from the given point
      ++ to the origin

   translateToOrigin: (PolyRing,ProjPt) -> PolyRing
      ++ translateToOrigin translate the polynomial from the given point
      ++ to the origin

   minimalForm: (PolyRing,ProjPt) -> PolyRing
      ++ minimalForm returns the minimal form after translation to
      ++ the origin.

   minimalForm: (PolyRing,ProjPt,Integer) -> PolyRing
      ++ minimalForm returns the minimal form after translation to
      ++ the origin.

   multiplicity: (PolyRing,ProjPt) -> NonNegativeInteger
      ++ multiplicity returns the multiplicity of the polynomial at
      ++ given point.

   multiplicity: (PolyRing,ProjPt,Integer) -> NonNegativeInteger
      ++ multiplicity returns the multiplicity of the polynomial at
      ++ given point.

Impl ==> add
   import PolyRing
   import ProjPt
   import PackPoly
translateToOrigin(pol, pt, nV) ==
    zero?(pt.nV) => error "Impossible de translater"
pt := homogenize(pt, nV)
lpt := list(pt)$$ProjPt
translate(pol, lpt, nV)

pointInIdeal?(lstPol, pt) ==
    temp:Boolean := true()$Boolean
    for pol in lstPol repeat
        temp:=(zero?(eval(pol, pt)) and temp)
    temp

eval(f, pt) ==
    zero? f => 0
lpt := list(K) := list pt
dd := entries degree f
lc := leadingCoefficient f
ee := reduce("*", [p**e for p in lpt for e in dd | zero?(e)], 1$K)
lc * ee + eval(reductum f, pt)

translateToOrigin(pol, pt) ==
    translateToOrigin(pol, pt, lastNonNull(pt))

multiplicity(crb, pt) ==
    degreeOfMinimalForm(translateToOrigin(crb, pt))

multiplicity(crb, pt, nV) ==
    degreeOfMinimalForm(translateToOrigin(crb, pt, nV))

minimalForm(crb, pt) ==
    minimalForm(translateToOrigin(crb, pt))

minimalForm(crb, pt, nV) ==
    minimalForm(translateToOrigin(crb, pt, nV))
package POLTOPOL PolToPol

--- PolToPol.input ---

)set break resume
)sys rm -f PolToPol.output
)spool PolToPol.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolToPol
--R
--R PolToPol(lv: List(Symbol),R: Ring) is a package constructor
--R Abbreviation for PolToPol is POLTOPOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for POLTOPOL
--R
--R------------------------------------------ Operations ------------------------------------------
--R dmpToHdmp : DistributedMultivariatePolynomial(lv,R) -> HomogeneousDistributedMultivariatePolynomial(lv,R)
--R dmpToP : DistributedMultivariatePolynomial(lv,R) -> Polynomial(R)
--R hdmpToDmp : HomogeneousDistributedMultivariatePolynomial(lv,R) -> DistributedMultivariatePolynomial(lv,R)
--R hdmpToP : HomogeneousDistributedMultivariatePolynomial(lv,R) -> Polynomial(R)
--R pToDmp : Polynomial(R) -> DistributedMultivariatePolynomial(lv,R)
--R pToHdmp : Polynomial(R) -> HomogeneousDistributedMultivariatePolynomial(lv,R)
--R
--E 1

)spool
)lisp (bye)

---

--- PolToPol.help ---

====================================================================
PolToPol examples
====================================================================

Package with the conversion functions among different kind of polynomials

See Also:
  o )show PolToPol

---
POLTOPOL (POLTOPOL)

Exports:
dmpToHdmp dmpToP hdmpToDmp hdmpToP pToDmp pToHdmp

— package POLTOPOL PolToPol —

)abbrev package POLTOPOL PolToPol
++ Author : P.Gianni, Summer '88
++ Description:
++ Package with the conversion functions among different kind of polynomials

PolToPol(lv,R) : C == T

where
  R : Ring
  lv : List Symbol
  NNI ==> NonNegativeInteger
  Ov ==> OrderedVariableList(lv)
  IES ==> IndexedExponents Symbol

  DP ==> DirectProduct(#lv,NonNegativeInteger)
  DPoly ==> DistributedMultivariatePolynomial(lv,R)

  HDP ==> HomogeneousDirectProduct(#lv,NonNegativeInteger)
  HDPoly ==> HomogeneousDistributedMultivariatePolynomial(lv,R)
  P ==> Polynomial R
  VV ==> Vector NNI
  MPC3 ==> MPolyCatFunctions3

C == with
  dmpToHdmp : DPoly -> HDPoly
    ++ dmpToHdmp(p) converts p from a \spadtype{DMP} to a \spadtype{HDMP}.
  hdmpToDmp : HDPoly -> DPoly
    ++ hdmpToDmp(p) converts p from a \spadtype{HDMP} to a \spadtype{DMP}.
  pToHdmp : P -> HDPoly
    ++ pToHdmp(p) converts p from a \spadtype{POLY} to a \spadtype{HDMP}.
hdmpToP : HDPoly -> P
++ hdmpToP(p) converts p from a \spadtype{HDPoly} to a \spadtype{POLY}.
dmpToP : DPoly -> P
++ dmpToP(p) converts p from a \spadtype{DMP} to a \spadtype{POLY}.
pToDmp : P -> DPoly
++ pToDmp(p) converts p from a \spadtype{POLY} to a \spadtype{DMP}.

T == add

variable1(x:Symbol):Ov == variable(x)::Ov

-- transform a P in a HDPoly --
ptoHdmp(pol:P) : HDPoly ==
  map(variable1,pol)$MPC3(Symbol,Ov,IES,HDP,R,P,HDPoly)

-- transform an HDPoly in a P --
hdmpToP(hdpol:HDPoly) : P ==
  map(convert,hdpol)$MPC3(Ov,Symbol,HDP,IES,R,HDPoly,P)

-- transform an DPoly in a P --
dmpToP(dpol:DPoly) : P ==
  map(convert,dpol)$MPC3(Ov,Symbol,DP,IES,R,DPoly,P)

-- transform a P in a DPoly --
pToDmp(pol:P) : DPoly ==
  map(variable1,pol)$MPC3(Symbol,Ov,IES,DP,R,P,DPoly)

-- transform a DPoly in a HDPoly --
dmpToHdmp(dpol:DPoly) : HDPoly ==
  dpol=0 => 0$HDPoly
  monomial(leadingCoefficient dpol,
    directProduct(degree(dpol)::VV)$HDP)$HDPoly+
    dmpToHdmp(reductum dpol)

-- transform a HDPoly in a DPoly --
hdmpToDmp(hdpol:HDPoly) : DPoly ==
  hdpol=0 => 0$DPoly
  dd:DP:= directProduct((degree hdpol)::VV)$DP
  monomial(leadingCoefficient hdpol,dd)$DPoly+
    hdmpToDmp(reductum hdpol)
package PGROEB PolyGroebner

--- PolyGroebner.input ---

)set break resume
)sys rm -f PolyGroebner.output
)spool PolyGroebner.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolyGroebner
--R
--R PolyGroebner(F: GcdDomain) is a package constructor
--R Abbreviation for PolyGroebner is PGROEB
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PGROEB
--R
--R-------------------------------- Operations --------------------------------
--R lexGroebner : (List(Polynomial(F)),List(Symbol)) -> List(Polynomial(F))
--R totalGroebner : (List(Polynomial(F)),List(Symbol)) -> List(Polynomial(F))
--R
--E 1

)spool
)lisp (bye)

---

--- PolyGroebner.help ---

====================================================================
PolyGroebner examples
====================================================================

This package is an interface package to the groebner basis package which allows you to compute groebner bases for polynomials in either lexicographic ordering or total degree ordering refined by reverse lex. The input is the ordinary polynomial type which is internally converted to a type with the required ordering. The resulting groebner basis is converted back to ordinary polynomials. The ordering among the variables is controlled by an explicit list of variables which is passed as a second argument. The coefficient
domain is allowed to be any gcd domain, but the groebner basis is computed as if the polynomials were over a field.

See Also:
  o )show PolyGroebner

---

PolyGroebner (PGROEB)

Exports:

  lexGroebner  totalGroebner

— package PGROEB PolyGroebner —

)abbrev package PGROEB PolyGroebner
++ Author: P. Gianni
++ Date Created: Summer 1988
++ Description:
++ Groebner functions for P F
++ This package is an interface package to the groebner basis
++ package which allows you to compute groebner bases for polynomials
++ in either lexicographic ordering or total degree ordering refined
++ by reverse lex. The input is the ordinary polynomial type which
++ is internally converted to a type with the required ordering.
++ The resulting groebner basis is converted back to ordinary polynomials.
++ The ordering among the variables is controlled by an explicit list
++ of variables which is passed as a second argument. The coefficient
++ domain is allowed to be any gcd domain, but the groebner basis is
++ computed as if the polynomials were over a field.

PolyGroebner(F) : C == T
where
F : GcdDomain
NNI ==> NonNegativeInteger
P ==> Polynomial F
L ==> List
E ==> Symbol

C == with
  lexGroebner : (L P,L E) -> L P
  ++ lexGroebner(lp,lv) computes Groebner basis
  ++ for the list of polynomials lp in lexicographic order.
  ++ The variables are ordered by their position in the list lv.
  totalGroebner : (L P, L E) -> L P
  ++ totalGroebner(lp,lv) computes Groebner basis
  ++ for the list of polynomials lp with the terms
  ++ ordered first by total degree and then
  ++ refined by reverse lexicographic ordering.
  ++ The variables are ordered by their position in the list lv.

T == add
  lexGroebner(lp: L P,lv:L E) : L P ==
  PP:= PolToPol(lv,F)
  DPoly := DistributedMultivariatePolynomial(lv,F)
  DP:=DirectProduct(#lv,NNI)
  OV:=OrderedVariableList lv
  b:L DPoly:=[pToDmp(pol)$PP for pol in lp]
  gb:L DPoly :=groebner(b)$GroebnerPackage(F,DP,OV,DPoly)
  [dmpToP(pp)$PP for pp in gb]
  totalGroebner(lp: L P,lv:L E) : L P ==
  PP:= PolToPol(lv,F)
  HDPoly := HomogeneousDistributedMultivariatePolynomial(lv,F)
  HDP:=HomogeneousDirectProduct(#lv,NNI)
  OV:=OrderedVariableList lv
  b:L HDPoly:=[pToHdmp(pol)$PP for pol in lp]
  gb:=groebner(b)$GroebnerPackage(F,HDP,OV,HDPoly)
  [hdmpToP(pp)$PP for pp in gb]
package PAN2EXPR PolynomialAN2Expression

--- PolynomialAN2Expression.input ---

)set break resume
)sys rm -f PolynomialAN2Expression.output
)spool PolynomialAN2Expression.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialAN2Expression
--R
--R PolynomialAN2Expression is a package constructor
--R Abbreviation for PolynomialAN2Expression is PAN2EXPR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PAN2EXPR
--R
--R----------------------------- Operations -----------------------------
--R coerce : Polynomial(AlgebraicNumber) -> Expression(Integer)
--R coerce : Fraction(Polynomial(AlgebraicNumber)) -> Expression(Integer)
--R
--E 1

)spool
)lisp (bye)

---

--- PolynomialAN2Expression.help ---

====================================================================
PolynomialAN2Expression examples
====================================================================

This package provides a coerce from polynomials over algebraic numbers
to Expression AlgebraicNumber.

See Also:
o )show PolynomialAN2Expression

---
PolynomialAN2Expression (PAN2EXPR)

Exports:
coerce

| package PAN2EXPR PolynomialAN2Expression |

)abbrev package PAN2EXPR PolynomialAN2Expression
++ Author: Barry Trager
++ Date Created: 8 Oct 1991
++ Description:
++ This package provides a coerce from polynomials over 
++ algebraic numbers to \spadtype{Expression AlgebraicNumber}.

PolynomialAN2Expression():Target == Implementation where
  EXPR ==> Expression(Integer)
  AN ==> AlgebraicNumber
  PAN ==> Polynomial AN
  SY ==> Symbol
  Target ==> with
    coerce: Polynomial AlgebraicNumber -> Expression(Integer)
      ++ coerce(p) converts the polynomial \spad{p} with algebraic number
      ++ coefficients to \spadtype{Expression Integer}.
    coerce: Fraction Polynomial AlgebraicNumber -> Expression(Integer)
      ++ coerce(rf) converts \spad{rf}, a fraction of polynomial
      ++ \spad{p} with
      ++ algebraic number coefficients to \spadtype{Expression Integer}.
  Implementation == add
    coerce(p:PAN):EXPR ==
      map(x+->x::EXPR, y+-y::EXPR, p)$PolynomialCategoryLifting(
        IndexedExponents SY, SY, AN, PAN, EXPR)
    coerce(rf:Fraction PAN):EXPR ==
      numer(rf)::EXPR / denom(rf)::EXPR

______
package POLYLIFT PolynomialCategoryLifting

— PolynomialCategoryLifting.input —

)set break resume
/sys rm -f PolynomialCategoryLifting.output
/spool PolynomialCategoryLifting.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialCategoryLifting
--R
--R PolynomialCategoryLifting(E: OrderedAbelianMonoidSup,Vars: OrderedSet,R: Ring,P: PolynomialCategory(R,E,Vars),S)where
--R S: SetCategorywith
--R ?+? : (%,%) -> %
--R ?*? : (%,%) -> %
--R ?**? : (%,NonNegativeInteger) -> % is a package constructor
--R Abbreviation for PolynomialCategoryLifting is POLYLIFT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for POLYLIFT
--R
--R--------------------------------- Operations ---------------------------------
--R map : ((Vars -> S),(R -> S),P) -> S
--R
--)spool
)--lisp (bye)

— PolynomialCategoryLifting.help —
CHAPTER 17. CHAPTER P

PolynomialCategoryLifting examples

This package provides a very general map function, which given a set S and polynomials over R with maps from the variables into S and the coefficients into S, maps polynomials into S. S is assumed to support +, * and **.

See Also:
- )show PolynomialCategoryLifting

PolynomialCategoryLifting (POLYLIFT)

Exports:
- map

--- package POLYLIFT PolynomialCategoryLifting ---

)abbrev package POLYLIFT PolynomialCategoryLifting
++ Author: Manuel Bronstein
++ Description:
++ This package provides a very general map function, which
++ given a set S and polynomials over R with maps from the
++ variables into S and the coefficients into S, maps polynomials
++ into S. S is assumed to support \spad{+}, \spad{*} and \spad{**}.

PolynomialCategoryLifting(E,Vars,R,P,S): Exports == Implementation where
  E : OrderedAbelianMonoidSup
  Vars: OrderedSet
  R : Ring
PACKAGE POLYCATQ POLYNOMIALCATEGORYQUOTIENTFUNCTIONS

P : PolynomialCategory(R, E, Vars)
S : SetCategory with
   "+" : (%1, %2) -> %3
   "*" : (%1, %2) -> %3
   "**": (%1, NonNegativeInteger) -> %3

Exports ==> with
map: (Vars -> S, R -> S, P) -> S
   ++ map(varmap, coefmap, p) takes a
   ++ varmap, a mapping from the variables of polynomial p into S,
   ++ coefmap, a mapping from coefficients of p into S, and p, and
   ++ produces a member of S using the corresponding arithmetic.
   ++ in S

Implementation ==> add
map(fv, fc, p) ==
   (x1 := mainVariable p) case "failed" => fc leadingCoefficient p
   up := univariate(p, x1::Vars)
   t := fv(x1::Vars)
   ans := fc 0
   while not ground? up repeat
      ans := ans + map(fv, fc, leadingCoefficient up) * t ** (degree up)
      up := reductum up
      ans + map(fv, fc, leadingCoefficient up)

— POLYLIFT.dotabb —

"POLYLIFT" [color="#FF4488",hhref="bookvol10.4.pdf#nameddest=POLYLIFT"]
"PFECAT" [color="#4488FF",hhref="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYLIFT" -> "PFECAT"

— POLYCATQ PolynomialCategoryQuotientFunctions —

— PolynomialCategoryQuotientFunctions.input —

)set break resume
)sys rm -f PolynomialCategoryQuotientFunctions.output
)spool PolynomialCategoryQuotientFunctions.output
(set message test on)
(set message auto off)
clear all

--S 1 of 1
)show PolynomialCategoryQuotientFunctions
--R
--R PolynomialCategoryQuotientFunctions(E: OrderedAbelianMonoidSup,V: OrderedSet,R: Ring,P: PolynomialCategory(R,E,V),F)
--R F: Fieldwith
--R coerce : P -> %
--R numer : % -> P
--R denom : % -> P is a package constructor
--R Abbreviation for PolynomialCategoryQuotientFunctions is POLYCATQ
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for POLYCATQ
--R
--R---------------------------------------- Operations -----------------------------------
--R variables : F -> List(V)
--R isExpt : F -> Union(Record(var: V,exponent: Integer),"failed")
--R isPlus : F -> Union(List(F),"failed")
--R isPower : F -> Union(Record(val: F,exponent: Integer),"failed")
--R isTimes : F -> Union(List(F),"failed")
--R mainVariable : F -> Union(V,"failed")
--R multivariate : (Fraction(SparseUnivariatePolynomial(F)),V) -> F
--R univariate : (F,V) -> Fraction(SparseUnivariatePolynomial(F))
--R univariate : (F,V,SparseUnivariatePolynomial(F)) -> SparseUnivariatePolynomial(F)
--R
--E 1

)spool
)lisp (bye)

---

--- PolynomialCategoryQuotientFunctions.help ---

====================================================================
PolynomialCategoryQuotientFunctions examples
====================================================================

This package transforms multivariate polynomials or fractions into univariate polynomials or fractions, and back.

See Also:
  o )show PolynomialCategoryQuotientFunctions

---
PolynomialCategoryQuotientFunctions (POLYCATQ)

Exports:
  variables  isExpt  isPlus  isPower  isTimes
  mainVariable  multivariate  univariate

— package POLYCATQ PolynomialCategoryQuotientFunctions —

)abbrev package POLYCATQ PolynomialCategoryQuotientFunctions
++ Author: Manuel Bronstein
++ Date Created: March 1988
++ Date Last Updated: 9 July 1990
++ Description:
++ Manipulations on polynomial quotients
++ This package transforms multivariate polynomials or fractions into
++ univariate polynomials or fractions, and back.

PolynomialCategoryQuotientFunctions(E, V, R, P, F):
Exports == Implementation where
  E: OrderedAbelianMonoidSup
  V: OrderedSet
  R: Ring
  P: PolynomialCategory(R, E, V)
  F: Field with
    coerce: P -> %
    numer : % -> P
    denom : % -> P

  UP ==> SparseUnivariatePolynomial F
  RF ==> Fraction UP

Exports ==> with
  variables : F -> List V
    ++ variables(f) returns the list of variables appearing
    ++ in the numerator or the denominator of f.
  mainVariable: F -> Union(V, "failed")
    ++ mainVariable(f) returns the highest variable appearing
univariate : (F, V) -> RF  

++ univariate(f, v) returns f viewed as a univariate 
++ rational function in v.

multivariate: (RF, V) -> F  

++ multivariate(f, v) applies both the numerator and 
++ denominator of f to v.

univariate : (F, V, UP) -> UP  

++ univariate(f, x, p) returns f viewed as a univariate 
++ polynomial in x, using the side-condition \( \text{spad}(p(x) = 0) \).

isPlus : F -> Union(List F, "failed")  

++ isPlus(p) returns \([m1,...,mn]\) if \( \text{spad}(p = m1 + ... + mn) \) and 
++ \( \text{spad}(n > 1) \), "failed" otherwise.

isTimes : F -> Union(List F, "failed")  

++ isTimes(p) returns \spad{\{a1,...,an\}} if 
++ \( \text{spad}(p = a1 ... an) \) and \( \text{spad}(n > 1) \}, 
++ "failed" otherwise.

isExpt : F -> Union(Record(var:V, exponent:Integer), "failed")  

++ isExpt(p) returns \spad{\{x, n\}} if \( \text{spad}(p = x**n) \) and \( \text{spad}(n <> 0) \), 
++ "failed" otherwise.

isPower : F -> Union(Record(val:F, exponent:Integer), "failed")  

++ isPower(p) returns \spad{\{x, n\}} if \( \text{spad}(p = x**n) \) and \( \text{spad}(n <> 0) \),
++ "failed" otherwise.

Implementation ==> add

P2UP: (P, V) -> UP

univariate(f, x) == P2UP(numer f, x) / P2UP(denom f, x)

univariate(f, x, modulus) ==

(bc := extendedEuclidean(P2UP(denom f, x), modulus, 1))

case "failed" => error "univariate: denominator is 0 mod p"
(P2UP(numer f, x) * bc.coef1) rem modulus

multivariate(f, x) ==

v := x::P::F

((numer f) v) / ((denom f) v)

mymerge:(List V,List V) ->List V

mymerge(l:List V,m:List V):List V==

empty? l => m

empty? m => l

first l = first m => cons(first l,mymerge(rest l,rest m))

first l > first m => cons(first l,mymerge(rest l,m))

cons(first m,mymerge(l,rest m))

variables f ==

mymerge(variables numer f, variables denom f)
isPower f ==
  (den := denom f) ^= 1 =>
  numer f ^= 1 => "failed"
  (ur := isExpt den) case "failed" => [den::F, -1]
  r := ur::Record(var:V, exponent:NonNegativeInteger)
  [r.var::P::F, - (r.exponent::Integer)]
  (ur := isExpt numerator f) case "failed" => "failed"
  r := ur::Record(var:V, exponent:NonNegativeInteger)
  [r.var::P::F, r.exponent::Integer]

isExpt f ==
  (ur := isExpt numerator f) case "failed" =>
  -- one? numerator f =>
  (numerator f) = 1 =>
  (ur := isExpt denom f) case "failed" => "failed"
  r := ur::Record(var:V, exponent:NonNegativeInteger)
  [r.var, - (r.exponent::Integer)]
  "failed"
  r := ur::Record(var:V, exponent:NonNegativeInteger)
  -- one? denom f => [r.var, r.exponent::Integer]
  (denom f) = 1 => [r.var, r.exponent::Integer] "failed"

isTimes f ==
  t := isTimes(num := numerator f)
  l:Union(List F, "failed") :=
  t case "failed" => "failed"
  [x::F for x in t]
  -- one?(den := denom f) => 1
  (den := denom f) = 1 => 1
  -- one? num => "failed"
  num = 1 => "failed"
  d := inv(den::F)
  l case "failed" => [num::F, d]
  concat_!(l::List(F), d)

isPlus f ==
  denom f ^= 1 => "failed"
  (s := isPlus numerator f) case "failed" => "failed"
  [x::F for x in s]

mainVariable f ==
  a := mainVariable numerator f
  (b := mainVariable denom f) case "failed" => a
  a case "failed" => b
  max(a::V, b::V)

P2UP(p, x) ==
  map(z --> z::F,
  univariate(p, x))$SparseUnivariatePolynomialFunctions2(P, F)
package PCOMP PolynomialComposition

— PolynomialComposition.input —

)set break resume
)sys rm -f PolynomialComposition.output
)spool PolynomialComposition.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialComposition
--R
--R PolynomialComposition(UP: UnivariatePolynomialCategory(R),R: Ring) is a package constructor
--R Abbreviation for PolynomialComposition is PCOMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PCOMP
--R
--R------------------------ Operations ------------------------
--R compose : (UP,UP) -> UP
--R
--E 1

)spool
)lisp (bye)

— PolynomialComposition.help —
PolyonomialComposition examples
====================================================================
Polyonomial composition and decomposition functions.
If \( f = g \circ h \) then \( g = \text{leftFactor}(f, h) \) and \( h = \text{rightFactor}(f, g) \)
See Also:
o )show PolynomialComposition

---

PolyonomialComposition (PCOMP)

Exports:
compose

— package PCOMP PolynomialComposition —

)abbrev package PCOMP PolynomialComposition
++ References: Kozen and Landau, Cornell University  TR 86-773
++ Description:
++ Polynomial composition and decomposition functions\br
++ If \( f = g \circ h \) then \( g = \text{leftFactor}(f, h) \) and \( h = \text{rightFactor}(f, g) \)

PolynomialComposition(UP: UnivariatePolynomialCategory(R), R: Ring): with
  compose: (UP, UP) -> UP
  ++ compose(p, q) \undocumented
== add
  compose(g, h) ==
  r: UP := 0
  while g ^= 0 repeat
    r := leadingCoefficient(g)*h**degree(g) + r
g := reductum g
r

——

— PCOMP.dotabb —

"PCOMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PCOMP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PCOMP" -> "PFECAT"

——

package PDECOMP PolynomialDecomposition

—— PolynomialDecomposition.input ——

)set break resume
)sys rm -f PolynomialDecomposition.output
)spool PolynomialDecomposition.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialDecomposition
--R
--R PolynomialDecomposition(UP: UnivariatePolynomialCategory(F),F: Field) is a package constructor
--R Abbreviation for PolynomialDecomposition is PDECOMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PDECOMP
--R
--R------------------------------- Operations --------------------------------
--R decompose : UP -> List(UP)
--R decompose : (UP,NonNegativeInteger,NonNegativeInteger) -> Union(Record(left: UP,right: UP),"
--R leftFactor : (UP,UP) -> Union(UP,"failed")
--R rightFactorCandidate : (UP,NonNegativeInteger) -> UP
--R
--E 1

)spool
)lisp (bye)

——
PolynomialDecomposition examples

Polynomial composition and decomposition functions

If \( f = g \circ h \) then \( g = \text{leftFactor}(f, h) \) and \( h = \text{rightFactor}(f, g) \)

See Also:
\( \text{p)show PolynomialDecomposition} \)

Exports:
\text{decompose leftFactor rightFactorCandidate}
UP: UnivariatePolynomialCategory F
NNI ==> NonNegativeInteger
LR ==> Record(left: UP, right: UP)

PDcat == with
decompose: UP -> List UP
  ++ decompose(up) \ undocumented
decompose: (UP, NNI, NNI) -> Union(LR, "failed")
  ++ decompose(up,m,n) \ undocumented
leftFactor: (UP, UP) -> Union(UP, "failed")
  ++ leftFactor(p,q) \ undocumented
rightFactorCandidate: (UP, NNI) -> UP
  ++ rightFactorCandidate(p,n) \ undocumented
PDdef == add
leftFactor(f, h) ==
g: UP := 0
  for i in 0.. while f ^= 0 repeat
    fr := divide(f, h)
    f := fr.quotient; r := fr.remainder
    degree r > 0 => return "failed"
    g := g + r * monomial(1, i)
g
decompose(f, dg, dh) ==
df := degree f
dg*dh ^= df => "failed"
h := rightFactorCandidate(f, dh)
g := leftFactor(f, h)
g case "failed" => "failed"
[g::UP, h]
decompose f ==
df := degree f
  for dh in 2..df-1 | df rem dh = 0 repeat
    h := rightFactorCandidate(f, dh)
g := leftFactor(f, h)
g case UP => return
      append(decompose(g::UP), decompose h)
    [f]
rightFactorCandidate(f, dh) ==
f := f/leadingCoefficient f
df := degree f
dg := df quo dh
h := monomial(1, dh)
  for k in 1..dh repeat
    hdg:= h**dg
    c := (coefficient(f,(df-k)::NNI)-coefficient(hdg,(df-k)::NNI))/(dg::F)
    h := h + monomial(c, (dh-k)::NNI)
h - monomial(coefficient(h, 0), 0) -- drop constant term
package PFBR PolynomialFactorizationByRecursion

---

PolynomialFactorizationByRecursion.input ---

)set break resume
)sys rm -f PolynomialFactorizationByRecursion.output
)spool PolynomialFactorizationByRecursion.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialFactorizationByRecursion
--R
--R PolynomialFactorizationByRecursion(R: PolynomialFactorizationExplicit, E: OrderedAbelianMonoidSup, VarSet: OrderedSet, S: PolynomialCategory(R, E, VarSet)) is a package constructor
--R Abbreviation for PolynomialFactorizationByRecursion is PFBR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFBR
--R
--R------------------------ Operations -------------------------------
--R randomR : () -> R
--R bivariateSLPEBR : (List(SparseUnivariatePolynomial(S)), SparseUnivariatePolynomial(S), VarSet) -> Union(List(SparseUnivariatePolynomial(S)), "failed")
--R factorByRecursion : SparseUnivariatePolynomial(S) -> Factored(SparseUnivariatePolynomial(S))
--R factorSFBR1cUnit : (List(VarSet), SparseUnivariatePolynomial(S)) -> Factored(SparseUnivariatePolynomial(S))
--R factorSquareFreeByRecursion : SparseUnivariatePolynomial(S) -> Factored(SparseUnivariatePolynomial(S))
--R solveLinearPolynomialEquationByRecursion : (List(SparseUnivariatePolynomial(S)), SparseUnivariatePolynomial(S)) -> Factored(SparseUnivariatePolynomial(S))
--R
--E 1

)spool
)lisp (bye)

---

PolynomialFactorizationByRecursion.help ---
PolynomialFactorizationByRecursion (PFBR)

Exports:
randomR  bivariateSLPEBR  factorByRecursion
factorSFBReUnit  factorSquareFreeByRecursion  solveLinearPolynomialEquationByRecursion

—— package PFBR PolynomialFactorizationByRecursion ——

)abbrev package PFBR PolynomialFactorizationByRecursion
++ Description:
++ PolynomialFactorizationByRecursion(R,E,VarSet,S)
++ is used for factorization of sparse univariate polynomials over
++ a domain S of multivariate polynomials over R.

PolynomialFactorizationByRecursion(R,E, VarSet:OrderedSet, S): public ==
private where
  R:PolynomialFactorizationExplicit
  E:OrderedAbelianMonoidSup
  S:PolynomialCategory(R,E,VarSet)
  PI ==> PositiveInteger
  SupR ==> SparseUnivariatePolynomial R
SupSupR ==> SparseUnivariatePolynomial SupR
SupS ==> SparseUnivariatePolynomial S
SupSupS ==> SparseUnivariatePolynomial SupS
LPEBFS ==> LinearPolynomialEquationByFractions(S)

public == with
solveLinearPolynomialEquationByRecursion: (List SupS, SupS) -> Union(List SupS,"failed")
++ \spad{solveLinearPolynomialEquationByRecursion([p1,...,pn],p)}
++ returns the list of polynomials \spad{[q1,...,qn]}
++ such that \spad{\sum q_i/p_i = p / \prod p_i}, a
++ recursion step for solveLinearPolynomialEquation
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{solveLinearPolynomialEquation}).
++ If no such list of q_i exists, then "failed" is returned.

factorByRecursion: SupS -> Factored SupS
++ factorByRecursion(p) factors polynomial p. This function
++ performs the recursion step for factorPolynomial,
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{factorPolynomial}).

factorSquareFreeByRecursion: SupS -> Factored SupS
++ factorSquareFreeByRecursion(p) returns the square free
++ factorization of p. This functions performs
++ the recursion step for factorSquareFreePolynomial,
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{factorSquareFreePolynomial}).

randomR: -> R -- has to be global, since has alternative definitions
++ randomR produces a random element of R

bivariateSLPEBR: (List SupS, SupS, VarSet) -> Union(List SupS,"failed")
++ bivariateSLPEBR(lp,p,v) implements
++ the bivariate case of solveLinearPolynomialEquationByRecursion
++ its implementation depends on R

factorSFBRlcUnit: (List VarSet, SupS) -> Factored SupS
++ factorSFBRlcUnit(p) returns the square free factorization of
++ polynomial p
++ (see \spadfun{factorSquareFreeByRecursion}{PolynomialFactorizationByRecursionUnivariate})
++ in the case where the leading coefficient of p
++ is a unit.

private == add
supR: SparseUnivariatePolynomial R
pp: SupS
lpolys,factors: List SupS
vv:VarSet
lvpolys,lvpp: List VarSet
r:R
lr:List R
import FactoredFunctionUtilities(SupS)
import FactoredFunctions2(S,SupS)
import FactoredFunctions2(SupR,SupS)
import CommuteUnivariatePolynomialCategory(S,SupS, SupSupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,SupS,SupSupS)
import UnivariatePolynomialCategoryFunctions2(SupS,SupSupS,S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,R,SupR)
import UnivariatePolynomialCategoryFunctions2(R,SupR,S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,SupR,SupSupR)
import UnivariatePolynomialCategoryFunctions2(SupR,SupSupR,S,SupS)

hensel: (SupS,VarSet,R,List SupS) ->
    Union(Record(fctrs:List SupS),"failed")
chooseSLPEViableSubstitutions: (List VarSet,List SupS,SupS) ->
    Record(substnsField:List R,lpolysRField:List SupR,ppRField:SupR)
    --++ chooseSLPEViableSubstitutions(lv,lp,p) chooses substitutions
    --++ for the variables in first arg (which are all
    --++ the variables that exist) so that the polys in second argument don't
    --++ drop in degree and remain square-free, and third arg doesn't drop
    --++ drop in degree
chooseFSQViableSubstitutions: (List VarSet,SupS) ->
    Record(substnsField:List R,ppRField:SupR)
    --++ chooseFSQViableSubstitutions(lv,p) chooses substitutions for the variables in first arg (which are all
    --++ the variables that exist) so that the second argument poly doesn't
    --++ drop in degree and remains square-free
raise: SupR -> SupS
lower: SupS -> SupR
SLPEBR: (List SupS, List VarSet, SupS, List VarSet) ->
    Union(List SupS,"failed")
factorSFBRlcUnitInner: (List VarSet, SupS,R) ->
    Union(Factored SupS,"failed")

hensel(pp,vv,r,factors) ==
    origFactors:=factors
    totdegree:=Integer:=0
    proddegree:=Integer:=
        "max"/[degree(u,vv) for u in coefficients pp]
    n:PI:=1
    prime:=vv::S - r::S
    foundFactors:List SupS:=empty()
    while (totdegree <= proddegree) repeat
        pn:=prime**n
        Ecart:=(pp-*/factors) exquo pn
        Ecart case "failed" =>$
            error "failed lifting in hensel in PFBR"
        zero? Ecart =>$
            -- then we have all the factors
            return [append(foundFactors, factors)]
        step:=solveLinearPolynomialEquation(origFactors,
            map(z1 +-> eval(z1,vv,r),
            Ecart))
        step case "failed" =>$ return "failed" -- must be a false split
        factors:=[a+b*pn for a in factors for b in step]
        for a in factors for c in origFactors repeat
            pp1:= pp exquo a
            pp1 case "failed" =>$ "next"
        pp:=pp1
proddegree := proddegree - "max"/\[degree(u,vv)  for u in coefficients a]\n    factors:=remove(a,factors)
    origFactors:=remove(c,origFactors)
    foundFactors:=[a,:foundFactors]
    #factors < 2 =>
    return [(empty? factors => foundFactors;
               [pp,:foundFactors])]
    totdegree:= +/["max"/[degree(u,vv)
               for u in coefficients u1]
               for u1 in factors]
    n:=n+1
    "failed" -- must have been a false split

factorSFBRlcUnitInner(lvpp,pp,r) ==
  -- pp is square-free as a Sup, and its coefficients have precisely
  -- the variables of lvpp. Furthermore, its LC is a unit
  -- returns "failed" if the substitution is bad, else a factorization
  ppR:=map(z1 +-> eval(z1,first lvpp,r),pp)
  degree ppR < degree pp => "failed"
  degree gcd(ppR,differentiate ppR) >0 => "failed"
  factors:=
    empty? rest lvpp =>
      fDown:=factorSquareFreePolynomial map(z1 +-> retract(z1)::R,ppR)
      [raise (unit fDown * factorList(fDown).first.fctr),
       :[raise u.fctr for u in factorList(fDown).rest]]
      fSame:=factorSFBRlcUnit(rest lvpp,ppR)
      [unit fSame * factorList(fSame).first.fctr,
       :[uu.fctr for uu in factorList(fSame).rest]]
    #factors = 1 => makeFR(1,[["irred",pp,1]])
    hen:=hensel(pp,first lvpp,r,factors)
    hen case "failed" => "failed"
    makeFR(1,[["irred",u,1] for u in hen.fctrs])
  if R has StepThrough then
    factorSFBRlcUnit(lvpp,pp) ==
      val:R := init()
      while true repeat
        tempAns:=factorSFBRlcUnitInner(lvpp,pp,val)
        not (tempAns case "failed") => return tempAns
        val1:=nextItem val
        val case "failed" =>
          error "at this point, we know we have a finite field"
        val := val1
    else
      factorSFBRlcUnit(lvpp,pp) ==
        val:R := randomR()
        while true repeat
          tempAns:=factorSFBRlcUnitInner(lvpp,pp,val)
          not (tempAns case "failed") => return tempAns
          val := randomR()
if R has random: -> R then
  randomR() == random()
else randomR() == (random()$Integer)::R
if R has FiniteFieldCategory then
  bivariateSLPEBR(lpolys,pp,v) ==
    lpolysR:List SupSupR:=[map(univariate,u) for u in lpolys]
    ppR: SupSupR:=map(univariate,pp)
    ans:=solveLinearPolynomialEquation(lpolysR,ppR)$SupR
    ans case "failed" => "failed"
    [map(z1 +-> multivariate(z1,v),w) for w in ans]
else
  bivariateSLPEBR(lpolys,pp,v) ==
    solveLinearPolynomialEquationByFractions(lpolys,pp)$LPEBFS
chooseFSQViableSubstitutions(lvpp,pp) ==
  substns:List R
  ppR: SupR
  while true repeat
    substns:=[randomR() for v in lvpp]
    zero? eval(leadingCoefficient pp,lvpp,substns ) => "next"
    ppR:=map(z1 +->(retract eval(z1,lvpp,substns))::R,pp)
    degree gcd(ppR,differentiate ppR)>0 => "next"
    leave
  [substns,ppR]
chooseSLPEViableSubstitutions(lvpolys,lpolys,pp) ==
  substns:List R
  lpolysR:List SupR
  ppR: SupR
  while true repeat
    substns:=[randomR() for v in lvpolys]
    zero? eval(leadingCoefficient pp,lpolys,substns ) => "next"
    "or"/[zero? eval(leadingCoefficient u,lpolys,substns)
      for u in lpolys] => "next"
    lpolysR:=[map(z1 +-> (retract eval(z1,lpolys,substns))::R,u)
      for u in lpolys]
    uu:=lpolysR
    while not empty? uu repeat
      "or"/[degree(gcd(uu.first,v))>0 for v in uu.rest] => leave
      uu:=rest uu
      not empty? uu => "next"
    leave
    ppR:=map(z1 +-> (retract eval(z1,lpolys,substns))::R,pp)
  [substns,lpolysR,ppR]
raise(supR) == map(z1 +-> z1:R::S,supR)
lower(pp) == map(z1 +-> retract(z1)::R,pp)
SLPEBR(lpolys,lvpolys,pp,lvpp) ==
  not empty? (m:=setDifference(lvpp,lvpolys)) =>
    v:=first m
    lvpp:=remove(v,lvpp)
  pp1:SupSupS :=swap map(z1 +-> univariate(z1,v),pp)
  -- pp1 is mathematically equal to pp, but is in S[z][v]
-- so we wish to operate on all of its coefficients
ans:=List SupSupS:= [0 for u in lpolys]
for m in reverse_! monomials pp1 repeat
  ans1:=SLPEBR(lpolys,lvpolys,leadingCoefficient m,lvpp)
  ans1 case "failed" => return "failed"
  d:=degree m
  ans:=[monomial(a1,d)+a for a in ans for a1 in ans1]
[map(z1 +-> multivariate(z1,v),swap pp1) for pp1 in ans]
empty? lvpolys =>
  lpolysR:List SupR
  ppR:SupR
  lpolysR:=[map(retract,u) for u in lpolys]
  ppR:=map(retract,pp)
  ansR:=solveLinearPolynomialEquation(lpolysR,ppR)
  ansR case "failed" => return "failed"
  [map(z1 +-> z1::S,uu) for uu in ansR]
cVS:=chooseSLPEViableSubstitutions(lvpolys,lpolys,pp)
ansR:=solveLinearPolynomialEquation(cVS.lpolysRField,cVS.ppRField)
ansR case "failed" => "failed"
#lvpolys = 1 => bivariateSLPEBR(lpolys,pp, first lvpolys)
solveLinearPolynomialEquationByFractions(lpolys,pp)$LPEBFS

solveLinearPolynomialEquationByRecursion(lpolys,pp) ==
  lvpolys := removeDuplicates_!
    concat [ concat [variables z for z in coefficients u] for u in lpolys]
  lvpp := removeDuplicates_!
    concat [variables z for z in coefficients pp]
  SLPEBR(lpolys,lvpolys,pp,lvpp)

factorByRecursion pp ==
  lv:List(VarSet) := removeDuplicates_!
    concat [variables z for z in coefficients pp]
  empty? lv =>
    map(raise,factorPolynomial lower pp)
  c:=content pp
  unit? c => refine(squareFree pp,factorSquareFreeByRecursion)
  pp:=(pp exquo c)::SupS
  mergeFactors(refine(squareFree pp,factorSquareFreeByRecursion),
    map(z1 +-> z1:S::SupS,factor(c)$S))
  factorSquareFreeByRecursion pp ==
  lv:List(VarSet) := removeDuplicates_!
    concat [variables z for z in coefficients pp]
  empty? lv =>
    map(raise,factorPolynomial lower pp)
  unit? (lcpp := leadingCoefficient pp) => factorSFBR1cUnit(lv,pp)
oldnfact:NonNegativeInteger:= 999999
  -- I hope we never have to factor a polynomial
  -- with more than this number of factors
lcppPow:S
while true repeat
  cVS := chooseFSQViableSubstitutions(lv, pp)
factorsR := factorSquareFreePolynomial(cVS.ppRField)
  (nfact := numberOfFactors factorsR) = 1 =>
    return makeFR(1, [["irred", pp, 1]])
  -- OK, force all leading coefficients to be equal to the leading
  -- coefficient of the input
  nfact > oldnfact => "next" -- can't be a good reduction
  oldnfact := nfact
  factors := ([lcpp exquo leadingCoefficient u.fctr>::S * raise u.fctr
    for u in factorList factorsR]
  ppAdjust := (lcppPow := lcpp**#(rest factors)) * pp
  lvppList := lv
  OK := true
  for u in lvppList for v in cVS.substnsField repeat
    hen := hensel(ppAdjust, u, v, factors)
    hen case "failed" =>
      OK := false
      "leave"
    factors := hen.fctrs
  OK => leave
  factors := [(lc := content w;
    lcppPow := (lcppPow exquo lc)::S;
    (w exquo lc)::SupS)
    for w in factors]
  not unit? lcppPow =>
    error "internal error in factorSquareFreeByRecursion"
  makeFR((recip lcppPow)::S::SupS,
    [["irred", w, 1] for w in factors])

——

— PFBR.dotabb —

"PFBR" [color="#FF4488", href="bookvol10.4.pdf#nameddest=PFBR"]
"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFBR" -> "PFECAT"

——

package PFBRU PolynomialFactorizationByRecursionUnivariate

— PolynomialFactorizationByRecursionUnivariate.input —
---S 1 of 1
)show PolynomialFactorizationByRecursionUnivariate
--R PolynomialFactorizationByRecursionUnivariate(R: PolynomialFactorizationExplicit,S: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for PolynomialFactorizationByRecursionUnivariate is PFBRU
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PFBRU
--R
--R------------------------------------------------------------------- Operations ----------------------------------
--R randomR : () -> R
--R factorByRecursion : SparseUnivariatePolynomial(S) -> Factored(SparseUnivariatePolynomial(S))
--R factorSFBRlcUnit : SparseUnivariatePolynomial(S) -> Factored(SparseUnivariatePolynomial(S))
--R factorSquareFreeByRecursion : SparseUnivariatePolynomial(S) -> Factored(SparseUnivariatePolynomial(S))
--R solveLinearPolynomialEquationByRecursion : (List(SparseUnivariatePolynomial(S)),SparseUnivariatePolynomial) -> Factored(SparseUnivariatePolynomial(S))
--E 1

)spool
)lisp (bye)

---

--- PolynomialFactorizationByRecursionUnivariate.help ---

====================================================================
PolynomialFactorizationByRecursionUnivariate examples
====================================================================

PolynomialFactorizationByRecursionUnivariate
R is a PolynomialFactorizationExplicit domain,
S is univariate polynomials over R
We are interested in handling SparseUnivariatePolynomials over
S, a variable we shall call z

See Also:
o )show PolynomialFactorizationByRecursionUnivariate

---
PolynomialFactorizationByRecursionUnivariate (PFBRU)

Exports:
- randomR
- factorByRecursion
- factorSquareFreeByRecursion
- solveLinearPolynomialEquationByRecursion

<table>
<thead>
<tr>
<th>package PFBRU PolynomialFactorizationByRecursionUnivariate</th>
</tr>
</thead>
</table>

-- package PFBRU PolynomialFactorizationByRecursionUnivariate

)abbrev package PFBRU PolynomialFactorizationByRecursionUnivariate
++ Description:
++ PolynomialFactorizationByRecursionUnivariate
++ R is a \spadfun{PolynomialFactorizationExplicit} domain,
++ S is univariate polynomials over R
++ We are interested in handling SparseUnivariatePolynomials over
++ S, is a variable we shall call \( z \)

PolynomialFactorizationByRecursionUnivariate(R, S): public == private where
R:PolynomialFactorizationExplicit
S:UnivariatePolynomialCategory(R)
PI ==> PositiveInteger
SupR ==> SparseUnivariatePolynomial R
SupSupR ==> SparseUnivariatePolynomial SupR
SupS ==> SparseUnivariatePolynomial S
SupSupS ==> SparseUnivariatePolynomial SupS
LPEBFS ==> LinearPolynomialEquationByFractions(S)

public == with
solveLinearPolynomialEquationByRecursion: (List SupS, SupS) ->
Union(List SupS,"failed")
++ \spad{solveLinearPolynomialEquationByRecursion([p1,...,pn],p)}
++ returns the list of polynomials \spad{[q1,...,qn]}
++ such that \spad{sum q_i/p_i = p / prod p_i}, a
++ recursion step for solveLinearPolynomialEquation
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{solveLinearPolynomialEquation}).
++ If no such list of qi exists, then "failed" is returned.

factorByRecursion: SupS -> Factored SupS
++ factorByRecursion(p) factors polynomial p. This function
++ performs the recursion step for factorPolynomial,
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{factorPolynomial}).
factorSquareFreeByRecursion: SupS -> Factored SupS
++ factorSquareFreeByRecursion(p) returns the square free
++ factorization of p. This function performs
++ the recursion step for factorSquareFreePolynomial,
++ as defined in \spadfun{PolynomialFactorizationExplicit} category
++ (see \spadfun{factorSquareFreePolynomial}).
randomR: -> R -- has to be global, since has alternative definitions
++ randomR() produces a random element of R
factorSFBRlcUnit: (SupS) -> Factored SupS
++ factorSFBRlcUnit(p) returns the square free factorization of
++ polynomial p
++ (see \spadfun{factorSquareFreeByRecursion}{PolynomialFactorizationByRecursionUnivariate})
++ in the case where the leading coefficient of p
++ is a unit.

private == add
supR: SparseUnivariatePolynomial R
pp: SupS
lpolys,factors: List SupS
r:R
lr:List R
import FactoredFunctionUtilities(SupS)
import FactoredFunctions2(SupR,SupS)
import FactoredFunctions2(S,SupS)
import UnivariatePolynomialCategoryFunctions2(S,SupS,R,SupR)
import UnivariatePolynomialCategoryFunctions2(R,SupR,S,SupS)
-- local function declarations
raise: SupR -> SupS
lower: SupS -> SupR
factorSFBRlcUnitInner: (SupS,R) -> Union(Factored SupS,"failed")
hensel: (SupS,R,List SupS) ->
chooseFSQViableSubstitutions: (SupS) ->
Record(substnsField: R,ppRField: SupR)
--- chooseFSQViableSubstitutions(p), p is a sup
--- ("sparse univariate polynomial")
--- over a sup over R, returns a record
--- \spad{[substnsField: r, ppRField: q]} where r is a substitution point
--- q is a sup over R so that the (implicit) variable in q
--- does not drop in degree and remains square-free.
-- here for the moment, until it compiles
-- N.B., we know that R is NOT a FiniteField, since
-- that is meant to have a special implementation, to break the
-- recursion
solveLinearPolynomialEquationByRecursion(lpolys,pp) ==
lhsdeg:="max"/"[max]/[degree v for v in coefficients u] for u in lpolys"
rhsdeg:="max"/"[degree v for v in coefficients pp]"
lhsdeg = 0 =>
lpolysLower := [lower u for u in lpolys]
answer: List SupS := [0 for u in lpolys]
for i in 0..rhsdeg repeat
  ppx := map((z1: S): R +-> coefficient(z1, i), pp)
  zero? ppx => "next"
  recAns := solveLinearPolynomialEquation(lpolysLower, ppx)
  recAns case "failed" => return "failed"
  answer := [monomial(1, i)$S * raise c + d
    for c in recAns for d in answer]
answer
solveLinearPolynomialEquationByFractions(lpolys, pp)$LPEBFS

-- local function definitions
hensel(pp, r, factors) ==
  -- factors is a relatively prime factorization of pp modulo
  -- (x-r), with suitably imposed leading coefficients.
  -- This is lifted, without re-combinations, to a factorization
  -- if this can't be done
  origFactors := factors
  totdegree: Integer := 0
  proddegree: Integer := "max"/[degree(u) for u in coefficients pp]
  n: PI := 1
  pn := prime := monomial(1, 1) - r::S
  foundFactors: List SupS := empty()
  while (totdegree <= proddegree) repeat
    Ecart := (pp - */factors) exquo pn
    Ecart case "failed" =>
      error "failed lifting in hensel in PFBRU"
    zero? Ecart =>
      return [append(foundFactors, factors)]
    step := solveLinearPolynomialEquation(origFactors, map(z1 +-> elt(z1, r::S), Ecart))
    step case "failed" => return "failed" -- must be a false split
    factors := [a + b*pn for a in factors for b in step]
    for a in factors for c in origFactors repeat
      pp1 := pp exquo a
      pp1 case "failed" => "next"
      pp := pp1
      proddegree := proddegree - "max"/[degree(u)
        for u in coefficients a]
      factors := remove(a, factors)
      origFactors := remove(c, origFactors)
      foundFactors := [a, :foundFactors]
    #factors < 2 =>
      return [(empty? factors => foundFactors;
        [pp, :foundFactors])]
    totdegree := +/"max"/[degree(u)
for \( u \) in coefficients \( u_1 \)
for \( u_1 \) in factors
\[ n := n + 1 \]
\[ p_n := p_n \times \text{prime} \]
"failed" -- must have been a false split
chooseFSQViableSubstitutions(pp) ==

\[ \text{substns} : \mathbb{R} \]
\[ \text{ppR} : \text{SupR} \]
while true repeat
\[ \text{substns} := \text{randomR()} \]
zero? \( \text{elt}(\text{leadingCoefficient pp, substns}) \) => "next"
\[ \text{ppR} := \text{map}(z_1 \mapsto \text{elt}(z_1, \text{substns}), \text{pp}) \]
\[ \text{degree gcd(ppR, differentiate ppR)} > 0 \Rightarrow "next" \]
leave
\[ [\text{substns, ppR}] \]
raise(\text{supR}) == \text{map}(z_1 \mapsto z_1 : \mathbb{R} :: \text{S}, \text{supR})

lower(pp) == \text{map}(z_1 \mapsto \text{retract}(z_1) :: \mathbb{R}, pp)

factorSFBRlcUnitInner(pp, r) ==

-- \( pp \) is square-free as a \( \text{Sup} \), but the \( \text{Up} \) variable occurs.
-- Furthermore, its \( \text{LC} \) is a unit
-- returns "failed" if the substitution is bad, else a factorization
\[ \text{ppR} := \text{map}(z_1 \mapsto \text{elt}(z_1, r), \text{pp}) \]
\[ \text{degree ppR} < \text{degree pp} \Rightarrow "failed" \]
\[ \text{degree gcd(ppR, differentiate ppR)} > 0 \Rightarrow "failed" \]
\[ \text{factors} := \]
\[ \text{fDown} := \text{factorSquareFreePolynomial ppR} \]
\[ [\text{raise (unit fDown} \ast \text{factorList(fDown).first.fctr)}, \]
\[ : [\text{raise u.fctr for u in factorList(fDown).rest]}] \]
\#factors = 1 \Rightarrow \text{makeFR(1, ["irred", pp, 1])}

\text{hen} := \text{hensel(pp, r, factors)}

\text{hen} case "failed" \Rightarrow "failed"

\text{makeFR(1, ["irred", u, 1] for u in hen.fctrs)}

-- exported function definitions
if \( \text{R has StepThrough} \) then

\text{factorSFBRlcUnit}(pp) ==
\[ \text{val} : \text{R} := \text{init()} \]
while true repeat
\[ \text{tempAns} := \text{factorSFBRlcUnitInner}(pp, \text{val}) \]
not (tempAns case "failed") \Rightarrow return tempAns
\[ \text{val1} := \text{nextItem val} \]
\[ \text{val1} \text{ case "failed" } \Rightarrow \]
\[ \text{error "at this point, we know we have a finite field"} \]
\[ \text{val} := \text{val1} \]
else

\text{factorSFBRlcUnit}(pp) ==
\[ \text{val} : \text{R} := \text{randomR()} \]
while true repeat
\[ \text{tempAns} := \text{factorSFBRlcUnitInner}(pp, \text{val}) \]
not (tempAns case "failed") \Rightarrow return tempAns
\[ \text{val} := \text{randomR()} \]
if R has StepThrough then
  randomCount:=init()
  randomR() ==
    v:=nextItem(randomCount)
    v case "failed" =>
      SAY$Lisp "Taking another set of random values"
      randomCount:=init()
      randomCount
    randomCount:=v
  randomCount
else if R has random: -> R then
  randomR() == random()
else randomR() == (random($Integer rem 100)::R)

factorByRecursion pp ==
  and/[zero? degree u for u in coefficients pp] =>
    map(raise,factorPolynomial lower pp)
  c:=content pp
  unit? c => refine(squareFree pp,factorSquareFreeByRecursion)
  pp:=(pp exquo c)::SupS
  mergeFactors(refine(squareFree pp,factorSquareFreeByRecursion),
    map(z1 ++> z1:S::SupS,factor(c)$S))

factorSquareFreeByRecursion pp ==
  and/[zero? degree u for u in coefficients pp] =>
    map(raise,factorSquareFreePolynomial lower pp)
  cVS:=chooseFSQViableSubstitutions(pp)
  newppR:=primitivePart cVS.ppRField
  factorsR:=factorSquareFreePolynomial(newppR)
  (nfact:=numberOfFactors factorsR) = 1 =>
    return makeFR(1,["irred",pp,1])
  lcppPow:S
  while true repeat -- a loop over possible false splits
    cVS:=choosePSQViableSubstitutions(pp)
    newppR:=primitivePart cVS.ppRField
    factorsR:=factorSquareFreePolynomial(newppR)
    (nfact:=numberOfFactors factorsR) = 1 =>
      return makeFR(1,["irred",pp,1])
    oldnfact:=nfact
    lcppR:=leadingCoefficient cVS.ppRField
    factors:=[raise(((lcppR exquo leadingCoefficient u.fctr) ::R * u.fctr)
      for u in factorList factorsR]
    -- factors now multiplies to give cVS.ppRField * lcppR^(#factors-1)
    -- Now change the leading coefficient to be lcpp
    factors:=[(lcpp exquo leadingCoefficient u.fctr):S * raise u.fctr
      for u in factorList factorsR]
    ppAdjust:=(lcppPow:=lcpp**(#factors-1)) * pp
    OK:=true
    hen:=hensel(ppAdjust,cVS.substnsField,factors)
hen case "failed" => "next"
factors:=hen.fctrs
leave
factors:=[(lc:=content w;
lcppPow:=(lcppPow exquo 1c)::S;
(w exquo 1c)::SupS)
for w in factors]
not unit? lcppPow =>
    error "internal error in factorSquareFreeByRecursion"
makeFR((recip lcppPow)::S::SupS,
[['irred',w,1] for w in factors])

---

PFBRU.dotabb ---

"PFBRU" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PFBRU"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PFBRU" -> "PFECAT"

---

package POLY2 PolynomialFunctions2

--- PolynomialFunctions2.input ---

)set break resume
)sys rm -f PolynomialFunctions2.output
)spool PolynomialFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PolynomialFunctions2
--R
--R PolynomialFunctions2(R: Ring,S: Ring) is a package constructor
--R Abbreviation for PolynomialFunctions2 is POLY2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for POLY2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((R -> S),Polynomial(R)) -> Polynomial(S)
--R
This package takes a mapping between coefficient rings, and lifts it to a mapping between polynomials over those rings.

See Also:
- )show PolynomialFunctions2

PolynomialFunctions2 (POLY2)

Exports:
map

--E 1

)spool
)lisp (bye)
PolynomialFunctions2(R:Ring, S:Ring): with
map: (R -> S, Polynomial R) -> Polynomial S
++ map(f, p) produces a new polynomial as a result of applying
++ the function f to every coefficient of the polynomial p.
== add
map(f, p) == map(x1 +-> x1::Polynomial(S), x2 +-> f(x2)::Polynomial(S),
p)$PolynomialCategoryLifting(IndexedExponents Symbol,
Symbol, R, Polynomial R, Polynomial S)

package PGCD PolynomialGcdPackage

--- PolynomialGcdPackage.input ---

)set break resume
)sys rm -f PolynomialGcdPackage.output
)spool PolynomialGcdPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialGcdPackage
--R
--R PolynomialGcdPackage(E: OrderedAbelianMonoidSup,OV: OrderedSet,R: EuclideanDomain,P: PolynomialCategory) is a package constructor
--R Abbreviation for PolynomialGcdPackage is PGCD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PGCD
--R
--R------------------------------------------ Operations ------------------------------------------
--R gcd : (P,P) -> P gcd : List(P) -> P
--R gcdPrimitive : (P,P) -> P gcdPrimitive : List(P) -> P
--R gcd : (SparseUnivariatePolynomial(P),SparseUnivariatePolynomial(P)) -> SparseUnivariatePolynomial(P)
CHAPTER 17. CHAPTER P

---R gcd : List(SparseUnivariatePolynomial(P)) -> SparseUnivariatePolynomial(P)
---R gcdPrimitive : (SparseUnivariatePolynomial(P),SparseUnivariatePolynomial(P)) -> SparseUnivariatePolynomial(P)
---E 1

)spool
)lisp (bye)

———

— PolynomialGcdPackage.help —

=====================================================================
PolynomialGcdPackage examples
=====================================================================

This package computes multivariate polynomial gcd's using a hensel
lifting strategy. The contraint on the coefficient domain is imposed
by the lifting strategy. It is assumed that the coefficient domain has
the property that almost all specializations preserve the degree of
the gcd.

See Also:
  o )show PolynomialGcdPackage

———

PolynomialGcdPackage (PGCD)

PGCD

PFECAT

Exports:
  gcd  gcdPrimitive

The simple Euclidean division algorithm,
given $a \neq 0$, $b \neq 0$ where $a$ and $b$ are two polynomials in the variable $x$, then

output $q$, the quotient and $r$, the remainder

begin

Let

$q = 0$
$r = a$
$d = \text{degree}(b)$
$c = \text{leadingCoefficient}(b)$

while $\text{degree}(r) \geq d$ do

$e = \text{degree}(r) - d$
$s = \frac{\text{leadingCoefficient}(r)}{e} x^{e}$
$q = q + s$
$r = r - sb$

endwhile

return $(q, r)$

end

For example, given

\[
p_1 := (x+1) \cdot (x+6) \\
p_2 := (x+1) \cdot (x-6)
\]
\[
gcd(p_1, p_2) \rightarrow x+1
\]
SUPP \implies\text{SparseUnivariatePolynomial P}

C == with
  gcd : (P,P) \rightarrow P
  \begin{align*}
  \text{gcd}(p,q) \text{ computes the gcd of the two polynomials } p \text{ and } q. \\
  \text{gcd}\left(p_1 := (x+1)*(x+6)\right) \\
  \text{gcd}\left(p_2 := (x+1)*(x-6)\right) \\
  \text{gcd}(p,q)
  \end{align*}

  gcd : List P \rightarrow P
  \begin{align*}
  \text{gcd}\left(lp\right) \text{ computes the gcd of the list of polynomials } lp.
  \end{align*}

  gcd : (SUPP,SUPP) \rightarrow SUPP
  \begin{align*}
  \text{gcd}(p,q) \text{ computes the gcd of the two polynomials } p \text{ and } q.
  \end{align*}

  gcd : List SUPP \rightarrow SUPP
  \begin{align*}
  \text{gcd}\left(lp\right) \text{ computes the gcd of the list of polynomials } lp.
  \end{align*}

  gcdPrimitive : (P,P) \rightarrow P
  \begin{align*}
  \text{gcdPrimitive}(p,q) \text{ computes the gcd of the primitive polynomials } p \text{ and } q.
  \end{align*}

  gcdPrimitive : (SUPP,SUPP) \rightarrow SUPP
  \begin{align*}
  \text{gcdPrimitive}(p,q) \text{ computes the gcd of the primitive polynomials } p \text{ and } q.
  \end{align*}

  gcdPrimitive : List P \rightarrow P
  \begin{align*}
  \text{gcdPrimitive}(lp) \text{ computes the gcd of the list of primitive polynomials } lp.
  \end{align*}

T == add

SUP \implies\text{SparseUnivariatePolynomial R}

L\text{Gcd} \implies\text{Record}(locgcd:SUPP,goodint:List List R)

U\text{Term} \implies\text{Record}(lpol:List SUP,lint:List List R,mpol:SUPP)

pmod:R := (\text{prevPrime}(2**26)$\text{IntegerPrimesPackage}(\text{Integer}))::R

import \text{MultivariateLifting}(E,OV,R,P)

import \text{FactoringUtilities}(E,OV,R,P)

---------- Local Functions ----------

\text{myran} : Integer \rightarrow \text{Union}(R,\text{"failed"})

\text{better} : (P,P) \rightarrow \text{Boolean}

\text{failtest} : (SUPP,SUPP,SUPP) \rightarrow \text{Boolean}

\text{monomContent} : (SUPP) \rightarrow SUPP

\text{gcdMonom} : (SUPP,SUPP) \rightarrow SUPP

\text{gcdTermList} : (P,P) \rightarrow P

\text{good} : (SUPP,List OV,List List R) \rightarrow \text{Record}(upol:SUP,invalid:List List R)

\text{chooseVal} : (SUPP,SUPP,List OV,List List R) \rightarrow \text{Union}(U\text{Term},\text{"failed"})

\text{localgcd} : (SUPP,SUPP,List OV,List List R) \rightarrow L\text{Gcd}

\text{notCoprime} : (SUPP,SUPP, List \text{NNI},List OV,List List R) \rightarrow SUPP

\text{imposeic} : (List SUP,List OV,List R,List P) \rightarrow
Union(List SUP, "failed")

lift? :(SUPP,SUPP,UTerm,List NNI,List OV) -> _
    Union(s:SUPP,failed:"failed",notCoprime:"notCoprime")
lift :(SUPP,SUPP,SUPP,P,List OV,List NNI,List R) -> Union(SUPP,"failed")

---- Local functions ----
-- test if something wrong happened in the gcd
failtest(f:SUPP,p1:SUPP,p2:SUPP) : Boolean ==
  (p1 exquo f) case "failed" or (p2 exquo f) case "failed"

-- Choose the integers
chooseVal(p1:SUPP,p2:SUPP,lvr:List OV,_
  ltry:List List R):Union(UTerm,"failed") ==
  d1:=degree(p1)
  d2:=degree(p2)
  dd:NNI:=0$NNI
  nvr:NNI:=#lvr
  lval:List R :=[]
  range:I:=8
  repeat
    range:=2*range
    lval:=[ran(range) for i in 1..nvr]
    member?(lval,ltry) => "new point"
    ltry:=cons(lval,ltry)
    uf1:SUP:=completeEval(p1,lvr,lval)
    degree uf1 ^= d1 => "new point"
    uf2:SUP:= completeEval(p2,lvr,lval)
    degree uf2 ^= d2 => "new point"
    u:=gcd(uf1,uf2)
    du:=degree u
    --the univariate gcd is 1
    if du=0 then return [[1$SUP],ltry,0$SUPP]$UTerm
  end repeat
  ugcd:List SUP:=[u,(uf1 exquo u)::SUP,(uf2 exquo u)::SUP]
  uterm:=[ugcd,ltry,0$SUPP]$UTerm
  dd=0 => dd:=du

--the degree is not changed
du=dd =>

--test if one of the polynomials is the gcd
dd=d1 =>
  if ^((f:=p2 exquo p1) case "failed") then
    return [[u],ltry,p1]$UTerm
  if dd^=d2 then dd:=(dd-1)::NNI
dd=d2 =>
  if ^((f:=p1 exquo p2) case "failed") then
    return [[u],ltry,p2]$UTerm
\[\text{dd} := (\text{dd}-1) :: \text{NNI} \]
\[\text{return uterm} \]

-- the new gcd has degree less
\[\text{du} < \text{dd} \Rightarrow \text{dd} := \text{du} \]

\[\text{good}(f: \text{SUPP}, lvr: \text{List OV}, _ \text{ltry: List List R}) := \]
\[\text{nvr}\!::\!\text{NNI}:=\#lvr \]
\[\text{range}::I:=1 \]
\[\text{while true repeat} \]
\[\text{range}:=2*\text{range} \]
\[\text{lval}:=\{\text{ran}(\text{range}) \text{ for } i \text{ in } 1..\text{nvr}\} \]
\[\text{member?}(\text{lval}, \text{ltry}) \Rightarrow "\text{new point}" \]
\[\text{ltry}::\text{cons}(\text{lval}, \text{ltry}) \]
\[\text{uf}::\text{completeEval}(f, \text{lvr}, \text{lval}) \]
\[\text{if degree gcd}(\text{uf}, \text{differentiate uf})=0 \text{ then return } [\text{uf}, \text{ltry}] \]

In Gathen [Gath99] we find a discussion of applying the Euclidean algorithm to elements of a field. In a field every nonzero rational number is a unit. If we want to define a single element such that
\[\gcd(f, g) \in \mathbb{Q}[x] \]
we choose a monic polynomial, that is, the element with a leading coefficient of 1. So
\[abs(a/\text{leadingCoefficient}(a)) \]
defines a normal form of \(a\).

Gathen also notes that for polynomials we should modify the Euclidean algorithm so all remainders are normalized. Without this constraint the remainders will have huge numerators and denominators.

--- package PGCD PolynomialGcdPackage ---

\[\text{-- impose the right leading condition, check for failure.} \]
\[\text{imposelc(lipol:List SUP, lvar:List OV, lval:List R, leadc:List P): Union(List SUP, "failed")} \]
\[\text{result:List SUP :=[]} \]
\[\text{for pol in lipol for leadpol in leadc repeat} \]
\[\text{pl := univariate eval(leadpol, lvar, lval) * pol} \]
\[\text{plu := pl exquo leadingCoefficient pol} \]
\[\text{plu case "failed"} \Rightarrow \text{return "failed"} \]
\[\text{result := cons(plu::SUP, result)} \]
\[\text{reverse result} \]

\[\text{--Compute the gcd between not coprime polynomials} \]
\[\text{notCoprime(g:SUPP, p2:SUPP, ldeg:List NNI, _} \]}
lvar1:List OV, ltry:List List R) : SUPP ==
g1:=gcd(g,differentiate g)
l1 := (g exquo g1)::SUPP
glg:LGcd:=localgcd(l1,p2,lvar1,ltry)
(1,ltry):=(lg.locgcd,lg.goodint)
lval:=ltry.first
p2l:=(p2 exquo 1)::SUPP
(gd1,gd2):=(1,1)
ul:=completeEval(l1,lvar1,lval)
d1:=degree ul
if degree gcd(ul,differentiate ul) ^=0 then
  newchoice:=good(1,lvar1,ltry)
  ul:=newchoice.upol
  ltry:=newchoice inval
  lval:=ltry.first
ug1:=completeEval(g1,lvar1,lval)
ulist:=[ug1,completeEval(p2l,lvar1,lval)]
lcpol:List P:=[leadingCoefficient g1, leadingCoefficient p2]
while true repeat
d:SUP:=gcd(cons(ul,ulist))
if degree d =0 then return gd1
ulquo:=(ul exquo d)::SUP
if degree ulquo ^=0 then
  lgcd:=gcd(cons(leadingCoefficient 1,lcpol))
  (gd1:=lift(1,d,ulquo,lgcd,lvar1,ldeg,lval)) case "failed" =>
    return notCoprime(g,p2,ldeg,lvar1,ltry)
  l:=gd2:=gd1::SUPP
  ul:=completeEval(1,lvar1,lval)
d1:=degree ul
  gd1:=gd1*gd2
  ulist:=[(uf exquo d)::SUP for uf in ulist]
gcdPrimitive(p1:SUPP,p2:SUPP) : SUPP ==
if (d1:=degree(p1)) > (d2:=degree(p2)) then
  (p1,p2):= (p2,p1)
  (d1,d2):= (d2,d1)
degree p1 = 0 =>
  p1 = 0 => unitCanonical p2
  unitCanonical p1
lvar:List OV:=
  sort((a:OV,b:OV):Boolean+->a>b,setUnion(variables p1,variables p2))
empty? lvar =>
  raisePolynomial(gcd(lowerPolynomial p1,lowerPolynomial p2))
(p2 exquo p1) case SUPP => unitCanonical p1
ltry:List List R:=empty()
totResult:=localgcd(p1,p2,lvar,ltry)
result: SUPP:=totResult.locgcd
-- special cases
result=1 => 1$SUPP
while failtest(result,p1,p2) repeat
-- SayLisp "retrying gcd"
ltry:=totResult.goodint
totResult:=localgcd(p1,p2,lvar,ltry)
result:=totResult.locgcd
result

-- Local function for the gcd: it returns the evaluation point too
localgcd(p1:SUPP,p2:SUPP,lvar:List(OV),ltry:List List R) : LGcd ==
  uterm:=chooseVal(p1,p2,lvar,ltry)::UTerm
  ltry:=uterm.lint
  listpol:= uterm.lpol
  ud:=listpol.first
  dd:= degree ud

  -- The univariate gcd is 1
  dd=0 => [1$SUPP,ltry]$LGcd

  -- One of the polynomials is the gcd
  dd=degree(p1) or dd=degree(p2) =>
    [uterm.mpol,ltry]$LGcd
  ldeg:List NNI:=map(min,degree(p1,lvar),degree(p2,lvar))

  -- If there is a polynomial g s.t. g/gcd and gcd are coprime ...
  -- I can lift
  (h:=lift?(p1,p2,uterm,ldeg,lvar)) case notCoprime =>
    [notCoprime(p1,p2,ldeg,lvar,ltry),ltry]$LGcd
  h case failed => localgcd(p1,p2,lvar,ltry) -- Skip bad values?
    [h.s,ltry]$LGcd

-- Content, internal functions return the poly if it is a monomial
monomContent(p:SUPP):SUPP ==
  degree(p)=0 => 1
  md:= minimumDegree(p)
  monomial(gcd sort(better,coefficients p),md)

-- Ordering for gcd purposes
better(p1:P,p2:P):Boolean ==
  ground? p1 => true
  ground? p2 => false
  degree(p1,mainVariable(p1)::OV) < degree(p2,mainVariable(p2)::OV)

best_to_front(l : List P) : List P ==
  ress := []
  best := first(l)
  for p in rest l repeat
    if better(p, best) then
      ress := cons(best, ress)
      best := p
    else
Hensel's lifting lemma states that if a polynomial equation has a simple root modulo a prime number \( p \), then this root corresponds to a unique root of the same equation modulo any higher power of \( p \), which can be found by iteratively “lifting” the solution modulo successive powers of \( p \).

See Volume 10.1 for more details.
uf := listpol(1)
f := p1
--note uf and d not necessarily primitive
if degree gcd(uf, d) != 0 then
    uf := listpol(2)
f := p2
if degree gcd(uf, d) != 0 then return "notCoprime"
lgcd := gcd(leadingCoefficient p1, leadingCoefficient p2)
l := lift(f, d, uf, lgcd, lvar, ldeg, lval)
l case "failed" => "failed"
[l : SUPP]

-- interface with the general "lifting" function
lift(f:SUPP,d:SUP,uf:SUP,lgcd:P,lvar:List OV,
ldeg:List NNI,lval:List R):Union(SUPP,"failed") ==

leadpol : Boolean := false
lcf : P
lcf := leadingCoefficient f
df := degree f
leadlist : List(P) := []

if lgcd != 1 then
    leadpol := true
    f := lgcd*f
    ldeg := [n0+n1 for n0 in ldeg for n1 in degree(lgcd, lvar)]
    ldeg := degree(lgcd) = 0 => retract lgcd
    retract(eval(lgcd, lvar, lval))
    du := (lgcd1*d) exquo lcd
    du case "failed" => "failed"
    d := du::SUP
    uf := lcd*uf
    leadlist := [lgcd, lcf]
    lgu := imposelc([d, uf], lvar, lval, leadlist)
    lgu case "failed" => "failed"
    lg := lgu::List(SUP)
    (pl := lifting(f,lvar,lg,leadlist,ldeg,pmod)) case "failed" => "failed"
    plist := pl :: List SUPP
    (p0 : SUPP, p1 : SUPP) := (plist.first, plist.2)
    if completeEval(p0, lvar, lval) "= lg.first then
        (p0, p1) := (p1, p0)
        not leadpol => p0
        p0 exquo content(p0)

-- Gcd for two multivariate polynomials
gcd(p1:P,p2:P) : P ==
    ground? p1 =>
        p1 := unitCanonical p1
p1 = 1$P => p1
p1 = 0$P => unitCanonical p2
ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
gcdTermList(p1,p2)
ground? p2 =>
p2 := unitCanonical p2
p2 = 1$P => p2
p2 = 0$P => unitCanonical p1
gcdTermList(p2,p1)
(p1:= unitCanonical(p1)) = (p2:= unitCanonical(p2)) => p1
mv1:= mainVariable(p1)::OV
mv2:= mainVariable(p2)::OV
mv1 = mv2 => multivariate(gcd(univariate(p1,mv1),
univariate(p2,mv1)),mv1)
mv1 < mv2 => gcdTermList(p1,p2)
gcdTermList(p2,p1)

-- Gcd for a list of multivariate polynomials
gcd(listp:List P) : P ==
lf := best_to_front(listp)
f:=lf.first
for g in lf.rest repeat
  f:=gcd(f,g)
  if f=1$P then return f
f

gcd(listp:List SUPP) : SUPP ==
lf:=sort((z1:SUPP,z2:SUPP):Boolean +-> degree(z1)<degree(z2),listp)
f:=lf.first
for g in lf.rest repeat
  f:=gcd(f,g)
  if f=1 then return f
f

-- Gcd for primitive polynomials
gcdPrimitive(p1:P,p2:P):P ==
(p1:= unitCanonical(p1)) = (p2:= unitCanonical(p2)) => p1
ground? p1 =>
ground? p2 => gcd((retract p1)@R,(retract p2)@R)::P
p1 = 0$P => p2
1$P
ground? p2 =>
p2 = 0$P => p1
1$P
mv1:= mainVariable(p1)::OV
mv2:= mainVariable(p2)::OV
mv1 = mv2 =>
md:=min(minimumDegree(p1,mv1),minimumDegree(p2,mv2))
mp:=1$P
```lisp
if md>1 then
    mp:=(mv1::P)**md
    p1:=(p1 exquo mp)::P
    p2:=(p2 exquo mp)::P
    up1 := univariate(p1,mv1)
    up2 := univariate(p2,mv2)
    mp*multivariate(gcdPrimitive(up1,up2),mv1)
```

-- Gcd for a list of primitive multivariate polynomials
gcdPrimitive(listp:List P) : P ==
    lf:=sort(better,listp)
    f:=lf.first
    for g in lf.rest repeat
        f:=gcdPrimitive(f,g)
        if f=1$P then return f
    f

— PGCD.dotabb —

"PGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PGCD"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PGCD" -> "PFECAT"

package PINTERP PolynomialInterpolation

— PolynomialInterpolation.input —

)set break resume
)sys rm -f PolynomialInterpolation.output
)spool PolynomialInterpolation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialInterpolation
--R
--R PolynomialInterpolation(xx: Symbol,F: Field) is a package constructor
--R Abbreviation for PolynomialInterpolation is PINTERP
This package exports interpolation algorithms

See Also:
o )show PolynomialInterpolation

---

PolynomialInterpolation (PINTERP)

Exports:
interpolate

---

package PINTERP PolynomialInterpolation ---
\texttt{CHAPTER 17. CHAPTER P}

)abbrev package PINTERP PolynomialInterpolation
++ Description:
++ This package exports interpolation algorithms

\texttt{PolynomialInterpolation}(xx, F): Cat == Body where
\begin{itemize}
\item xx: Symbol
\item F: Field
\end{itemize}
\begin{itemize}
\item UP ==> UnivariatePolynomial
\item SUP ==> SparseUnivariatePolynomial
\end{itemize}

Cat ==> with
\begin{itemize}
\item interpolate: (UP(xx,F), List F, List F) -> UP(xx,F)
\item interpolate: (List F, List F) -> SUP F
\end{itemize}

Body ==> add
\begin{itemize}
\item PIA ==> PolynomialInterpolationAlgorithms
\item interpolate(qx, lx, ly) ==
\begin{itemize}
\item px := \texttt{LagrangeInterpolation}(lx, ly)$\texttt{PIA}$(F, \texttt{UP}(xx, F))
\item elt(px, qx)
\end{itemize}
\item interpolate(lx, ly) ==
\begin{itemize}
\item \texttt{LagrangeInterpolation}(lx, ly)$\texttt{PIA}$(F, SUP F)
\end{itemize}
\end{itemize}

\begin{verbatim}
— PINTERP.dotabb —

"PINTERP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PINTERP"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"PINTERP" -> "FIELD"

— PolynomialInterpolationAlgorithms.input —

)set break resume
)sys rm -f PolynomialInterpolationAlgorithms.output
)spool PolynomialInterpolationAlgorithms.output
)set message test on
\end{verbatim}
This package exports interpolation algorithms

See Also:

o )show PolynomialInterpolationAlgorithms
PolynomialInterpolationAlgorithms (PINTERPA)

Exports:
LagrangeInterpolation

— package PINTERPA PolynomialInterpolationAlgorithms —

)abbrev package PINTERPA PolynomialInterpolationAlgorithms
++ Description:
++ This package exports interpolation algorithms

PolynomialInterpolationAlgorithms(F, P): Cat == Body where
F: Field
P: UnivariatePolynomialCategory(F)

Cat ==> with
LagrangeInterpolation: (List F, List F) -> P
++ LagrangeInterpolation(l1,l2) undocumented

Body ==> add
LagrangeInterpolation(lx, ly) ==
#lx ^= #ly =>
   error "Different number of points and values."
   ip: P := 0
   for xi in lx for yi in ly for i in 0.. repeat
      pp: P := 1
      xp: F := 1
      for xj in lx for j in 0.. | i ^= j repeat
         pp := pp * (monomial(1,1) - monomial(xj,0))
         xp := xp * (xi - xj)
      ip := ip + (yi/xp) * pp
   ip
package PNTHEORY PolynomialNumberTheoryFunctions

— PolynomialNumberTheoryFunctions.input —

)set break resume
)sys rm -f PolynomialNumberTheoryFunctions.output
)spool PolynomialNumberTheoryFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialNumberTheoryFunctions
--R
--R PolynomialNumberTheoryFunctions is a package constructor
--R Abbreviation for PolynomialNumberTheoryFunctions is PNTHEORY
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PNTHEORY
--R
--R---------------------------------- Operations -----------------------------
--R bernoulli : Integer -> SparseUnivariatePolynomial(Fraction(Integer))
--R chebyshevT : Integer -> SparseUnivariatePolynomial(Integer)
--R chebyshevU : Integer -> SparseUnivariatePolynomial(Integer)
--R cyclotomic : Integer -> SparseUnivariatePolynomial(Integer)
--R euler : Integer -> SparseUnivariatePolynomial(Fraction(Integer))
--R fixedDivisor : SparseUnivariatePolynomial(Integer) -> Integer
--R hermite : Integer -> SparseUnivariatePolynomial(Integer)
--R laguerre : Integer -> SparseUnivariatePolynomial(Integer)
--R legendre : Integer -> SparseUnivariatePolynomial(Fraction(Integer))
--R
--E 1

)spool
)lisp (bye)

——

— PolynomialNumberTheoryFunctions.help —
This package provides various polynomial number theoretic functions over the integers.

See Also:
- \( \)show PolynomialNumberTheoryFunctions

---

PolynomialNumberTheoryFunctions (PNTHEORY)

Exports:
bernoulli chebyshevT chebyshevU cyclotomic euler
fixedDivisor hermite laguerre legendre

--- package PNTHEORY PolynomialNumberTheoryFunctions ---

)abbrev package PNTHEORY PolynomialNumberTheoryFunctions
++ Author: Michael Monagan, Clifton J. Williamson
++ Date Created: June 1987
++ Date Last Updated: 10 November 1996 (Claude Quitte)
++ References: Knuth, The Art of Computer Programming Vol.2
++ Description:
++ This package provides various polynomial number theoretic functions
++ over the integers.

PolynomialNumberTheoryFunctions(): Exports == Implementation where
I ==> Integer
RN ==> Fraction I
SUP ==> SparseUnivariatePolynomial
NNI ==> NonNegativeInteger
Exports == with bernoulli : I -> SUP RN
  ++ bernoulli(n) returns the nth Bernoulli polynomial \( B[n](x) \).
  ++ Bernoulli polynomials denoted \( B(n,x) \) computed by solving the
  ++ differential equation \( \text{differentiate}(B(n,x),x) = n B(n-1,x) \) where
  ++ \( B(0,x) = 1 \) and initial condition comes from \( B(n) = B(n,0) \).
chebyshevT: I -> SUP I
  ++ chebyshevT(n) returns the nth Chebyshev polynomial \( T[n](x) \).
  ++ Note that Chebyshev polynomials of the first kind,
  ++ denoted \( T(n,x) \),
  ++ computed from the two term recurrence. The generating function
  ++ \( \frac{(1-t*x)}{(1-2*t*x+t**2)} = \sum T[n](x)*t**n, n=0..infinity) \).
chebyshevU: I -> SUP I
  ++ chebyshevU(n) returns the nth Chebyshev polynomial \( U[n](x) \).
  ++ Note that Chebyshev polynomials of the second kind,
  ++ denoted \( U(n,x) \),
  ++ computed from the two term recurrence. The generating function
  ++ \( \frac{1}{(1-2*t*x+t**2)} = \sum T[n](x)*t**n, n=0..infinity) \).
cyclotomic: I -> SUP I
  ++ cyclotomic(n) returns the nth cyclotomic polynomial \( \phi[n](x) \).
  ++ Note that \( \phi[n](x) \) is the factor of \( x**n - 1 \) whose roots
  ++ are the primitive nth roots of unity.
euler : I -> SUP RN
  ++ euler(n) returns the nth Euler polynomial \( E[n](x) \).
  ++ Note that Euler polynomials denoted \( E(n,x) \) computed by solving
  ++ the differential equation
  ++ \( \text{differentiate}(E(n,x),x) = n E(n-1,x) \) where
  ++ \( E(0,x) = 1 \) and initial condition comes
  ++ from \( E(n) = 2**n E(n,1/2) \).
fixedDivisor: SUP I -> I
  ++ fixedDivisor(a) for \( a(x) \) in \( Z[x] \) is the largest integer
  ++ \( f \) such that \( f \) divides \( a(x=k) \) for all integers \( k \).
  ++ Note that fixed divisor of \( a \) is
  ++ \( \text{gcd}([a(x=k) \text{ for } k \text{ in } 0..\text{degree}(a)]) \).
hermite : I -> SUP I
  ++ hermite(n) returns the nth Hermite polynomial \( H[n](x) \).
  ++ Note that Hermite polynomials, denoted \( H[n](x) \), are computed from
  ++ the two term recurrence. The generating function is:
  ++ \( \exp(2*t*x-t**2) = \sum H[n](x)*t**n/n!, n=0..infinity) \).
laguerre : I -> SUP I
  ++ laguerre(n) returns the nth Laguerre polynomial \( L[n](x) \).
  ++ Note that Laguerre polynomials, denoted \( L[n](x) \), are computed
  ++ from the two term recurrence. The generating function is:
  ++ \( \exp(x*t/(t-1))/(1-t) = \sum L[n](x)*t**n/n!, n=0..infinity) \).
legendre : I -> SUP RN
  ++ legendre(n) returns the nth Legendre polynomial \( P[n](x) \).
  ++ Note that Legendre polynomials, denoted \( P[n](x) \), are computed
  ++ from the two term recurrence. The generating function is:
  ++ \( 1/sqrt(1-2*t*x+t**2) = \sum P[n](x)*t**n, n=0..infinity) \).
Implementation ==> add
import IntegerPrimesPackage(I)

x := monomial(1,1)$SUP(I)
y := monomial(1,1)$SUP(RN)

-- For functions computed via a fixed term recurrence we record
-- previous values so that the next value can be computed directly

E : Record(En:I, Ev:SUP(RN)) := [0,1]
B : Record( Bn:I, Bv:SUP(RN) ) := [0,1]
H : Record( Hn:I, H1:SUP(I), H2:SUP(I) ) := [0,1,x]
L : Record( Ln:I, L1:SUP(I), L2:SUP(I) ) := [0,1,x]
P : Record( Pn:I, P1:SUP(RN), P2:SUP(RN) ) := [0,1,y]
CT : Record( Tn:I, T1:SUP(I), T2:SUP(I) ) := [0,1,x]
U : Record( Un:I, U1:SUP(I), U2:SUP(I) ) := [0,1,0]

MonicQuotient: (SUP(I),SUP(I)) -> SUP(I)
MonicQuotient (a,b) ==
  leadingCoefficient(b) ^= 1 => error "divisor must be monic"
  b = 1 => a
  da := degree a
  db := degree b -- assertion: degree b > 0
  q:SUP(I) := 0
  while da >= db repeat
    t := monomial(leadingCoefficient a, (da-db)::NNI)
    a := a - b * t
    q := q + t
    da := degree a
  q

cyclotomic n ==
  +++ cyclotomic polynomial denoted phi[n](x)
p:I; q:I; r:I; s:I; m:NNI; c:SUP(I); t:SUP(I)
n < 0 => error "cyclotomic not defined for negative integers"
n = 0 => x
k := n; s := p := 1
while k > 1 repeat
  p := nextPrime p
  (q,r) := divide(k, p)
  if r = 0 then
    while r = 0 repeat (k := q; (q,r) := divide(k,p))
    t := multiplyExponents(c,p::NNI)
    c := MonicQuotient(t,c)
    s := s * p
  m := (n quo s) :: NNI
  multiplyExponents(c,m)

euler n ==
p : SUP(RN); t : SUP(RN); c : RN; s : I
n < 0 => error "euler not defined for negative integers"
if n < E.En then (s,p) := (0$I,1$SUP(RN)) else (s,p) := E
for i in s+1 .. n repeat
  t := (i::RN) * integrate p
  c := euler(i)$IntegerNumberTheoryFunctions / 2**(i::NNI) - t(i/2)
  p := t + c::SUP(RN)
E.En := n
E.Ev := p

bernoulli n ==
p : SUP RN; t : SUP RN; c : RN; s : I
n < 0 => error "bernoulli not defined for negative integers"
if n < B.Bn then (s,p) := (0$I,1$SUP(RN)) else (s,p) := B
for i in s+1 .. n repeat
  t := (i::RN) * integrate p
  c := bernoulli(i)$IntegerNumberTheoryFunctions
  p := t + c::SUP(RN)
B.Bn := n
B.Bv := p

fixedDivisor a ==
g:I; d:NNI; SUP(I)
d := degree a
for k in 1..d while g > 1 repeat g := gcd(g,a k)
g

hermite n ==
s : I; p : SUP(I); q : SUP(I)
n < 0 => error "hermite not defined for negative integers"
if n < H.Hn then (s := 0; p := 1; q := x) else (s,p,q) := H
for k in s+1 .. n repeat
  (p,q) := (2*x*p-2*(k-1)*q,p)
H.Hn := n
H.H1 := p
H.H2 := q

legendre n ==
s,I; t:; p:SUP(RN); q:SUP(RN)
n < 0 => error "legendre not defined for negative integers"
if n < P.Pn then (s := 0; p := 1; q := y) else (s,p,q) := P
for k in s+1 .. n repeat
  t := k-1
(p,q) := ((k+t)$I/k*y*p - t/k*q,p)

P.Pn := n
P.P1 := p
P.P2 := q

laguerre n ==
k:I; s:I; t:I; p:SUP(I); q:SUP(I)
n < 0 => error "laguerre not defined for negative integers"
-- (s,p,q) := if n < L.Ln then (0,1,x) else L
if n < L.Ln then (s := 0; p := 1; q := x) else (s,p,q) := L
for k in s+1 .. n repeat
  t := k-1
  (p,q) := ((((k+t)$I)::SUP(I)-x)*p-t**2*q,p)
L.Ln := n
L.L1 := p
L.L2 := q
P

chebyshevT n ==
s : I; p : SUP(I); q : SUP(I)
n < 0 => error "chebyshevT not defined for negative integers"
-- (s,p,q) := if n < CT.Tn then (0,1,x) else CT
if n < CT.Tn then (s := 0; p := 1; q := x) else (s,p,q) := CT
for k in s+1 .. n repeat (p,q) := ((2*x*p - q),p)
CT.Tn := n
CT.T1 := p
CT.T2 := q
P

chebyshevU n ==
s : I; p : SUP(I); q : SUP(I)
n < 0 => error "chebyshevU not defined for negative integers"
if n < U.Un then (s := 0; p := 1; q := 0) else (s,p,q) := U
for k in s+1 .. n repeat (p,q) := ((2*x*p - q),p)
U.Un := n
U.U1 := p
U.U2 := q
P

— PNTHEORY.dotabb —

"PNTHEORY" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PNTHEORY"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PNTHEORY" -> "PFECAT"
package POLYROOT PolynomialRoots

---

PolynomialRoots(input) ---

)set break resume
/sys rm -f PolynomialRoots.output
/spool PolynomialRoots.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialRoots

--R PolynomialRoots(E: OrderedAbelianMonoidSup,V: OrderedSet,R: IntegralDomain,P: PolynomialCategory(R,E,V),F: Field) where
--R numer : % -> P
--R denom : % -> P
--R coerce : P -> % is a package constructor

--R Abbreviation for PolynomialRoots is POLYROOT

--R This constructor is not exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for POLYROOT

--R

--- Operations -----------------------------

--R froot : (F,NonNegativeInteger) -> Record(exponent: NonNegativeInteger,coef: F,radicand: F) if R has GCDGCD
--R nthr : (P,NonNegativeInteger) -> Record(exponent: NonNegativeInteger,coef: P,radicand: List(P))
--R qroot : (Fraction(Integer),NonNegativeInteger) -> Record(exponent: NonNegativeInteger,coef: F,radicand: F)
--R rroot : (R,NonNegativeInteger) -> Record(exponent: NonNegativeInteger,coef: F,radicand: F)

--)E 1

)spool
)lisp (bye)

---

PolynomialRoots.help ---

====================================================================

PolynomialRoots examples

Computes n-th roots of quotients of multivariate polynomials
CHAPTER 17. CHAPTER P

See Also:
- )show PolynomialRoots

---

**PolynomialRoots (POLYROOT)**

Exports:
- froot
- nthr
- qroot
- rroot

— package POLYROOT PolynomialRoots —

)`abbrev package POLYROOT PolynomialRoots
++ Author: Manuel Bronstein
++ Date Created: 15 July 1988
++ Date Last Updated: 10 November 1993
++ Description:
++ Computes n-th roots of quotients of multivariate polynomials
-- not visible to the user

PolynomialRoots(E, V, R, P, F):Exports == Implementation where
E: OrderedAbelianMonoidSup
V: OrderedSet
R: IntegralDomain
P: PolynomialCategory(R, E, V)
F: Field with
  numer : $ -> P
  -- numer(x) \
  denom : $ -> P
  -- denom(x) \
  coerce: P -> $
  -- coerce(p) \undocumented
PACKAGE POLYROOT POLYNOMIALROOTS

N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Z
REC ==> Record(exponent:N, coef:F, radicand:F)

Exports ==> with
rroot: (R, N) -> REC
++ rroot(f, n) returns \spad{[m,c,r]} such
++ that \spad{f**(1/n) = c * r**(1/m)}.
qroot : (Q, N) -> REC
++ qroot(f, n) returns \spad{[m,c,r]} such
++ that \spad{f**(1/n) = c * r**(1/m)}.

if R has GcdDomain then froot: (F, N) -> REC
++ froot(f, n) returns \spad{[m,c,r]} such
++ that \spad{f**(1/n) = c * r**(1/m)}.

nthr: (P, N) -> Record(exponent:N, coef:P, radicand:List P)
++ nthr(p,n) should be local but conditional

Implementation ==> add
import FactoredFunctions Z
import FactoredFunctions P

rsplit: List P -> Record(coef:R, poly:P)
zroot : (Z, N) -> Record(exponent:N, coef:Z, radicand:Z)

zroot(x, n) ==
-- zero? x or one? x => [1, x, 1]
zero? x or (x = 1) => [1, x, 1]
s := nthRoot(squareFree x, n)
[s.exponent, s.coef, */s.radicand]

if R has imaginary: () -> R then
czroot: (Z, N) -> REC

czroot(x, n) ==
rec := zroot(x, n)
rec.exponent = 2 and rec.radicand < 0 =>
[rec.exponent, rec.coef * imaginary():P::F, (-rec.radicand)::F]
[rec.exponent, rec.coef::F, rec.radicand::F]

qroot(x, n) ==
sn := czroot(numer x, n)
sd := czroot(denom x, n)
m := lcm(sn.exponent, sd.exponent)::N
[m, sn.coef / sd.coef,
 (sn.radicand ** (m quo sn.exponent)) /
 (sd.radicand ** (m quo sd.exponent))]

else
qroot(x, n) ==
sn := zroot(numer x, n)
sd := zroot(denom x, n)
m := lcm(sn.exponent, sd.exponent)::N
[m, sn.coef::F / sd.coef::F,
 (sn.radicand ** (m quo sn.exponent))::F /
 (sd.radicand ** (m quo sd.exponent))::F]

if R has RetractableTo Fraction Z then
rroot(x, n) ==
 (r := retractIfCan(x)@Union(Fraction Z,"failed")) case "failed"
 => [n, 1, x::P::F]
qroot(r::Q, n)
else
if R has RetractableTo Z then
rroot(x, n) ==
 (r := retractIfCan(x)@Union(Z,"failed")) case "failed"
 => [n, 1, x::P::F]
qroot(r::Z::Q, n)
else
rroot(x, n) == [n, 1, x::P::F]

rsplit l ==
r := 1$R
p := 1$P
for q in l repeat
 if (u := retractIfCan(q)@Union(R, "failed")) case "failed"
 then p := p * q
 else r := r * u::R
 [r, p]

if R has GcdDomain then
if R has RetractableTo Z then
nthr(x, n) ==
 (r := retractIfCan(x)@Union(Z,"failed")) case "failed"
 => nthRoot(squareFree x, n)
 rec := zroot(r::Z, n)
 [rec.exponent, rec.coef::P, [rec.radicand::P]]
else nthr(x, n) == nthRoot(squareFree x, n)

froot(x, n) ==
 -- zero? x or one? x => [1, x, 1]
 zero? x or (x = 1) => [1, x, 1]
 sn := nthr(numer x, n)
 sd := nthr(denom x, n)
 pn := rsplit(sn.radicand)
 pd := rsplit(sd.radicand)
 rn := rroot(pn.coef, sn.exponent)
 rd := rroot(pd.coef, sd.exponent)
m := lcm([rn.exponent, rd.exponent, sn.exponent, sd.exponent])::N
[m, (sn.coef::F / sd.coef::F) * (rn.coef / rd.coef),]
((rm.radicand ** (m quo rn.exponent)) / 
  (rd.radicand ** (m quo rd.exponent))) * 
  (pn.poly ** (m quo sn.exponent))::F / 
  (pd.poly ** (m quo sd.exponent))::F)

POLYROOT.dotabb

"POLYROOT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLYROOT"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLYROOT" -> "PFECAT"

package PSETPK PolynomialSetUtilitiesPackage

--- PolynomialSetUtilitiesPackage.input ---

)set break resume
)sys rm -f PolynomialSetUtilitiesPackage.output
)spool PolynomialSetUtilitiesPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialSetUtilitiesPackage
--R
--R PolynomialSetUtilitiesPackage(R: IntegralDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategory(R,E,V)) is a package constructor
--R Abbreviation for PolynomialSetUtilitiesPackage is PSETPK
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PSETPK
--R
--R------------------------------- Operations --------------------------------
--R bivariate? : P -> Boolean
--R interReduce : List(P) -> List(P)
--R univariate? : P -> Boolean
--R bivariatePolynomials : List(P) -> Record(goodPols: List(P),badPols: List(P))
--R certainlySubVariety? : (List(P),List(P)) -> Boolean
--R irreducibleFactors : List(P) -> List(P) if R has CHARZ and R has EUCDOM
--R lazyIrreducibleFactors : List(P) -> List(P) if R has CHARZ and R has EUCDOM
--R linearPolynomials : List(P) -> Record(goodPols: List(P),badPols: List(P))
This package provides modest routines for polynomial system solving. The aim of many of the operations of this package is to remove certain factors in some polynomials in order to avoid unnecessary computations in algorithms involving splitting techniques by partial factorization.

See Also:
- )show PolynomialSetUtilitiesPackage
PolynomialSetUtilitiesPackage (PSETPK)

Exports:
- bivariate?
- interReduce
- univariate?
- certainlySubVariety?
- lazyIrreducibleFactors
- possiblyNewVariety?
- quasiMonicPolynomials
- removeRedundantFactors
- removeRedundantFactorsInPols
- removeRoughlyRedundantFactorsInPol
- removeSquaredIfCan
- rewriteSetByReducingWithParticularGenerators
- selectAndPolynomials
- selectPolynomials
- univariatePolynomials
- unprotectedRemoveRedundantFactors
- crushedSet
- linear?
- bivariatePolynomials
- irreducibleFactors
- linearPolynomials
- probablyZeroDim?
- removeIrreducibleRedundantFactors
- removeRedundantFactorsInContents
- removeRoughlyRedundantFactorsInPols
- rewriteIdealWithQuasiMonicGenerators
- roughlyBasicSet
- selectOrPolynomials
- squareFreeFactors
- univariatePolynomialsGcds

— package PSETPK PolynomialSetUtilitiesPackage —

)abbrev package PSETPK PolynomialSetUtilitiesPackage
++ Author: Marc Moreno Maza (marc@nag.co.uk)
++ Date Created: 12/01/1995
++ Date Last Updated: 12/15/1998
++ Description:
++ This package provides modest routines for polynomial system solving.
++ The aim of many of the operations of this package is to remove certain
++ factors in some polynomials in order to avoid unnecessary computations
++ in algorithms involving splitting techniques by partial factorization.
PolynomialSetUtilitiesPackage (R,E,V,P) : Exports == Implementation where

R : IntegralDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LP ==> List P
FP ==> Factored P
T ==> GeneralTriangularSet(R,E,V,P)
RRZ ==> Record(factor: P,exponent: Integer)
RBT ==> Record(bas:T,top:LP)
RUL ==> Record(chs:Union(T,"failed"),rfs:LP)
GPS ==> GeneralPolynomialSet(R,E,V,P)
pf ==> MultivariateFactorize(V, E, R, P)

Exports == with

removeRedundantFactors: LP -> LP
++ \texttt{removeRedundantFactors(lp)} returns \texttt{lq} such that if
++ \texttt{lp = \{p1,...,pn\}} and \texttt{lq = \{q1,...,qm\}}
++ then the product \texttt{p1*p2*...*pn} vanishes iff the product \texttt{q1*q2*...*qm} vanishes,
++ and the product of degrees of the \texttt{qi} is not greater than
++ the one of the \texttt{pj}, and no polynomial in \texttt{lq}
++ divides another polynomial in \texttt{lq}. In particular,
++ polynomials lying in the base ring \texttt{R} are removed.
++ Moreover, \texttt{lq} is sorted \texttt{w.r.t. infRittWu?}.
++ Furthermore, if \texttt{R} is gcd-domain, the polynomials in \texttt{lq} are
++ pairwise without common non trivial factor.
removeRedundantFactors: (P,P) -> LP
++ \texttt{removeRedundantFactors(p,q)} returns the same as
++ \texttt{removeRedundantFactors(\{p,q\})}

removeSquaresIfCan : LP -> LP
++ \texttt{removeSquaresIfCan(lp)} returns
++ \texttt{removeDuplicates [squareFreePart(p)$P for p in lp\}}
++ if \texttt{R} is gcd-domain else returns \texttt{lp}.

unprotectedRemoveRedundantFactors: (P,P) -> LP
++ \texttt{unprotectedRemoveRedundantFactors(p,q)} returns the same as
++ \texttt{removeRedundantFactors(p,q)} but does assume that neither
++ \texttt{p} nor \texttt{q} lie in the base ring \texttt{R} and assumes that
++ \texttt{infRittWu?(p,q)} holds. Moreover, if \texttt{R} is gcd-domain,
++ then \texttt{p} and \texttt{q} are assumed to be square free.

removeRedundantFactors: (LP,P) -> LP
++ \texttt{removeRedundantFactors(lp,q)} returns the same as
++ \texttt{removeReduced(cons(q,lp))} assuming
++ that \texttt{lp} returns \texttt{lp} up to replacing some polynomial \texttt{pj} in \texttt{lp}
++ by some some polynomial \texttt{qj} associated to \texttt{pj}.
removeRedundantFactors : (LP,LP) -> LP
++ \axiom{removeRedundantFactors(lp,lq)} returns the same as
++ \axiom{removeRedundantFactors(concat(lp,lq))} assuming
++ that \axiom{removeRedundantFactors(lp)} returns \axiom{lp}
++ up to replacing some polynomial \axiom{pj} in \axiom{lp}
++ by some polynomial \axiom{qj} associated to \axiom{pj}.
removeRedundantFactors : (LP,LP,(LP -> LP)) -> LP
++ \axiom{removeRedundantFactors(lp,lq,remOp)} returns the same as
++ \axiom{concat(remOp(removeRoughlyRedundantFactorsInPols(lp,lq)),lq)}
++ assuming that \axiom{remOp(lp)} returns \axiom{lp} up to similarity.
certainlySubVariety? : (LP,LP) -> B
++ \axiom{certainlySubVariety?(newlp,lp)} returns true iff for every \axiom{p}
++ in \axiom{lp} the remainder of \axiom{p} by \axiom{newlp} using the division algorithm
++ of Groebner techniques is zero.
possiblyNewVariety? : (LP, List LP) -> B
++ \axiom{possiblyNewVariety?(newlp,llp)} returns true iff for every \axiom{lp}
++ in \axiom{llp} certainlySubVariety?(newlp,lp) does not hold.
probablyZeroDim?: LP -> B
++ \axiom{probablyZeroDim?(lp)} returns true iff the number of polynomials
++ in \axiom{lp} is not smaller than the number of variables occurring
++ in these polynomials.
selectPolynomials : ((P -> B),LP) -> Record(goodPols:LP,badPols:LP)
++ \axiom{selectPolynomials(pred?,ps)} returns \axiom{gps,bps} where
++ \axiom{gps} is a list of the polynomial \axiom{p} in \axiom{ps}
++ such that \axiom{pred?(p)} holds and \axiom{bps} are the other ones.
selectOrPolynomials : (List (P -> B),LP) -> Record(goodPols:LP,badPols:LP)
++ \axiom{selectOrPolynomials(lpred?,ps)} returns \axiom{gps,bps} where
++ \axiom{gps} is a list of the polynomial \axiom{p} in \axiom{ps}
++ such that \axiom{pred?(p)} holds for some \axiom{pred?} in \axiom{lpred?}
++ and \axiom{bps} are the other ones.
selectAndPolynomials : (List (P -> B),LP) -> Record(goodPols:LP,badPols:LP)
++ \axiom{selectAndPolynomials(lpred?,ps)} returns \axiom{gps,bps} where
++ \axiom{gps} is a list of the polynomial \axiom{p} in \axiom{ps}
++ such that \axiom{pred?(p)} holds for every \axiom{pred?} in \axiom{lpred?}
++ and \axiom{bps} are the other ones.
quasiMonicPolynomials : LP -> Record(goodPols:LP,badPols:LP)
++ \axiom{quasiMonicPolynomials(lp)} returns \axiom{qmps,nqmps} where
++ \axiom{qmps} is a list of the quasi-monic polynomials in \axiom{lp}
++ and \axiom{nqmps} are the other ones.
univariate? : P -> B
++ \axiom{univariate?(p)} returns true iff \axiom{p} involves one and
++ only one variable.
univariatePolynomials : LP -> Record(goodPols:LP,badPols:LP)
++ \axiom{univariatePolynomials(lp)} returns \axiom{ups,nups} where
++ \axiom{ups} is a list of the univariate polynomials,
++ and \axiom{nups} are the other ones.
linear? : P -> B
++ \axiom{linear?(p)} returns true iff \axiom{p} does not lie
++ in the base ring \axiom{R} and has main degree \axiom{1}.
linearPolynomials : LP -> Record(goodPols:LP,badPols:LP)
CHAPTER 17. CHAPTER P

++ \texttt{linearPolynomials(lp)} returns \texttt{\{lps, nlps\}} where
++ \texttt{lps} is a list of the linear polynomials in \texttt{lp},
++ and \texttt{\{nlps\}} are the other ones.

\textbf{bivariate? : P -> B}
++ \texttt{bivariate?(p)} returns true iff \texttt{\{p\}} involves two and
++ only two variables.

\textbf{bivariatePolynomials : LP -> Record(goodPols:LP, badPols:LP)}
++ \texttt{\{bivariatePolynomials(lp)\}} returns \texttt{\{bps, nbps\}} where
++ \texttt{\{bps\}} is a list of the bivariate polynomials,
++ and \texttt{\{nbps\}} are the other ones.

\textbf{removeRoughlyRedundantFactorsInPols : (LP, LP) -> LP}
++ \texttt{\{removeRoughlyRedundantFactorsInPols(lp, lf)\}} returns
++ \texttt{\{newlp\}} where \texttt{\{newlp\}} is obtained from \texttt{\{lp\}}
++ by removing in every polynomial \texttt{\{p\}} of \texttt{\{lp\}}
++ any occurrence of a polynomial \texttt{\{f\}} in \texttt{\{lf\}}.
++ This may involve a lot of exact-quotients computations.

\textbf{removeRoughlyRedundantFactorsInPols : (LP, LP, B) -> LP}
++ \texttt{\{removeRoughlyRedundantFactorsInPols(lp, lf, opt)\}} returns
++ the same as \texttt{\{removeRoughlyRedundantFactorsInPols(lp, lf)\}}
++ if \texttt{\{opt\}} is \texttt{\{false\}} and if the previous operation
++ does not return any non null and constant polynomial,
++ else return \texttt{\{[1]\}}.

\textbf{removeRoughlyRedundantFactorsInPol : (P, LP) -> P}
++ \texttt{\{removeRoughlyRedundantFactorsInPol(p, lf)\}} returns the same as
++ \texttt{\{removeRoughlyRedundantFactorsInPols([p], lf, true)\}}

\textbf{intReduce : LP -> LP}
++ \texttt{\{intReduce(lp)\}} returns \texttt{\{lq\}} such that \texttt{\{lp\}}
++ and \texttt{\{lq\}} generate the same ideal and no polynomial
++ in \texttt{\{lq\}} is reducible by the others in the sense
++ of Groebner bases. Since no assumptions are required
++ the result may depend on the ordering the reductions are
++ performed.

\textbf{roughBasicSet : LP -> Union(Record(bas:T, top:LP), "failed")}
++ \texttt{\{roughBasicSet(lp)\}} returns the smallest (with Ritt-Wu
++ ordering) triangular set contained in \texttt{\{lp\}}.

\textbf{crushedSet : LP -> LP}
++ \texttt{\{crushedSet(lp)\}} returns \texttt{\{lq\}} such that \texttt{\{lp\}} and
++ \texttt{\{lq\}} generate the same ideal and no rough basic
++ sets reduce (in the sense of Groebner bases) the other
++ polynomials in \texttt{\{lq\}}.

\textbf{rewriteSetByReducingWithParticularGenerators : (LP, (P -> B), ((P,P) -> B), ((P,P) -> P)) -> LP}
++ \texttt{\{rewriteSetByReducingWithParticularGenerators(lp, pred?, redOp?, redOp)\}}
++ returns \texttt{\{lq\}} where \texttt{\{lq\}} is computed by the following
++ algorithm. Chose a basic set w.r.t. the reduction-test \texttt{\{redOp?\}}
++ among the polynomials satisfying property \texttt{\{pred?\}},
++ if it is empty then leave, else reduce the other polynomials by
++ this basic set w.r.t. the reduction-operation \texttt{\{redOp\}}.
++ Repeat while another basic set with smaller rank can be computed.
++ See code. If \texttt{\{pred?\}} is \texttt{\{quasiMonic?\}} the ideal is unchanged.

\textbf{rewriteIdealWithQuasiMonicGenerators : (LP, (P,P) -> B), ((P,P) -> P)) -> LP}
++ \texttt{rewriteIdealWithQuasiMonicGenerators(lp,redOp?,redOp)} returns
++ \texttt{lq} where \texttt{lq} and \texttt{lp} generate
++ the same ideal in \texttt{"R"^(-1) P} and \texttt{lp}
++ has rank not higher than the one of \texttt{lp}.
++ Moreover, \texttt{lq} is computed by reducing \texttt{lp}
++ w.r.t. some basic set of the ideal generated by
++ the quasi-monic polynomials in \texttt{lp}.

if \texttt{R} has \texttt{GcdDomain}
then

\texttt{squareFreeFactors : P \rightarrow LP}
++ \texttt{squareFreeFactors(p)} returns the square-free factors of \texttt{p}
++ over \texttt{R}

\texttt{univariatePolynomialsGcds : LP \rightarrow LP}
++ \texttt{univariatePolynomialsGcds(lp)} returns \texttt{lg} where
++ \texttt{lg} is a list of the gcds of every pair in \texttt{lp}
++ of univariate polynomials in the same main variable.

\texttt{univariatePolynomialsGcds : (LP,B) \rightarrow LP}
++ \texttt{univariatePolynomialsGcds(lp,opt)} returns the same as
++ \texttt{univariatePolynomialsGcds(lp)} if \texttt{opt} is
++ \texttt{false} and if the previous operation does not return
++ any non null and constant polynomial, else return \texttt{axiom\{1\}}.

\texttt{removeRoughlyRedundantFactorsInContents : (LP, LP) \rightarrow LP}
++ \texttt{removeRoughlyRedundantFactorsInContents(lp,lf)} returns
++ \texttt{newlp} where \texttt{newlp} is obtained from \texttt{lp}
++ by removing in the content of every polynomial of \texttt{lp}
++ any occurrence of a polynomial \texttt{f} in \texttt{lf}. Moreover,
++ squares over \texttt{R} are first removed in the content
++ of every polynomial of \texttt{lp}.

\texttt{removeRedundantFactorsInContents : (LP, LP) \rightarrow LP}
++ \texttt{removeRedundantFactorsInContents(lp,lf)} returns \texttt{newlp}
++ where \texttt{newlp} is obtained from \texttt{lp} by removing
++ in the content of every polynomial of \texttt{lp} any non trivial
++ factor of any polynomial \texttt{f} in \texttt{lf}. Moreover,
++ squares over \texttt{R} are first removed in the content
++ of every polynomial of \texttt{lp}.

\texttt{removeRedundantFactorsInPols : (LP, LP) \rightarrow LP}
++ \texttt{removeRedundantFactorsInPols(lp,lf)} returns \texttt{newlp}
++ where \texttt{newlp} is obtained from \texttt{lp} by removing
++ in every polynomial \texttt{p} of \texttt{lp} any non trivial
++ factor of any polynomial \texttt{f} in \texttt{lf}. Moreover,
++ squares over \texttt{R} are first removed in every
++ polynomial \texttt{lp}.

if (\texttt{R} has \texttt{EuclideanDomain}) and (\texttt{R} has \texttt{CharacteristicZero})
then

\texttt{irreducibleFactors : LP \rightarrow LP}
++ \texttt{irreducibleFactors(lp)} returns \texttt{lf} such that if
++ \texttt{lp} = \texttt{[p_1,...,p_n]} and \texttt{lf} = \texttt{[f_1,...,f_m]} then
++ \texttt{p_1*p_2*...*p_n=0} means \texttt{f_1*f_2*...*f_m=0}, and the \texttt{\{f_i\}}
++ are irreducible over \texttt{R} and are pairwise distinct.

\texttt{lazyIrreducibleFactors : LP \rightarrow LP}
CHAPTER 17. CHAPTER P

++ \texttt{lazyIrreducibleFactors(lp)} returns \texttt{lf} such that if
++ \texttt{lp = [p1,...,pn]} and \texttt{lf = [f1,...,fm]} then
++ \texttt{p1*p2*...*pn=0} means \texttt{f1*f2*...*fm=0}, and the \texttt{fi}
++ are irreducible over \texttt{R} and are pairwise distinct.
++ The algorithm tries to avoid factorization into irreducible
++ factors as far as possible and makes previously use of gcd
++ techniques over \texttt{R}.

++ \texttt{removeIrreducibleRedundantFactors(lp, lq)} returns the same
++ as \texttt{irreducibleFactors(concat(lp, lq))} assuming
++ that \texttt{irreducibleFactors(lp)} returns \texttt{lp}
++ up to replacing some polynomial \texttt{pj} in \texttt{lp}
++ by some polynomial \texttt{qj} associated to \texttt{pj}.

\texttt{Implementation == add}

autoRemainder: T -> List(P)

removeAssociates (lp:LP):LP ==
    removeDuplicates [primPartElseUnitCanonical(p) for p in lp]

selectPolynomials (pred?, ps) ==
    gps : LP := []
    bps : LP := []
    while not empty? ps repeat
        p := first ps
        ps := rest ps
        if pred?(p)
            then
                gps := cons(p, gps)
            else
                bps := cons(p, bps)
    gps := sort(infRittWu?, gps)
    bps := sort(infRittWu?, bps)
    [gps, bps]

selectOrPolynomials (lpred?, ps) ==
    gps : LP := []
    bps : LP := []
    while not empty? ps repeat
        p := first ps
        ps := rest ps
        clpred? := lpred?
        while (not empty? clpred?) and (not (first clpred?)(p)) repeat
            clpred? := rest clpred?
        if not empty?(clpred?)
            then
                gps := cons(p, gps)
            else
                bps := cons(p, bps)
gps := sort(infRittWu?,gps)
bps := sort(infRittWu?,bps)

[gps,bps]

selectAndPolynomials (lpred?,ps) ==
gps : LP := []
bps : LP := []
while not empty? ps repeat
    p := first ps
    ps := rest ps
    clpred? := lpred?
    while (not empty? clpred?) and ((first clpred?)(p)) repeat
        clpred? := rest clpred?
    if empty?(clpred?)
        then
            gps := cons(p,gps)
        else
            bps := cons(p,bps)
gps := sort(infRittWu?,gps)
bps := sort(infRittWu?,bps)
[gps,bps]

linear? p ==
    ground? p => false
-- one?(mdeg(p))
    (mdeg(p) = 1)

linearPolynomials ps ==
    selectPolynomials(linear?,ps)

univariate? p ==
    ground? p => false
    not(ground?(init(p))) => false
    tp := tail(p)
    ground?(tp) => true
    not (mvar(p) = mvar(tp)) => false
    univariate?(tp)

univariatePolynomials ps ==
    selectPolynomials(univariate?,ps)

bivariate? p ==
    ground? p => false
    ground?(tail(p)) => univariate?(init(p))
    vp := mvar(p)
    vtp := mvar(tail(p))
    ((ground? init(p)) and (vp = vtp)) => bivariate? tail(p)
    ((ground? init(p)) and (vp > vtp)) => univariate? tail(p)
    not univariate?(init(p)) => false
    vip := mvar(init(p))
vip > vtp => false
vip = vtp => univariate? tail(p)
vtw < vtw => false
zero? degree(tail(p),vip) => univariate? tail(p)
bivariate? tail(p)

bivariatePolynomials ps ==
    selectPolynomials(bivariate?,ps)

quasiMonicPolynomials ps ==
    selectPolynomials(quasiMonic?,ps)

removeRoughlyRedundantFactorsInPols (lp,lf,opt) ==
    empty? lp => lp
    newlp : LP := []
    stop : B := false
    lp := remove(zero?,lp)
    lf := sort(infRittWu?,lf)
    test : Union(P,"failed")
    while (not empty? lp) and (not stop) repeat
        p := first lp
        lp := rest lp
        copylf := lf
        while (not empty? copylf) and (not ground? p) and (not (mvar(p) < mvar(first copylf))) repeat
            f := first copylf
            copylf := rest copylf
        while (((test := p exquo$P f)) case P) repeat
            p := test::P
        stop := opt and ground?(p)
        newlp := cons(unitCanonical(p),newlp)
    stop => [1$P]
    newlp

removeRoughlyRedundantFactorsInPol(p,lf) ==
    zero? p => p
    lp : LP := [p]
    first removeRoughlyRedundantFactorsInPols (lp,lf,true()$B)

removeRoughlyRedundantFactorsInPols (lp,lf) ==
    removeRoughlyRedundantFactorsInPols (lp,lf,false()$B)

possiblyNewVariety?(newlp,llp) ==
    while (not empty? llp) and _
    (not certainlySubVariety?(newlp,first(llp))) repeat
        llp := rest llp
    empty? llp

certainlySubVariety?(lp,lq) ==
    gs := construct(lp)$GPS
    while (not empty? lq) and _
(zero? (remainder(first(lq),gs)$GPS).polnum) repeat
  lq := rest lq
empty? lq

probablyZeroDim?(lp: List P) : Boolean ==
  m := #lp
  lv : List V := variables(first lp)
while not empty? (lp := rest lp) repeat
  lv := concat(variables(first lp),lv)
n := #(removeDuplicates lv)
not (n > m)

interReduce(lp: LP): LP ==
  ps := lp
  rs: List(P) := []
repeat
  empty? ps => return rs
  ps := sort(supRittWu?, ps)
  p := first ps
  ps := rest ps
  r := remainder(p,[ps]$GPS).polnum
  zero? r => "leave"
  ground? r => return []
  associates?(r,p) => rs := cons(r,rs)
  ps := concat(ps,cons(r,rs))
  rs := []

  ground? p => false
  ground? q => true
  mvar(p) > mvar(q)

roughBasicSet(lp) == basicSet(lp,roughRed?)$T

autoRemainder(ts:T): List(P) ==
  empty? ts => members(ts)
  lp := sort(infRittWu?, reverse members(ts))
newlp : List(P) := [primPartElseUnitCanonical first(lp)]
  lp := rest(lp)
while not empty? lp repeat
  p := (remainder(first(lp),construct(newlp)$GPS)$GPS).polnum
  if not zero? p then
    if ground? p then
      newlp := [1$P]
      lp := []
    else
      newlp := cons(p,newlp)
      lp := rest(lp)
else
lp := rest(lp)
newlp

crushedSet(lp) ==
rec := roughBasicSet(lp)
contradiction := (rec case "failed")@B
finished := B := false
while (not finished) and (not contradiction) repeat
bs := (rec::RBT).bas
rs := (rec::RBT).top
rs := rewriteIdealWithRemainder(rs,bs)$T
-- contradiction := ((not empty? rs) and (one? first(rs)))  
contradiction := ((not empty? rs) and (first(rs) = 1))
if not contradiction then
rs := concat(rs,autoRemainder(bs))
rec := roughBasicSet(rs)
contradiction := (rec case "failed")@B
not contradiction => finished := not infRittWu?((rec::RBT).bas,bs)
contradiction => [1$P]
rs

rewriteSetByReducingWithParticularGenerators (ps,pred?,redOp?,redOp) ==
rs : LP := remove(zero?,ps)
any?(ground?,rs) => [1$P]
contradiction := B := false
bs1 : T := empty()$T
rec : Union(RBT,"failed")
ar : Union(T,List(P))
stop := B := false
while (not contradiction) and (not stop) repeat
rec := basicSet(rs,pred?,redOp?)$T
bs2 : T := (rec::RBT).bas
rs := (rec::RBT).top
-- ar := autoReduce(bs2,lazyPrem,reduced?)@Union(T,List(P))
ar := bs2::Union(T,List(P))
if (ar case T)@B then
bs2 := ar::T
if infRittWu?(bs2,bs1) then
rs := rewriteSetWithReduction(rs,bs2,redOp,redOp?)$T
bs1 := bs2
else
stop := true
rs := concat(members(bs2),rs)
else
rs := concat(ar::LP,rs)
if any?(ground?,rs)
then
  contradiction := true
  rs := [1$P]
rs

removeRedundantFactors (lp:LP,lq :LP, remOp : (LP -> LP)) ==
  -- ASSUME remOp(lp) returns lp up to similarity
  lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
lq := remOp lq
  sort(infRittWu?,concat(lp,lq))

removeRedundantFactors (lp:LP,lq :LP) ==
  lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
lq := removeRedundantFactors lq
  sort(infRittWu?,concat(lp,lq))

if (R has EuclideanDomain) and (R has CharacteristicZero) then
  irreducibleFactors lp ==
    newlp : LP := []
lrrz : List RRZ
  rrz : RRZ
fp : FP
    while not empty? lp repeat
      p := first lp
      lp := rest lp
      fp := factor(p)$pf
      lrrz := factors(fp)$FP
      lf := remove(ground?,[rrz.factor for rrz in lrrz])
      newlp := concat(lf,newlp)
    removeDuplicates newlp
  lazyIrreducibleFactors lp ==
    lp := removeRedundantFactors(lp)
    newlp : LP := []
lrrz : List RRZ
  rrz : RRZ
fp : FP
    while not empty? lp repeat
      p := first lp
      lp := rest lp
      fp := factor(p)$pf
      lrrz := factors(fp)$FP
      lf := remove(ground?,[rrz.factor for rrz in lrrz])
      newlp := concat(lf,newlp)
    newlp

removeIrreducibleRedundantFactors (lp:LP,lq :LP) ==
  -- ASSUME lp only contains irreducible factors over R
  lq := removeRoughlyRedundantFactorsInPols(lq,lp,false)
lq := irreducibleFactors lq
sort(infRittWu?,concat(lp,lq))

if R has GcdDomain
then

squareFreeFactors(p:P) ==
sfp: Factored P := squareFree(p)$P
lsf: List P := [foo.factor for foo in factors(sfp)]
    lsf

univariatePolynomialsGcds (ps,opt) ==
lg : LP := []
pInV : LP
stop : B := false
ps := sort(infRittWu?,ps)
p,g : P
v : V
while (not empty? ps) and (not stop) repeat
    while (not empty? ps) and (not univariate?((p := first(ps)))) repeat
        ps := rest ps
    if not empty? ps
    then
        v := mvar(p)$P
        pInV := [p]
        while (not empty? ps) and (mvar((p := first(ps))) = v) repeat
            if (univariate?(p))
            then
                pInV := cons(p,pInV)
                ps := rest ps
                g := gcd(pInV)$P
                stop := opt and (ground? g)
                lg := cons(g,lg)
                stop => [1$P]
        lg

univariatePolynomialsGcds ps ==
    univariatePolynomialsGcds (ps,false)

removeSquaresIfCan lp ==
empty? lp => lp
    removeDuplicates [squareFreePart(p)$P for p in lp]

rewriteIdealWithQuasiMonicGenerators (ps,redOp?,redOp) ==
ups := removeSquaresIfCan(univariatePolynomialsGcds(ps,true))
ps := removeDuplicates concat(ups,ps)
rewriteSetByReducingWithParticularGenerators(ps,quasiMonic?,redOp?,redOp)

removeRoughlyRedundantFactorsInContents (ps,lf) ==
empty? ps => ps
newps : LP := []
p,newp,cp,newcp,f,g : P
test : Union(P,"failed")
copylf : LP
while not empty? ps repeat
    p := first ps
    ps := rest ps
    cp := mainContent(p)$P
    newcp := squareFreePart(cp)$P
    newp := (p exquo$P cp)::P
    if not ground? newp then
        copylf := [f for f in lf | mvar(f) <= mvar(newcp)]
        while (not empty? copylf) and (not ground? newcp) repeat
            f := first copylf
            copylf := rest copylf
            test := (newcp exquo$P f)
            if (test case P)$B then
                newcp := test::P
            if ground? newcp then
                newp := unitCanonical(newp)
            else
                newp := unitCanonical(newp * newcp)
        newps := cons(newp,newps)
    newps

removeRedundantFactorsInContents (ps,lf) ==
    empty? ps => ps
    newps : LP := []
p,newp,cp,newcp,f,g : P
while not empty? ps repeat
    p := first ps
    ps := rest ps
    cp := mainContent(p)$P
    newcp := squareFreePart(cp)$P
    newp := (p exquo$P cp)::P
    if not ground? newcp then
        copylf := lf
        while (not empty? copylf) and (not ground? newcp) repeat
            f := first copylf
            copylf := rest copylf
            g := gcd(newcp,f)$P
            if not ground? g then
                newcp := (newcp exquo$P g)::P
            if ground? newcp then
                newp := (newp exquo$P g)::P
    if ground? newcp then
        newp := (newp exquo$P g)::P
newp := unitCanonical(newp)  
else  
  newp := unitCanonical(newp * newcp)  
newps := cons(newp,newps)  
newps

removeRedundantFactorsInPols (ps,lf) ==  
  empty? ps => ps  
  newps : LP := []  
  p,newp,cp,newcp,f,g : P  
  while not empty? ps repeat  
    p := first ps  
    ps := rest ps  
    cp := mainContent(p)$P  
    newcp := squareFreePart(cp)$P  
    newp := (p exquo$P cp)::P  
    newp := squareFreePart(newp)$P  
    copylf := lf  
    while not empty? copylf repeat  
      f := first copylf  
      copylf := rest copylf  
      if not ground? newcp  
        then  
          g := gcd(newcp,f)$P  
          if not ground? g  
            then  
              newcp := (newcp exquo$P g)::P  
      if not ground? newp  
        then  
          g := gcd(newp,f)$P  
          if not ground? g  
            then  
              newp := (newp exquo$P g)::P  
      if ground? newcp  
        then  
          newp := unitCanonical(newp)  
        else  
          newp := unitCanonical(newp * newcp)  
          newps := cons(newp,newps)  
          newps

removeRedundantFactors (a:P,b:P) : LP ==  
  a := primPartElseUnitCanonical(squareFreePart(a))  
  b := primPartElseUnitCanonical(squareFreePart(b))  
  if not infRittWu?(a,b)  
    then  
      (a,b) := (b,a)  
    if ground? a  
      then  
        if ground? b
then
    return([])
else
    return([b])
else
    if ground? b
        then
            return([a])
        else
            return(removeRedundantFactors(a,b))

unprotectedRemoveRedundantFactors (a,b) ==
c := b exquo$P a
if (c case P)@B
    then
d : P := c::P
    if ground? d
        then
            return([a])
        else
            return([a,d])
    else
g : P := gcd(a,b)$P
    if ground? g
        then
            return([a,b])
        else
            return([g,(a exquo$P g)::P,(b exquo$P g)::P])
else
removeSquaresIfCan lp ==
lp

rewriteIdealWithQuasiMonicGenerators (ps,redOp?,redOp) ==
    rewriteSetByReducingWithParticularGenerators(ps,quasiMonic?,redOp?,redOp)

removeRedundantFactors (a:P,b:P) ==
a := primPartElseUnitCanonical(a)
b := primPartElseUnitCanonical(b)
if not infRittWu?(a,b)
    then
        (a,b) := (b,a)
if ground? a
    then
        if ground? b
            then
                return([])
            else
                return([b])
else
  if ground? b
    then
      return([a])
  else
    return(unprotectedRemoveRedundantFactors(a,b))

 unprotectedRemoveRedundantFactors (a,b) ==
  c := b exquo$P a
  if (c case P)@B
    then
      d : P := c::P
      if ground? d
        then
          return([a])
        else
          if infRittWu?(d,a) then (a,d) := (d,a)
          return(unprotectedRemoveRedundantFactors(a,d))
    else
      return([a,b])

 removeRedundantFactors (lp:LP) ==
  lp := remove(ground?, lp)
  lp := removeDuplicates [primPartElseUnitCanonical(p) for p in lp]
  lp := removeSquaresIfCan lp
  lp := removeDuplicates [unitCanonical(p) for p in lp]
  empty? lp => lp
  size?(lp,1$N)$(List P) => lp
  lp := sort(infRittWu?,lp)
  p : P := first lp
  lp := rest lp
  base : LP := unprotectedRemoveRedundantFactors(p,first lp)
  top : LP := rest lp
  while not empty? top repeat
    p := first top
    base := removeRedundantFactors(base,p)
    top := rest top
  base

 removeRedundantFactors (lp:LP,a:P) ==
  lp := remove(ground?, lp)
  lp := sort(infRittWu?,lp)
  ground? a => lp
  empty? lp => [a]
  toSee : LP := lp
  toSave : LP := []
  while not empty? toSee repeat
    b := first toSee
    toSee := rest toSee
    if not infRittWu?(b,a)
then
  (c,d) := (a,b)
else
  (c,d) := (b,a)
rrf := unprotectedRemoveRedundantFactors(c,d)
empty? rrf => error"in removeRedundantFactors : (LP,P) -> LP from PSETPK"
c := first rrf
rrf := rest rrf
if empty? rrf
  then
    if associates?(c,b)
    then
      toSave := concat(toSave,toSee)
a := b
toSee := []
else
  a := c
toSee := concat(toSave,toSee)
toSave := []
else
d := first rrf
rrf := rest rrf
if empty? rrf
  then
    if associates?(c,b)
    then
      toSave := concat(toSave,[b])
a := d
else
  if associates?(d,b)
  then
toSave := concat(toSave,[b])
a := c
else
toSave := removeRedundantFactors(toSave,c)
a := d
else
e := first rrf
not empty? rest(rrf) => error"in removeRedundantFactors:(LP,P)-->LP from PSETPK"
-- ASSUME that neither c, nor d, nor e may be associated to b
toSave := removeRedundantFactors(toSave,c)
toSave := removeRedundantFactors(toSave,d)
a := e
if empty? toSee
  then
toSave := sort(infRittWu?,cons(a,toSave))
package SOLVEFOR PolynomialSolveByFormulas

-- PolynomialSolveByFormulas.input --

)set break resume
/sys rm -f PolynomialSolveByFormulas.output
/spool PolynomialSolveByFormulas.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialSolveByFormulas
--R
--R PolynomialSolveByFormulas(UP: UnivariatePolynomialCategory(F),F)where
--R F: Field
--R *** : (%,Fraction(Integer)) -> % is a package constructor
--R Abbreviation for PolynomialSolveByFormulas is SOLVEFOR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SOLVEFOR
--R
--R----------------------------------- Operations -----------------------------------
--R aCubic : (F,F,F,F) -> F           aLinear : (F,F) -> F
--R aQuadratic : (F,F,F) -> F         aQuadratic : (F,F,F,F,F) -> F
--R cubic : UP -> List(F)             cubic : (F,F,F,F) -> List(F)
--R linear : UP -> List(F)            linear : (F,F) -> List(F)
--R particularSolution : UP -> F      quadratic : UP -> List(F)
--R quadratic : (F,F,F) -> List(F)    quartic : UP -> List(F)
--R quartic : (F,F,F,F,F) -> List(F)  solve : UP -> List(F)
--R mapSolve : (UP,(F -> F)) -> Record(solns: List(F),maps: List(Record(arg: F,res: F)))
--R
--E 1

)spool
)lisp (bye)
This package factors the formulas out of the general solve code, allowing their recursive use over different domains. Care is taken to introduce few radicals so that radical extension domains can more easily simplify the results.

See Also:

- )show PolynomialSolveByFormulas

---

Exports:

- aCubic
- aLinear
- aQuadratic
- aQuartic
- cubic
- linear
- particularSolution
- quadratic
- quartic
- solve
- mapSolve

---

)abbrev package SOLVEFOR PolynomialSolveByFormulas
++ Author: SMW June 86, BMT Sept 93
++ Description:
++ This package factors the formulas out of the general solve code,
++ allowing their recursive use over different domains.
++ Care is taken to introduce few radicals so that radical extension
++ domains can more easily simplify the results.
PolynomialSolveByFormulas(UP, F): PSFcat == PSFdef where

    UP: UnivariatePolynomialCategory F
    F: Field with "**": (%, Fraction Integer) -> %

    L => List

    PSFcat == with
        solve: UP -> L F
        ++ solve(u) \undocumented
        particularSolution: UP -> F
        ++ particularSolution(u) \undocumented
        mapSolve: (UP, F -> F) -> Record(solns: L F,
            maps: L Record(arg:F,res:F))
        ++ mapSolve(u,f) \undocumented
        linear: UP -> L F
        ++ linear(u) \undocumented
        quadratic: UP -> L F
        ++ quadratic(u) \undocumented
        cubic: UP -> L F
        ++ cubic(u) \undocumented
        quartic: UP -> L F
        ++ quartic(u) \undocumented

        -- Arguments give coefs from high to low degree.
        linear: (F, F) -> L F
        ++ linear(f,g) \undocumented
        quadratic: (F, F, F) -> L F
        ++ quadratic(f,g,h) \undocumented
        cubic: (F, F, F) -> L F
        ++ cubic(f,g,h,i) \undocumented
        quartic: (F, F, F, F) -> L F
        ++ quartic(f,g,h,i,j) \undocumented

        aLinear: (F, F) -> F
        ++ aLinear(f,g) \undocumented
        aQuadratic: (F, F, F) -> F
        ++ aQuadratic(f,g,h) \undocumented
        aCubic: (F, F, F) -> F
        ++ aCubic(f,g,h,j) \undocumented
        aQuartic: (F, F, F, F) -> F
        ++ aQuartic(f,g,h,i,k) \undocumented

    PSFdef == add
        id ==> (IDENTITY$Lisp)
maplist: List Record(arg: F, res: F) := []
mapSolving?: Boolean := false
-- map: F -> F := id #1 replaced with line below
map: Boolean := false

mapSolve(p, fn) ==
  -- map := fn #1 replaced with line below
  locmap: F -> F := x +-> fn x; map := id locmap
  mapSolving? := true; maplist := []
  slist := solve p
  mapSolving? := false;
  -- map := id #1 replaced with line below
  locmap := x +-> id x; map := id locmap
  [slist, maplist]

part(s: F): F ==
  not mapSolving? => s
  -- t := map s replaced with line below
  t: F := SPADCALL(s, map)$Lisp
  t = s => s
  maplist := cons([t, s], maplist)
  t

-----------------------------------------------------------------
-- Entry points and error handling
-----------------------------------------------------------------
cc ==> coefficient

-- local intsolve
intsolve(u:UP):L(F) ==
  u := (factors squareFree u).1.factor
  n := degree u
  n=1 => linear (cc(u,1), cc(u,0))
  n=2 => quadratic (cc(u,2), cc(u,1), cc(u,0))
  n=3 => cubic (cc(u,3), cc(u,2), cc(u,1), cc(u,0))
  n=4 => quartic (cc(u,4), cc(u,3), cc(u,2), cc(u,1), cc(u,0))
  error "All sqfr factors of polynomial must be of degree < 5"

solve u ==
  ls := nil$L(F)
  for f in factors squareFree u repeat
    lsf := intsolve f.factor
    for i in 1..(f.exponent) repeat ls := [:lsf,:ls]
    ls

particularSolution u ==
  u := (factors squareFree u).1.factor
  n := degree u
  n=1 => aLinear (cc(u,1), cc(u,0))
n=2 => aQuadratic (cc(u,2), cc(u,1), cc(u,0))
n=3 => aCubic (cc(u,3), cc(u,2), cc(u,1), cc(u,0))
n=4 => aQuartic (cc(u,4), cc(u,3), cc(u,2), cc(u,1), cc(u,0))
error "All sqfr factors of polynomial must be of degree < 5"

needDegree(n: Integer, u: UP): Boolean ==
  degree u = n => true
  error concat("Polynomial must be of degree ", n::String)

needLcoef(cn: F): Boolean ==
  cn ^= 0 => true
  error "Leading coefficient must not be 0."

needChar0(): Boolean ==
  characteristic()$F = 0 => true
  error "Formula defined only for fields of characteristic 0."

linear u ==
  needDegree(1, u)
  linear (coefficient(u,1), coefficient(u,0))

quadratic u ==
  needDegree(2, u)
  quadratic (coefficient(u,2), coefficient(u,1), coefficient(u,0))

cubic u ==
  needDegree(3, u)
  cubic (coefficient(u,3), coefficient(u,2), coefficient(u,1), coefficient(u,0))

quartic u ==
  needDegree(4, u)
  quartic (coefficient(u,4), coefficient(u,3), coefficient(u,2), coefficient(u,1), coefficient(u,0))

-- The formulas
---------------------------------------------------------

-- local function for testing equality of radicals.
-- This function is necessary to detect at least some of the
-- situations like sqrt(9)-3 = 0 --> false.
equ(x:F,y:F):Boolean ==
  ( (recip(x-y)) case "failed" ) => true
  false

linear(c1, c0) ==
  needLcoef c1
  [- c0/c1 ]
aLinear(c1, c0) ==
    first linear(c1, c0)

quadratic(c2, c1, c0) ==
    needLcoef c2; needChar0()
    (c0 = 0) => [0$F,:linear(c2, c1)]
    (c1 = 0) => [(-c0/c2)**(1/2),(-c0/c2)**(1/2)]
    D := part(c1**2 - 4*c2*c0)**(1/2)
    [(-c1+D)/(2*c2), (-c1-D)/(2*c2)]

aQuadratic(c2, c1, c0) ==
    needLcoef c2; needChar0()
    (c0 = 0) => 0$F
    (c1 = 0) => (-c0/c2)**(1/2)
    D := part(c1**2 - 4*c2*c0)**(1/2)
    (-c1+D)/(2*c2)

w3: F := (-1 + (-3::F)**(1/2)) / 2::F

cubic(c3, c2, c1, c0) ==
    needLcoef c3; needChar0()
    -- case one root = 0, not necessary but keeps result small
    (c0 = 0) => [0$F,:quadratic(c3, c2, c1)]
    a1 := c2/c3; a2 := c1/c3; a3 := c0/c3
    -- case x**3-a3 = 0, not necessary but keeps result small
    (a1 = 0 and a2 = 0) =>
        [ u*(-a3)**(1/3) for u in [1, w3, w3**2 ] ]
    -- case x**3 + a1*x**2 + a1**2*x/3 + a3 = 0, the general for-
    -- mula is not valid in this case, but solution is easy.
    P := part(-a1/3::F)
    equ(a1**2,3*a2) =>
        S := part((- a3 + (a1**3)/27::F)**(1/3))
        [ P + S*u for u in [1,w3,w3**2] ]
    -- general case
    Q := part((3*a2 - a1**2)/9::F)
    R := part((9*a1*a2 - 27*a3 - 2*a1**3)/54::F)
    D := part(Q**3 + R**2)**(1/2)
    S := part(R + D)**(1/3)
    -- S = 0 is done in the previous case
    [ P + S*u - Q/(S*u) for u in [1,w3,w3**2] ]

aCubic(c3, c2, c1, c0) ==
    needLcoef c3; needChar0()
    (c0 = 0) => 0$F
    a1 := c2/c3; a2 := c1/c3; a3 := c0/c3
\[(a_1 = 0 \text{ and } a_2 = 0) \Rightarrow (-a_3)^{1/3}\]

\[P := \text{part}(-a_1/3:\mathbb{F})\]

\[\text{equ}(a_1^2, 3a_2) \Rightarrow\]

\[S := \text{part}((-a_3 + (a_1^3)/27::\mathbb{F})^{1/3})\]

\[P + S\]

\[Q := \text{part}((3a_2 - a_1^2)/9::\mathbb{F})\]

\[R := \text{part}((9a_1a_2 - 27a_3 - 2a_1^3)/54::\mathbb{F})\]

\[D := \text{part}(Q^{*3} + R^{*2})^{1/2}\]

\[S := \text{part}(R + D)^{1/3}\]

\[P + S - Q/S\]

\[\text{quartic}(c_4, c_3, c_2, c_1, c_0) ==\]

\[\text{needLcoeff}(c_4); \text{needChar0}()\]

\[\text{-- case one root }= 0, \text{ not necessary but keeps result small}\]

\[\text{-- Make monic:}\]

\[a_1 := c_3/c_4; a_2 := c_2/c_4; a_3 := c_1/c_4; a_4 := c_0/c_4\]

\[\text{-- case }x^4 + a_4 = 0 \Leftrightarrow (x^2 + \sqrt{-a_4})*(x^2 - \sqrt{-a_4})\]

\[\text{-- not necessary but keeps result small.}\]

\[\text{-- Translate }w = x + a_1/4 \text{ to eliminate }a_1: \ w^4 + p*w^2 + q*w + r\]

\[p := \text{part}(a_2 - 3a_1^2/8::\mathbb{F})\]

\[q := \text{part}(a_3 - a_1a_2/2::\mathbb{F} + a_1^3/8::\mathbb{F})\]

\[r := \text{part}(a_4 - a_1a_3/4::\mathbb{F} + a_1^2a_2/16::\mathbb{F} - 3a_1^4/256::\mathbb{F})\]

\[\text{-- The roots of the translated polynomial are those of}\]

\[\text{-- two quadratics. (What about rt=0 ?)\}

\[\text{-- rt=0 can be avoided by picking a root }\neq p \text{ of the cubic}\]

\[\text{-- polynomial above. This is always possible provided that}\]

\[\text{-- the input is squarefree. In this case the two other roots}\]

\[\text{-- are }+(-) 2r^{**}(1/2).\]

\[\text{if equ(q,0) }\text{ -- this means p is a root}\]

\[\text{then t0 := part}(2*(r^{**}(1/2))\]

\[\text{else t0 := acubic}(1, -p, -4*r, 4*p*r - q^{**2})\]

\[\text{rt := part}(t0 - p^{**}(1/2))\]

\[\text{slist := append( quadratic}(1, rt, (-q/rt + t0)/2::\mathbb{F} ),\]

\[\text{quadratic}(1, -rt, (q/rt + t0)/2::\mathbb{F}))\]

\[\text{-- Translate back:}\]

\[\text{[s - a1/4::\mathbb{F} for s in slist]}\]

\[\text{aQuartic}(c_4, c_3, c_2, c_1, c_0) ==\]

\[\text{needLcoeff}(c_4); \text{needChar0}()\]

\[\text{(c0 = 0) }\Rightarrow 0\mathbb{F}\]

\[a_1 := c_3/c_4; a_2 := c_2/c_4; a_3 := c_1/c_4; a_4 := c_0/c_4\]

\[(a_1 = 0 \text{ and } a_2 = 0 \text{ and } a_3 = 0) \Rightarrow (-a_4)^{1/4}\]
p := part(a2-3*a1*a1/8::F)
q := part(a3-a1*a2/2::F + a1**2*a1/8::F)
r := part(a4-a1*a3/4::F + a1**2*a2/16::F - 3*a1**4/256::F)
if equ(q,0)
  then t0 := part(2*(r**(1/2)))
  else t0 := aCubic(1, -p, -4*r, 4*p*r - q**2)
rt := part(t0 - p)**(1/2)
s := aQuadratic( 1, rt, (-q/rt + t0)/2::F )
s - a1/4::F

—— SOLVEFOR.dotabb ——

"SOLVEFOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVEFOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SOLVEFOR" -> "PFECAT"

—— package PSQFR PolynomialSquareFree ——

package PSQFR PolynomialSquareFree

— PolynomialSquareFree.input —

set break resume
sys rm -f PolynomialSquareFree.output
spool PolynomialSquareFree.output
set message test on
set message auto off
clear all
--S 1 of 1
show PolynomialSquareFree
--R
--R PolynomialSquareFree(VarSet: OrderedSet,E: OrderedAbelianMonoidSup,RC: GcdDomain,P: PolynomialCategory(RC,E,VarSet)) is a package constructor
--R Abbreviation for PolynomialSquareFree is PSQFR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PSQFR
--R
--R----------------------------------- Operations -----------------------------------
--R squareFree : P -> Factored(P)
--R
--E 1
This package computes square-free decomposition of multivariate polynomials over a coefficient ring which is an arbitrary gcd domain. The requirement on the coefficient domain guarantees that the content can be removed so that factors will be primitive as well as square-free. Over an infinite ring of finite characteristic, it may not be possible to guarantee that the factors are square-free.

See Also:
o )show PolynomialSquareFree

Exports:
squareFree

---

PolynomialSquareFree (PSQFR)

Exports:
squareFree

---

)abbrev package PSQFR PolynomialSquareFree
++ Date Last Updated: November 1993, (P.Gianni)
++ Description:
++ This package computes square-free decomposition of multivariate
++ polynomials over a coefficient ring which is an arbitrary gcd domain.
++ The requirement on the coefficient domain guarantees that the
++ \spadfun{content} can be
++ removed so that factors will be primitive as well as square-free.
++ Over an infinite ring of finite characteristic, it may not be possible to
++ guarantee that the factors are square-free.

PolynomialSquareFree(VarSet:OrderedSet,E,RC:GcdDomain,P):C == T where
  E:OrderedAbelianMonoidSup
  P:PolynomialCategory(RC,E,VarSet)

C == with
  squareFree : P -> Factored P
    ++ squareFree(p) returns the square-free factorization of the
    ++ polynomial p. Each factor has no repeated roots, and the
    ++ factors are pairwise relatively prime.

T == add
  SUP    ==> SparseUnivariatePolynomial(P)
  NNI    ==> NonNegativeInteger
  fUnion ==> Union("nil", "sqfr", "irred", "prime")
  FF     ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

finSqFr : (P,List VarSet) -> Factored P
pthPower : P -> Factored P
pPolRoot : P -> P
putPth   : P -> P

chrc:=characteristic$RC

if RC has CharacteristicNonZero then
  -- find the p-th root of a polynomial
  pPolRoot(f:P) : P ==
    lvar:=variables f
    empty? lvar => f
    mv:=first lvar
    uf:=univariate(f,mv)
    uf:=divideExponents(uf,chrc)::SUP
    uf:=map(pPolRoot,uf)
    multivariate(uf,mv)

  -- substitute variables with their p-th power
  putPth(f:P) : P ==
    lvar:=variables f
    empty? lvar => f
    mv:=first lvar
    uf:=univariate(f,mv)
    uf:=multiplyExponents(uf,chrc)::SUP
pf:=map(putPth,uf)
multivariate(uf,mv)

-- the polynomial is a perfect power
pthPower(f:P) : Factored P ==
  proot : P := 0
  isSq : Boolean := false
  if (g:=charthRoot f) case "failed" then proot:=pPolRoot(f)
  else
    proot := g :: P
    isSq := true
  psqfr:=finSqFr(proot,variables f)
  isSq =>
    makeFR((unit psqfr)**chrc,[[u.flg,u.fctr,
      (u.xpnt)*chrc] for u in factorList psqfr])
  makeFR((unit psqfr),[["nil",putPth u.fctr,u.xpnt]
    for u in factorList psqfr])

-- compute the square free decomposition, finite characteristic case
finSqFr(f:P,lvar:List VarSet) : Factored P ==
  empty? lvar => pthPower(f)
  mv:=first lvar
  lvar:=lvar.rest
  differentiate(f,mv)=0 => finSqFr(f,lvar)
  uf:=univariate(f,mv)
  cont := content uf
  cont1:P:=1
  uf := (uf exquo cont)::SUP
  squf := squareFree(uf)$UnivariatePolynomialSquareFree(P,SUP)
pfaglist:List FF :=[]
  for u in factorList squf repeat
    uexp:NNI:=(u.xpnt):NNI
    u.flg = "sqfr" => -- the square free factor is OK
      pfaglist:= cons([u.flg,multivariate(u.fctr,mv),uexp],
        pfaglist)
    --listfin1:= finSqFr(multivariate(u.fctr,mv),lvar)
    listfin1:= squareFree multivariate(u.fctr,mv)
    flistfin1=[[[uu.flg,uu.fctr,uu.xpnt*uexp]
      for uu in factorList listfin1]
      cont1:=cont1*((unit listfin1)**uexp)
    pfaglist:=append(flistfin1,pfaglist)
    cont:=cont*cont1
    cont := 1 =>
      sqp := squareFree cont
      pfaglist:= append (factorList sqp,pfaglist)
      makeFR(unit(sqp)*coefficient(unit squf,0),pfaglist)
      makeFR(coefficient(unit squf,0),pfaglist)
  makeFR(unit(sqp)*coefficient(unit squf,0),pfaglist)

squareFree(p:P) ==
  mv:=mainVariable p
mv case "failed" => makeFR(p,[])$Factored(P)
characteristic$RC ^=0 => finSqFr(p,variables p)
up:=univariate(p,mv)
cont := content up
up := (up exquo cont)::SUP
squp := squareFree(up)$UnivariatePolynomialSquareFree(P,SUP)
pfaclist:List FF :=[
[u.flg,multivariate(u.fctr,mv),u.xpnt]
for u in factorList squp]
cont ^= 1 =>
   sqp := squareFree cont
   makeFR(unit(sqp)*coefficient(unit squp,0),
       append(factorList sqp, pfaclist))
   makeFR(coefficient(unit squp,0),pfaclist)

---

--- PSQFR.dotabb ---

"PSQFR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PSQFR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PSQFR" -> "PFECAT"

---

package POLY2UP PolynomialToUnivariatePolynomial

--- PolynomialToUnivariatePolynomial.input ---

)set break resume
)sys rm -f PolynomialToUnivariatePolynomial.output
)spool PolynomialToUnivariatePolynomial.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PolynomialToUnivariatePolynomial
--R
--R PolynomialToUnivariatePolynomial(x: Symbol,R: Ring) is a package constructor
--R Abbreviation for PolynomialToUnivariatePolynomial is POLY2UP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for POLY2UP
--R
This package is primarily to help the interpreter do coercions. It allows you to view a polynomial as a univariate polynomial in one of its variables with coefficients which are again a polynomial in all the other variables.

See Also:
- \show PolynomialToUnivariatePolynomial

---

**Exports:**
- univariate

---

**Poly2UnivariatePolynomial (POLY2UP)**

---

**package POLY2UP PolynomialToUnivariatePolynomial**
\begin{verbatim}
)abbrev package POLY2UP PolynomialToUnivariatePolynomial
++ Description:
++ This package is primarily to help the interpreter do coercions.
++ It allows you to view a polynomial as a
++ univariate polynomial in one of its variables with
++ coefficients which are again a polynomial in all the
++ other variables.

PolynomialToUnivariatePolynomial(x:Symbol, R:Ring): with
    univariate: (Polynomial R, Variable x) ->
        UnivariatePolynomial(x, Polynomial R)
        ++ \texttt{univariate(p, x)} converts the polynomial \texttt{p} to a one of type
        ++ \spad{UnivariatePolynomial(x,Polynomial(R))}, \texttt{ie. as a member of} \spad{R[[...]][x]}.

== add
    \texttt{univariate(p, y) ==}
    q:SparseUnivariatePolynomial(Polynomial R) := \texttt{univariate(p, x)}
    \texttt{map(x1+->x1, q}$\texttt{Poly2UnivariatePolynomialCategoryFunctions2(Polynomial R,}
    \texttt{SparseUnivariatePolynomial Polynomial R, Polynomial R,}
    \texttt{UnivariatePolynomial(x, Polynomial R))}

POLY2UP.dotabb

"POLY2UP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=POLY2UP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLY2UP" -> "PFECAT"

package LIMITPS PowerSeriesLimitPackage

-- PowerSeriesLimitPackage.input --

)set break resume
)sys rm -f PowerSeriesLimitPackage.output
)spool PowerSeriesLimitPackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show PowerSeriesLimitPackage
--R
\end{verbatim}
--R PowerSeriesLimitPackage(R: Join(GcdDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),FE: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(R))) is a package constructor
--R Abbreviation for PowerSeriesLimitPackage is LIMITPS
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LIMITPS
--R
--R------------------------------- Operations --------------------------------
--R complexLimit : (FE,Equation(OnePointCompletion(FE))) -> Union(OnePointCompletion(FE),"failed")
--R limit : (FE,Equation(OrderedCompletion(FE))) -> Union(OrderedCompletion(FE),Record(leftHandLimit: Union(OrderedCompletion(FE),"failed"),rightHandLimit: Union(OrderedCompletion(FE),"failed")),"failed")
--R limit : (FE,Equation(FE),String) -> Union(OrderedCompletion(FE),"failed")
--R
--E 1

)spool
)lisp (bye)

——
—— PowerSeriesLimitPackage.help ——

=================================== PowerSeriesLimitPackage examples ===================================
===================================

PowerSeriesLimitPackage implements limits of expressions
in one or more variables as one of the variables approaches a
limiting value. Included are two-sided limits, left- and right-
hand limits, and limits at plus or minus infinity.

See Also:
o )show PowerSeriesLimitPackage

——

PowerSeriesLimitPackage (LIMITPS)
Exports:
   complexLimit  limit

   package LIMITPS PowerSeriesLimitPackage

)abbrev package LIMITPS PowerSeriesLimitPackage
++ Author: Clifton J. Williamson
++ Date Created: 21 March 1989
++ Date Last Updated: 30 March 1994
++ Description:
++ PowerSeriesLimitPackage implements limits of expressions
++ in one or more variables as one of the variables approaches a
++ limiting value. Included are two-sided limits, left- and right-
++ hand limits, and limits at plus or minus infinity.

PowerSeriesLimitPackage(R,FE): Exports == Implementation where
   R : Join(GcdDomain,OrderedSet,RetractableTo Integer, _
         LinearlyExplicitRingOver Integer)
   FE : Join(AlgebraicallyClosedField,TranscendentalFunctionCategory, _
            FunctionSpace R)
   Z  ==> Integer
   RN ==> Fraction Integer
   RF ==> Fraction Polynomial R
   OFE ==> OrderedCompletion FE
   OPF ==> OnePointCompletion FE
   SY  ==> Symbol
   EQ  ==> Equation
   LF  ==> LiouvillianFunction
   UTS ==> UnivariateTaylorSeries
   ULS ==> UnivariateLaurentSeries
   UPXS ==> UnivariatePuiseuxSeries
   EFULS ==> ElementaryFunctionsUnivariateTaylorSeries
   EFUPXS ==> ElementaryFunctionsUnivariatePuiseuxSeries
   FS2UPS ==> FunctionSpaceToUnivariatePowerSeries
   FS2EXPXP ==> FunctionSpaceToExponentialExpansion
   Problem ==> Record(func::String,prob::String)
   RESULT ==> Union(OFE,"failed")
   TwoSide ==> Record(leftHandLimit:RESULT,rightHandLimit:RESULT)
   U   ==> Union(OFE,TwoSide,"failed")
   SIGNEF ==> ElementaryFunctionSign(R,FE)

Exports ==> with
   limit: (FE,EQ OFE) -> U
   ++ limit(f(x),x = a) computes the real limit \spad{lim(x \to a,f(x))}.

   complexLimit: (FE,EQ OPF) -> Union(OPF, "failed")
   ++ complexLimit(f(x),x = a) computes the complex limit
++ \spad{\lim(x \to a,f(x))}.

\textbf{limit}: (FE,EQ FE,String) -> RESULT
++ \textit{limit(f(x),x=a,"left")} computes the left hand real limit
++ \spad{\lim(x \to a-,f(x))};
++ \spad{\lim(f(x),x=a,"right")} computes the right hand real limit
++ \spad{\lim(x \to a+,f(x))}.

\textbf{Implementation} ==> add
import ToolsForSign(R)
import ElementaryFunctionStructurePackage(R,FE)

zeroFE:FE := 0
anyRootsOrAtrigs?: FE -> Boolean
complLimit: (FE,SY) -> Union(OPF,"failed")
okProblem?: (String,String) -> Boolean
realLimit: (FE,SY) -> U
xxpLimit: (FE,SY) -> RESULT
limitPlus: (FE,SY) -> RESULT
localsubst: (FE,Kernel FE,Z,FE) -> FE
locallimit: (FE,SY,OFE) -> U
locallimitcomplex: (FE,SY,OPF) -> Union(OPF,"failed")
poleLimit:(RN,FE,SY) -> U
poleLimitPlus:(RN,FE,SY) -> RESULT

noX?:(FE,SY) -> Boolean
noX?(fcn,x) == not member?(x,variables fcn)

constant?:: FE -> Boolean
constant? fcn == empty? variables fcn

firstNonLogPtr: (FE,SY) -> List Kernel FE
firstNonLogPtr(fcn,x) ==
  -- returns a pointer to the first element of kernels(fcn) which
  -- has 'x' as a variable, which is not a logarithm, and which is
  -- not simply 'x'
  list := kernels fcn
  while not empty? list repeat
    ker := first list
    not is?(ker,"log" :: Symbol) and member?(x,variables(ker::FE)) _
    and not(x = name(ker)) =>
      return list
    list := rest list
  empty()

finiteValueAtInfinity?:: Kernel FE -> Boolean
finiteValueAtInfinity? ker ==
is?(ker,"erf" :: Symbol) => true
is?(ker,"sech" :: Symbol) => true
is?(ker,"csch" :: Symbol) => true
is?(ker, "tanh" :: Symbol) => true
is?(ker, "coth" :: Symbol) => true
is?(ker, "atan" :: Symbol) => true
is?(ker, "acot" :: Symbol) => true
is?(ker, "asec" :: Symbol) => true
is?(ker, "acsc" :: Symbol) => true
is?(ker, "acsch" :: Symbol) => true
is?(ker, "acoth" :: Symbol) => true
is?(ker, "fresnelS" :: Symbol) => true
is?(ker, "fresnelC" :: Symbol) => true
error "finiteValueAtInfinity? true, but unknown value at infinity"

knownValueAtInfinity?: Kernel FE -> Boolean
knownValueAtInfinity? ker ==
  is?(ker, "exp" :: Symbol) => true
  is?(ker, "sinh" :: Symbol) => true
  is?(ker, "cosh" :: Symbol) => true
false

leftOrRight: (FE, SY, FE) -> SingleInteger
leftOrRight(fcn, x, limVal) ==
  -- function is called when limitPlus(fcn, x) = limVal
  -- determines whether the limiting value is approached
  -- from the left or from the right
  (value := limitPlus(inv(fcn - limVal), x)) case "failed" => 0
  (inf := whatInfinity(val := value :: OFE)) = 0 =>
    error "limit package: internal error"
  inf

specialLimit1: (FE, SY) -> RESULT
specialLimitKernel: (Kernel FE, SY) -> RESULT
specialLimitNormalize: (FE, SY) -> RESULT
specialLimit: (FE, SY) -> RESULT
specialLimit(fcn, x) ==
xkers := [k for k in kernels fcn | member?(x, variables(k::FE))]
#xkers = 1 => specialLimit1(fcn, x)
num := numerator fcn
den := denominator fcn
for k in xkers repeat
  (fval := limitPlus(k::FE, x)) case "failed" =>
    return specialLimitNormalize(fcn, x)
  whatInfinity(val := fval::OFE) ^= 0 =>
    return specialLimitNormalize(fcn, x)
  valu := retractIfCan(val)@Union(FE, "failed")
  finVal := valu :: FE
  num := eval(num, k, finVal)
den := eval(den, k, finVal)
den = 0 => return specialLimitNormalize(fcn, x)
specialLimitNormalize(fcn,x) == -- tries to normalize result first
   nfcn := normalize(fcn)
   fcn ^= nfcn => limitPlus(nfcn,x)
   xkers := [k for k in tower fcn | member?(x,variables(k::FE))]
   # xkers ^= 2 => "failed"
   expKers := [k for k in xkers | is?(k, "exp" :: Symbol)]
   # expKers ^= 1 => "failed"
   -- fcn is a rational function of x and exp(g(x)) for some rational function g
   expKer := first expKers
   (fval := limitPlus(expKer::FE,x)) case "failed" => "failed"
   vv := new()$SY; eq := EQ FE := equation(expKer :: FE,vv :: FE)
   cc := eval(fcn,eq)
   expKerLim := fval :: OFE
   -- following test for "failed" is needed due to compiler bug
   -- limVal case OFE generates EQCAR(limVal, 1) which fails on atom "failed"
   (limVal := locallimit(cc,vv,expKerLim)) case "failed" => "failed"
   limVal case OFE =>
      limm := limVal :: OFE
      (lim := retractIfCan(limm)@Union(FE,"failed")) case "failed" =>
         "failed" -- need special handling for directions at infinity
         limitPlus(lim, x)
         "failed"

   -- limit of expression having only 1 kernel involving x
specialLimit1(fcn,x) ==
   -- find the first interesting kernel in tower(fcn)
   xkers := [k for k in kernels fcn | member?(x,variables(k::FE))]
   #xkers ^= 1 => "failed"
   ker := first xkers
   vv := new()$SY; eq := EQ FE := equation(ker :: FE,vv :: FE)
   cc := eval(fcn,eq)
   member?(x,variables cc) => "failed" -- error "No argument"
   (lim := specialLimitKernel(ker, x)) case "failed" => lim
   argLim : OFE := lim :: OFE
   (limVal := locallimit(cc,vv,expKerLim)) case "failed" => "failed"
   limVal case OFE => limVal :: OFE
   "failed"

   -- limit of single kernel involving x
specialLimitKernel(ker,x) ==
   is?(ker,"log" :: Symbol) =>
      args := argument ker
      empty? args => "failed" -- error "No argument"
      not empty? rest args => "failed" -- error "Too many arguments"
      arg := first args
      -- compute limit(x -> 0+,arg)
      (limm := limitPlus(arg,x)) case "failed" => "failed"
      lim := limm :: OFE
(inf := whatInfinity lim) = -1 => "failed"
argLim : OFE :=
   -- log(+infinity) = +infinity
   inf = 1 => lim
   -- now 'lim' must be finite
   (li := retractIfCan(lim)@Union(FE,"failed") :: FE) = 0 =>
      -- log(0) = -infinity
      leftOrRight(arg,x,0) = 1 => minusInfinity()
      return "failed"
   log(li) :: OFE
   -- kernel should be a function of one argument f(arg)
   args := argument(ker)
   empty? args => "failed" -- error "No argument"
   not empty? rest args => "failed" -- error "Too many arguments"
   arg := first args
   -- compute limit(x -> 0+,arg)
   (limm := limitPlus(arg,x)) case "failed" => "failed"
   lim := limm :: OFE
   f := elt(operator ker,(var := new()$SY) :: FE)
   -- compute limit(x -> 0+,f(arg))
   -- case where 'lim' is finite
   (inf := whatInfinity lim) = 0 =>
      is?(ker,"erf" :: Symbol) => erf(retract(lim)@FE)$LF(R,FE) :: OFE
      (kerValue := locallimit(f,var,lim)) case "failed" => "failed"
      kerValue case OFE => kerValue :: OFE
      "failed"
   -- case where 'lim' is plus infinity
   inf = 1 =>
      finiteValueAtInfinity? ker =>
      val : FE :=
      is?(ker,"erf" :: Symbol) => 1
      is?(ker,"sech" :: Symbol) => 0
      is?(ker,"csch" :: Symbol) => 0
      is?(ker,"tanh" :: Symbol) => 0
      is?(ker,"coth" :: Symbol) => 0
      is?(ker,"atan" :: Symbol) => pi()/(2 :: FE)
      is?(ker,"acot" :: Symbol) => 0
      is?(ker,"asec" :: Symbol) => pi()/(2 :: FE)
      is?(ker,"acsc" :: Symbol) => 0
      is?(ker,"acsch" :: Symbol) => 0
      is?(ker,"fresnelS" :: Symbol) => -sqrt(pi()/(8::FE))
      is?(ker,"fresnelC" :: Symbol) => -sqrt(pi()/(8::FE))
      error "finiteValueAtInfinity? true, but unknown value at infinity"
   -- ker must be acoth
   0
      val :: OFE
   knownValueAtInfinity? ker =>
      lim -- limit(exp, cosh, sinh ,x=inf) = inf
      "failed"
   -- case where 'lim' is minus infinity
finiteValueAtInfinity? ker =>
  val : FE :=
  is?(ker,"erf" :: Symbol) => -1
  is?(ker,"sech" :: Symbol) => 0
  is?(ker,"csch" :: Symbol) => 0
  is?(ker,"tanh" :: Symbol) => 0
  is?(ker,"coth" :: Symbol) => 0
  is?(ker,"atan" :: Symbol) => -pi()/2 :: FE
  is?(ker,"acot" :: Symbol) => pi()
  is?(ker,"asec" :: Symbol) => -pi()/2 :: FE
  is?(ker,"acsc" :: Symbol) => -pi()
  is?(ker,"acsch" :: Symbol) => 0
  -- ker must be acoth
  0
val :: OFE

knownValueAtInfinity? ker =>
  is?(ker,"exp" :: Symbol) => (0@FE) :: OFE
  is?(ker,"sinh" :: Symbol) => lim
  is?(ker,"cosh" :: Symbol) => plusInfinity()
  "failed"
  "failed"

logOnlyLimit: (FE,SY) -> RESULT
logOnlyLimit(coef,x) ==
  -- this function is called when the 'constant' coefficient involves
  -- the variable 'x'. Its purpose is to compute a right hand limit
  -- of an expression involving log x. Here log x is replaced by -1/v,
  -- where v is a new variable. If the new expression no longer involves
  -- x, then take the right hand limit as v -> 0+
  vv := new()$SY
  eq : EQ FE := equation(log(x :: FE),-inv(vv :: FE))
  member?(x,variables(cc := eval(coef,eq))) => "failed"
  limitPlus(cc,vv)

locallimit(fcn,x,a) ==
  -- Here 'fcn' is a function f(x) = f(x,...) in 'x' and possibly
  -- other variables, and 'a' is a limiting value. The function
  -- computes \( \lim(x \to a, f(x)) \).
  xK := retract(x::FE)@Kernel(FE)
  (n := whatInfinity a) = 0 =>
    realLimit(localsubst(fcn,xK,1,retract(a)@FE),x)
  (u := limitPlus(eval(fcn,xK,n * inv(xK::FE)),x))
    case "failed" => "failed"
  u::OFE

localsubst(fcn, k, n, a) ==
  a = 0 and n = 1 => fcn
  eval(fcn,k,n * (k::FE) + a)

locallimitcomplex(fcn,x,a) ==
xK := retract(x::FE)@Kernel(FE)
(g := retractIfCan(a)@Union(FE,"failed")) case FE =>
complLimit(localsubst(fcn,xK,1,g::FE),x)
complLimit(eval(fcn,xK,inv(xK::FE)),x)

limit(fcn,eq,str) ==
(xx := retractIfCan(lhs eq)@Union(SY,"failed")) case "failed" =>
  error "limit:left hand side must be a variable"
x := xx :: SY; a := rhs eq
xK := retract(x::FE)@Kernel(FE)
limitPlus(localsubst(fcn,xK,direction str,a),x)

anyRootsOrAtrigs? fcn ==
  -- determines if 'fcn' has any kernels which are roots
  -- or if 'fcn' has any kernels which are inverse trig functions
  -- which could produce series expansions with fractional exponents
  for kernel in tower fcn repeat
    is?(kernel,"nthRoot" :: Symbol) => return true
    is?(kernel,"asin" :: Symbol) => return true
    is?(kernel,"acos" :: Symbol) => return true
    is?(kernel,"asec" :: Symbol) => return true
    is?(kernel,"acsc" :: Symbol) => return true
  false

complLimit(fcn,x) ==
  -- computes lim(x -> 0,fcn) using a Puiseux expansion of fcn,
  -- if fcn is an expression involving roots, and using a Laurent
  -- expansion of fcn otherwise
  lim : FE :=
  anyRootsOrAtrigs? fcn =>
    ppack := FS2UPS(R,FE,RN,_,
    UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),_,
    EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
    pseries := exprToUPS(fcn,false,"complex")$ppack
    pseries case %problem => return "failed"
    if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
    pole? upxs => return infinity()
    coefficient(upxs,0)
    lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_,
    EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
    lseries := exprToUPS(fcn,false,"complex")$lpack
    lseries case %problem => return "failed"
    if pole?(uls := lseries.%series) then uls := map(normalize,uls)
    pole? uls => return infinity()
    coefficient(uls,0)
  -- can the following happen?
  member?(x,variables lim) =>
    member?(x,variables(answer := normalize lim)) =>
      error "limit: can't evaluate limit"
    answer :: OPF
CHAPTER 17. CHAPTER P

\[ \text{lim :: FE :: OPF} \]

`okProblem?(function,problem) ==
(function = "log") or (function = "nth root") =>
(problem = "series of non-zero order") or _
(function = "atan") => problem = "branch problem"
(function = "erf") => problem = "unknown kernel"
problem = "essential singularity"

`poleLimit(order,coef,x) ==
-- compute limit for function with pole
not member?(x,variables coef) =>
(s := sign(coef)$SIGNEF) case Integer =>
  rtLim := (s :: Integer) * plusInfinity()
  even? numer order => rtLim
  even? denom order => ["failed",rtLim]$TwoSide
  [-rtLim,rtLim]$TwoSide
-- infinite limit, but cannot determine sign
"failed"
error "limit: can't evaluate limit"

`poleLimitPlus(order,coef,x) ==
-- compute right hand limit for function with pole
not member?(x,variables coef) =>
(s := sign(coef)$SIGNEF) case Integer =>
  (s :: Integer) * plusInfinity()
-- infinite limit, but cannot determine sign
"failed"
(clim := specialLimit(coef,x)) case "failed" => "failed"
zero? (lim := clim :: OFE) =>
  -- in this event, we need to determine if the limit of
  -- the coef is 0+ or 0-
  (cclim := specialLimit(inv coef,x)) case "failed" => "failed"
  ss := whatInfinity(cclim :: OFE) :: Z
  zero? ss =>
    error "limit: internal error"
  ss * plusInfinity()
  t := whatInfinity(lim :: OFE) :: Z
  zero? t =>
    (tt := sign(coef)$SIGNEF) case Integer =>
      (tt :: Integer) * plusInfinity()
-- infinite limit, but cannot determine sign
"failed"
  t * plusInfinity()

`realLimit(fcn,x) ==
-- computes lim(x -> 0, fcn) using a Puiseux expansion of fcn,
-- if fcn is an expression involving roots, and using a Laurent
-- expansion of fcn otherwise
lim : Union(FE,"failed") :=
anyRootsOrAttrigs? fcn =>
  ppack := FS2UPS(R,FE,RN,
    UPXS(FE,x,zeroFE),EFUPXS(FE,ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),
    EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
pseries := exprToUPS(fcn,true,"real: two sides")$ppack
pseries case %problem =>
  trouble := pseries.%problem
  function := trouble.func; problem := trouble.prob
  okProblem?(function,problem) =>
    left :=
      xK : Kernel FE := kernel x
      fcn0 := eval(fcn,xK,-(xK :: FE))
      limitPlus(fcn0,x)
    right := limitPlus(fcn,x)
    (left case "failed") and (right case "failed") =>
      return "failed"
    if (left case OFE) and (right case OFE) then
      (left :: OFE) = (right :: OFE) => return (left :: OFE)
    return([left,right]$TwoSide)
    return "failed"
  if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
pole? upxs =>
    cp := coefficient(upxs,ordp := order upxs)
    return poleLimit(ordp,cp,x)
  coefficient(upxs,0)
lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),
  EFULS(FE,UTS(FE,x,zeroFE),ULS(FE,x,zeroFE))),x)
lseries := exprToUPS(fcn,true,"real: two sides")$lpack
lseries case %problem =>
  trouble := lseries.%problem
  function := trouble.func; problem := trouble.prob
  okProblem?(function,problem) =>
    left :=
      xK : Kernel FE := kernel x
      fcn0 := eval(fcn,xK,-(xK :: FE))
      limitPlus(fcn0,x)
    right := limitPlus(fcn,x)
    (left case "failed") and (right case "failed") =>
      return "failed"
    if (left case OFE) and (right case OFE) then
      (left :: OFE) = (right :: OFE) => return (left :: OFE)
    return([left,right]$TwoSide)
    return "failed"
  if pole?(uls := lseries.%series) then uls := map(normalize,uls)
pole? uls =>
    cl := coefficient(uls,ordl := order uls)
    return poleLimit(ordl :: RN,cl,x)
  coefficient(uls,0)
lim case "failed" => "failed"
member?(x,variables(lim :: FE)) =>
  member?(x,variables(answer := normalize(lim :: FE))) =>
  error "limit: can't evaluate limit"
answer :: OFE
lim :: FE :: OFE

xxpLimit(fcn,x) ==
  -- computes \( \lim(x \to 0^+, fcn) \) using an exponential expansion of \( fcn \)
  xpack := FS2EXPXP(R,FE,x,zeroFE)
  xxp := exprToXXP(fcn,true)$xpack
  xxp case %problem => "failed"
  limitPlus(xxp.%expansion)

limitPlus(fcn,x) ==
  -- computes \( \lim(x \to 0^+, fcn) \) using a generalized Puiseux expansion
  -- of \( fcn \), if \( fcn \) is an expression involving roots, and using a
  -- generalized Laurent expansion of \( fcn \) otherwise
  lim : Union(FE,"failed") :=
  anyRootsOrAtrigs? fcn =>
    ppack := FS2UPS(R,FE,RN,_
        UPSX(FE,x,zeroFE),EFUPXSF(ULS(FE,x,zeroFE),UPXS(FE,x,zeroFE),EFULS(FE,x,zeroFE)),EFULS(FE,x,zeroFE),x)
    pseries := exprToGenUPS(fcn,true,"real: right side")$ppack
    pseries case %problem =>
        trouble := pseries.%problem
        ff := trouble.func; pp := trouble.prob
        (pp = "negative leading coefficient") => return "failed"
        "failed"
    -- pseries case %problem => return "failed"
    if pole?(upxs := pseries.%series) then upxs := map(normalize,upxs)
    pole? upxs =>
        cp := coefficient(upxs,ordp := order upxs)
        return poleLimitPlus(ordp,cp,x)
    coefficient(upxs,0)
  lpack := FS2UPS(R,FE,Z,ULS(FE,x,zeroFE),_
        EFULS(FE,x,zeroFE),ULS(FE,x,zeroFE)),x)
  lseries := exprToGenUPS(fcn,true,"real: right side")$lpack
  lseries case %problem =>
    trouble := lseries.%problem
    ff := trouble.func; pp := trouble.prob
    (pp = "negative leading coefficient") => return "failed"
    "failed"
  -- lseries case %problem => return "failed"
  if pole?(uls := lseries.%series) then uls := map(normalize,uls)
  pole? uls =>
    cl := coefficient(uls,ordl := order uls)
    return poleLimitPlus(ordl :: RN,cl,x)
  coefficient(uls,0)
lim case "failed" =>
  (xLim := xxpLimit(fcn,x)) case "failed" => specialLimit(fcn,x)
package PREASSOC PrecomputedAssociatedEquations

<table>
<thead>
<tr>
<th>PrecomputedAssociatedEquations.input</th>
</tr>
</thead>
</table>

xLim
member?(x,variables(lim :: FE)) =>
member?(x,variables(answer := normalize(lim :: FE))) =>
(xLim := xxpLimit(answer,x)) case "failed" => specialLimit(answer,x)
xLim
answer :: OFE
lim :: FE :: OFE

limit(fcn:FE,eq:EQ OFE) ==
(f := retractIfCan(lhs eq)@Union(FE,"failed")) case "failed" =>
  error "limit:left hand side must be a variable"
(xx := retractIfCan(f)@Union(SY,"failed")) case "failed" =>
  error "limit:left hand side must be a variable"
x := xx :: SY; a := rhs eq
locallimit(fcn,x,a)

complexLimit(fcn:FE,eq:EQ OPF) ==
(f := retractIfCan(lhs eq)@Union(FE,"failed")) case "failed" =>
  error "limit:left hand side must be a variable"
(xx := retractIfCan(f)@Union(SY,"failed")) case "failed" =>
  error "limit:left hand side must be a variable"
x := xx :: SY; a := rhs eq
locallimitcomplex(fcn,x,a)

———

— LIMITPS.dotabb —

"LIMITPS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=LIMITPS"]
"ULSCCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ULSCCAT"]
"LIMITPS" -> "ULSCCAT"

———

package PREASSOC PrecomputedAssociatedEquations

<table>
<thead>
<tr>
<th>PrecomputedAssociatedEquations.input</th>
</tr>
</thead>
</table>

)set break resume
)sy s rm -f PrecomputedAssociatedEquations.output
)spool PrecomputedAssociatedEquations.output
)set message test on
)set message auto off
)clear all
PrecomputedAssociatedEquations stores some generic precomputations which speed up the computations of the associated equations needed for factoring operators.

See Also:
  o )show PrecomputedAssociatedEquations
PrecomputedAssociatedEquations (PREASSOC)

Exports:
firstUncouplingMatrix

— package PREASSOC PrecomputedAssociatedEquations —

)abbrev package PREASSOC PrecomputedAssociatedEquations
++ Author: Manuel Bronstein
++ Date Created: 13 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{PrecomputedAssociatedEquations} stores some generic
++ precomputations which speed up the computations of the
++ associated equations needed for factoring operators.

PrecomputedAssociatedEquations(R, L): Exports == Implementation where
R: IntegralDomain
L: LinearOrdinaryDifferentialOperatorCategory R

PI ==> PositiveInteger
N ==> NonNegativeInteger
A ==> PrimitiveArray R
U ==> Union(Matrix R, "failed")

Exports ==> with
firstUncouplingMatrix: (L, PI) -> U
  ++ firstUncouplingMatrix(op, m) returns the matrix A such that
  ++ \spad{A w = (W',W'',...,W^N)} in the corresponding associated
  ++ equations for right-factors of order m of op.
  ++ Returns "failed" if the matrix A has not been precomputed for
  ++ the particular combination \spad{degree(L), m}.

Implementation ==> add
A32:  L -> U
A42:  L -> U
A425: (A, A, A) -> List R
A426: \((A, A, A) \rightarrow \text{List } R\)
makeMonic: \(L \rightarrow \text{Union}(A, "failed")\)

diff:L := D()

firstUncouplingMatrix(op, m) ==
  n := degree op
  n = 3 and m = 2 => A32 op
  n = 4 and m = 2 => A42 op
  "failed"

makeMonic op ==
  lc := leadingCoefficient op
  a:A := new(n := degree op, 0)
  for i in 0..(n-1)::N repeat
    (u := coefficient(op, i) exquo lc) case "failed" => return "failed"
    a.i := - (u::R)
  a

A32 op ==
  (u := makeMonic op) case "failed" => "failed"
  a := u::A
  matrix [[0, 1, 0], [a.1, a.2, 1],
          [diff(a.1) + a.1 * a.2 - a.0, diff(a.2) + a.2**2 + a.1, 2 * a.2]]

A42 op ==
  (u := makeMonic op) case "failed" => "failed"
  a := u::A
  a':A := new(4, 0)
  a'':A := new(4, 0)
  for i in 0..3 repeat
    a'.i := diff(a.i)
    a''.i := diff(a'.i)
  matrix [[0, 1, 0, 0, 0, 0], [0, 0, 1, 1, 0, 0], [a.1,a.2,0,a.3,2::R,0],
          [diff(a.1) + a.1 * a.2 - a.0, diff(a.2) + a.2**2 + a.1, 3 * a.2,
           a'.3 + a.3 ** 2 + a.2, 3 * a.3, 2::R],
          A425(a, a', a''), A426(a, a', a'')]

A425(a, a', a'') ==
  [a''.1 + 2 * a.1 * a'.3 + a.3 * a'.1 - 2 * a'.0 + a.1 * a.3 ** 2
   - 3 * a.0 * a.3 + a.1 * a.2,
   a''.2 + 2 * a.2 * a'.3 + a.3 * a'.2 + 2 * a'.1 + a.2 * a.3 ** 2
   + a.1 * a.3 + a.2 ** 2 - 4 * a.0,
   4 * a'.2 + 4 * a.2 * a.3 - a.1,
   a''.3 + 3 * a.3 * a'.3 + 2 * a'.2 + a.3 ** 3 + 2 * a.2 * a.3 + a.1,
   4 * a'.3 + 4 * a.3 ** 2 + 4 * a.2, 5 * a.3]

A426(a, a', a'') ==
  [diff(a''.1) + 3 * a.1 * a''.3 + a.3 * a''.1 - 2 * a''.0
   + (3 * a'.1 + 5 * a.1 * a.3 - 7 * a.0) * a'.3 + 3 * a.1 * a'.2]
(a.3 ** 2 + a.2) * a'.1 - 3 * a.3 * a'.0 + a.1 * a.3 ** 3
- 4 * a.0 * a.3 ** 2 + 2 * a.1 * a.2 * a.3 - 4 * a.0 * a.2 + a.1 ** 2,
diff(a'',2) + 3 * a.2 * a'''.3 + a.3 * a'''.2 + 3 * a'''.1
+ (3*a'.2 + 5*a.2 * a.3 + 3 * a.1) * a'.3 + (a.3**2 + 4*a.2)*a'.2
+ 2 * a.3 * a'.1 - 6 * a'.0 + a.2 * a.3 ** 3 + a.1 * a.3 ** 2
+ (2 * a.2**2 - 8 * a.0) * a.3 + 2 * a.1 * a.2,
5 * a'''.2 + 10 * a.2 * a'.3 + 5 * a.3 * a'.2 + a'.1
+ 5 * a.2 * a.3 ** 2 - 4 * a.1 * a.3 + 5 * a.2**2 - 4 * a.0,
diff(a'''.3) + 4 * a.3 * a'''.3 + 3*a'''.2 + 3 * a'''.3**2
+ (6 * a.3**2 + 4 * a.2) * a'.3 + 5 * a.3 * a'.2 + 3 * a'.1
+ a.3**4 + 3 * a.2 * a.3**2 + 2 * a.1 * a.3 + a.2**2 - 4*a.0,
5 * a'''.3 + 15 * a.3 * a'.3 + 10 * a'.2 + 5 * a.3**3
+ 10 * a.2 * a.3, 9 * a'.3 + 9 * a.3**2 + 4 * a.2]

— PREASSOC.dotabb —

"PREASSOC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PREASSOC"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"PREASSOC" -> "OREPCAT"
"PREASSOC" -> "A1AGG"

package PRIMARR2 PrimitiveArrayFunctions2

— PrimitiveArrayFunctions2.input —

)set break resume
)sys rm -f PrimitiveArrayFunctions2.output
)spool PrimitiveArrayFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrimitiveArrayFunctions2

--R
--R PrimitiveArrayFunctions2(A: Type,B: Type) is a package constructor
--R Abbreviation for PrimitiveArrayFunctions2 is PRIMARR2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PRIMARR2
--R
--R--------------------------------------------- Operations ----------------------------------
--R map : ((A -> B),PrimitiveArray(A)) -> PrimitiveArray(B)
--R reduce : (((A,B) -> B),PrimitiveArray(A),B) -> B
--R scan : (((A,B) -> B),PrimitiveArray(A),B) -> PrimitiveArray(B)
--R
--E 1

)spool
)lisp (bye)

---

— PrimitiveArrayFunctions2.help —

====================================================================
PrimitiveArrayFunctions2 examples
====================================================================

This package provides tools for operating on primitive arrays
with unary and binary functions involving different underlying types

See Also:
  o )show PrimitiveArrayFunctions2

---

PrimitiveArrayFunctions2 (PRIMARR2)

Exports:
  map  reduce  scan

--- package PRIMARR2 PrimitiveArrayFunctions2 ---
++)abbrev package PRIMARR2 PrimitiveArrayFunctions2
++ Description:
++ This package provides tools for operating on primitive arrays
++ with unary and binary functions involving different underlying types

PrimitiveArrayFunctions2(A, B): Exports == Implementation where
   A, B: Type

   VA ==> PrimitiveArray A
   VB ==> PrimitiveArray B
   O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)

Exports ==> with
   scan : ((A, B) -> B, VA, B) -> VB
      ++ scan(f,a,r) successively applies
      ++ \spad{reduce(f,x,r)} to more and more leading sub-arrays
      ++ x of primitive array \spad{a}.
      ++ More precisely, if \spad{a} is \spad{[a1,a2,...]}, then
      ++ \spad{scan(f,a,r)} returns
      ++ \spad{[reduce(f,[a1],r),reduce(f,[a1,a2],r),...]}.
      ++
      ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
      ++X adder(a:Integer,b:Integer):Integer == a+b
      ++X scan(adder,[i for i in 1..10],0)$T1

   reduce : ((A, B) -> B, VA, B) -> B
      ++ reduce(f,a,r) applies function f to each
      ++ successive element of the
      ++ primitive array \spad{a} and an accumulant initialized to r.
      ++ For example, \spad{reduce(_+$Integer,[1,2,3],0)}
      ++ does \spad{3+(2+(1+0))}. Note that third argument r
      ++ may be regarded as the identity element for the function f.
      ++
      ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
      ++X adder(a:Integer,b:Integer):Integer == a+b
      ++X reduce(adder,[i for i in 1..10],0)$T1

   map : (A -> B, VA) -> VB
      ++ map(f,a) applies function f to each member of primitive array
      ++ \spad{a} resulting in a new primitive array over a
      ++ possibly different underlying domain.
      ++
      ++X T1:=PrimitiveArrayFunctions2(Integer,Integer)
      ++X map(x+->x+2,[i for i in 1..10])$T1

Implementation ==> add
   map(f, v) == map(f, v)$O2
   scan(f, v, b) == scan(f, v, b)$O2
   reduce(f, v, b) == reduce(f, v, b)$O2
package PRIMELT PrimitiveElement

-- PrimitiveElement.input --

)set break resume
)sys rm -f PrimitiveElement.output
)spool PrimitiveElement.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show PrimitiveElement
--R
--R PrimitiveElement(F: Join(Field,CharacteristicZero)) is a package constructor
--R Abbreviation for PrimitiveElement is PRIMELT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PRIMELT
--R
--R---------------------------------- Operations ----------------------------------
--R primitiveElement : (Polynomial(F),Symbol,Polynomial(F),Symbol) -> Record(coef1: Integer,coef2: Integer,prim: SparseUnivariatePolynomial(F))
--R primitiveElement : (List(Polynomial(F)),List(Symbol)) -> Record(coef: List(Integer),poly: List(SparseUnivariatePolynomial(F)),prim: SparseUnivariatePolynomial(F))
--R primitiveElement : (List(Polynomial(F)),List(Symbol),Symbol) -> Record(coef: List(Integer),poly: List(SparseUnivariatePolynomial(F)),prim: SparseUnivariatePolynomial(F))
--R
--E 1

)spool
)lisp (bye)

-- PrimitiveElement.help --

====================================================================
PrimitiveElement examples
====================================================================

PrimitiveElement provides functions to compute primitive elements
in algebraic extensions;

See Also:
o )show PrimitiveElement

---

PrimitiveElement (PRIMELT)

Exports:
primitiveElement

--- package PRIMELT PrimitiveElement ---

)abbrev package PRIMELT PrimitiveElement
++ Author: Manuel Bronstein
++ Date Created: 6 Jun 1990
++ Date Last Updated: 25 April 1991
++ Description:
++ PrimitiveElement provides functions to compute primitive elements
++ in algebraic extensions;

PrimitiveElement(F):Exports == Implementation where
  F : Join(Field, CharacteristicZero)
  SY ==> Symbol
  P  ==> Polynomial F
  UP ==> SparseUnivariatePolynomial F
  RC  ==> Record(coef1: Integer, coef2: Integer, prim:UP)


CHAPTER 17.  CHAPTER P

REC ==> Record(coef: List Integer, poly:List UP, prim: UP)

Exports ==> with
primitiveElement: (P, SY, P, SY) -> RC
++ primitiveElement(p1, a1, p2, a2) returns \spad{[c1, c2, q]}
++ such that \spad{k(a1, a2) = k(a)}
++ where \spad{a = c1 a1 + c2 a2}, and \spad{q(a) = 0}.
++ The pi's are the defining polynomials for the ai's.
++ The p2 may involve a1, but p1 must not involve a2.
++ This operation uses \spadfun{resultant}.

primitiveElement: (List P, List SY) -> REC
++ primitiveElement([p1,...,pn], [a1,...,an]) returns
++ \spad{[[c1,...,cn], [q1,...,qn], q]} such that then \spad{k(a1,...,an) = k(a)},
++ where \spad{a = a1 c1 + ... + an cn},
++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
++ The pi's are the defining polynomials for the ai's.
++ This operation uses the technique of
++ \spadglossSee{groebner bases}{Groebner basis}.

primitiveElement: (List P, List SY, SY) -> REC
++ primitiveElement([p1,...,pn], [a1,...,an], a) returns
++ \spad{[[c1,...,cn], [q1,...,qn], q]} such that then \spad{k(a1,...,an) = k(a)},
++ where \spad{a = a1 c1 + ... + an cn},
++ \spad{ai = qi(a)}, and \spad{q(a) = 0}.
++ The pi's are the defining polynomials for the ai's.
++ This operation uses the technique of
++ \spadglossSee{groebner bases}{Groebner basis}.

Implementation ==> add
import PolyGroebner(F)

multi : (UP, SY) -> P
randomInts: (NonNegativeInteger, NonNegativeInteger) -> List Integer
findUniv : (List P, SY, SY) -> Union(P, "failed")
incl? : (List SY, List SY) -> Boolean
triangularLinearIfCan:(List P,List SY,SY) -> Union(List UP,"failed")
innerPrimitiveElement: (List P, List SY, SY) -> REC

multi(p, v) == multivariate(map((f1:F):F +-> f1, p), v)
randomInts(n, m) == [symmetricRemainder(random()$Integer, m) for i in 1..n]
incl?(a, b) == every?(s1:SY):Boolean +-> member?(s1, b), a)
primitiveElement(l, v) == primitiveElement(l, v, new()$SY)

primitiveElement(p1, a1, p2, a2) ==
  -- one? degree(p2, a1) => [0, 1, univariate resultant(p1, p2, a1)]
  (degree(p2, a1) = 1) => [0, 1, univariate resultant(p1, p2, a1)]
  u := (new()$SY)::P
  b := a2::P
  for i in 10.. repeat

c := symmetricRemainder(random($Integer, i)
w := u - c * b
r := univariate resultant(eval(p1, a1, w), eval(p2, a1, w), a2)
not zero? r and r = squareFreePart r => return [i, c, r]

findUniv(l, v, opt) ==
  for p in l repeat
    degree(p, v) > 0 and incl?(variables p, [v, opt]) => return p
"failed"

triangularLinearIfCan(l, lv, w) ==
  (u := findUniv(l, w, w)) case "failed" => "failed"
pw := univariate(u::P)
l1 := nil()$List(UP)
for v in lv repeat
  ((u := findUniv(l, v, w)) case "failed") or
  (degree(p := univariate(u::P, v)) ^= 1) => return "failed"
  (bc := extendedEuclidean(univariate leadingCoefficient p, pw,1))
    case "failed" => error "Should not happen"
  ll := concat(map((z1:F):F +-> z1,
    (- univariate(coefficient(p,0)) * bc.coef1) rem pw), ll)
  concat(map((f1:F):F +-> f1, pw), reverse_! ll)

primitiveElement(l, vars, uu) ==
  u := uu::P
  vv := [v::P for v in vars]
  elim := concat(vars, uu)
  w := uu::P
  n := #l
  for i in 10.. repeat
    cf := randomInts(n, i)
    (tt := triangularLinearIfCan(lexGroebner(cons(w + /[c * t for c in cf for t in vv], l), elim),
      vars, uu)) case List(UP) =>
    ltt := tt::List(UP)
    return([cf, rest ltt, first ltt])
package ODEPRIM PrimitiveRatDE

— PrimitiveRatDE.input —

)set break resume
)sys rm -f PrimitiveRatDE.output
)spool PrimitiveRatDE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrimitiveRatDE
--R
--R PrimitiveRatDE(F: Join(Field,CharacteristicZero,RetractableTo(Fraction(Integer))),UP: Universal)
--R Abbreviation for PrimitiveRatDE is ODEPRIM
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODEPRIM
--R
--R------------------------------- Operations --------------------------------
--R indicialEquation : (L,F) -> UP indicialEquation : (LQ,F) -> UP
--R denomLODE : (L,Fraction(UP)) -> Union(UP,"failed")
--R denomLODE : (L,List(Fraction(UP))) -> UP
--R indicialEquations : L -> List(Record(center: UP,equation: UP))
--R indicialEquations : (L,UP) -> List(Record(center: UP,equation: UP))
--R indicialEquations : LQ -> List(Record(center: UP,equation: UP))
--R indicialEquations : (LQ,UP) -> List(Record(center: UP,equation: UP))
--R splitDenominator : (LQ,List(Fraction(UP))) -> Record(eq: L,rh: List(Fraction(UP)))
--R
--E 1

)spool
)lisp (bye)

——

— PrimitiveRatDE.help —

====================================================================
PrimitiveRatDE examples
====================================================================

PrimitiveRatDE provides functions for in-field solutions of linear
ordinary differential equations, in the transcendental case.
The derivation to use is given by the parameter L.

See Also:
o )show PrimitiveRatDE
PrimitiveRatDE (ODEPRIM)

Exports:
indicialEquation  denomLODE  indicialEquations  splitDenominator

— package ODEPRIM PrimitiveRatDE —

)abbrev package ODEPRIM PrimitiveRatDE
++ Author: Manuel Bronstein
++ Date Created: 1 March 1991
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{PrimitiveRatDE} provides functions for in-field solutions of linear
++ ordinary differential equations, in the transcendental case.
++ The derivation to use is given by the parameter \spad{L}.

PrimitiveRatDE(F, UP, L, LQ):Exports == Implementation where
   F : Join(Field, CharacteristicZero, RetractableTo Fraction Integer)
   UP : UnivariatePolynomialCategory F
   L : LinearOrdinaryDifferentialOperandCategory UP
   LQ : LinearOrdinaryDifferentialOperandCategory Fraction UP

N  ==>  NonNegativeInteger
Z  ==>  Integer
RF ==>  Fraction UP
UP2 ==>  SparseUnivariatePolynomial UP
REC ==>  Record(center:UP, equation:UP)

Exports ==> with
   denomLODE: (L, RF) -> Union(UP, "failed")
++ denomLODE(op, g) returns a polynomial d such that
++ any rational solution of \spad{op y = g} is of the form \spad{p/d} for some polynomial p, and
++ "failed", if the equation has no rational solution.

\texttt{denomLODE: (L, List RF) -> UP}
++ denomLODE(op, [g1,...,gm]) returns a polynomial d such that any rational solution of \spad{op y = c1 g1 + ... + cm gm} is of the form \spad{p/d} for some polynomial p.

\texttt{indicialEquations: L -> List REC}
++ indicialEquations op returns \spad{[[d1,e1],...,[dq,eq]]} where
++ the \spad{d_i}'s are the affine singularities of \spad{op},
++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.

\texttt{indicialEquations: (L, UP) -> List REC}
++ indicialEquations(op, p) returns \spad{[[d1,e1],...,[dq,eq]]} where
++ the \spad{d_i}'s are the affine singularities of \spad{op}
++ above the roots of \spad{p},
++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.

\texttt{indicialEquation: (L, F) -> UP}
++ indicialEquation(op, a) returns the indicial equation of \spad{op}
++ at \spad{a}.

\texttt{indicialEquations: LQ -> List REC}
++ indicialEquations op returns \spad{[[d1,e1],...,[dq,eq]]} where
++ the \spad{d_i}'s are the affine singularities of \spad{op},
++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.

\texttt{indicialEquations: (LQ, UP) -> List REC}
++ indicialEquations(op, p) returns \spad{[[d1,e1],...,[dq,eq]]} where
++ the \spad{d_i}'s are the affine singularities of \spad{op}
++ above the roots of \spad{p},
++ and the \spad{e_i}'s are the indicial equations at each \spad{d_i}.

\texttt{indicialEquation: (LQ, F) -> UP}
++ indicialEquation(op, a) returns the indicial equation of \spad{op}
++ at \spad{a}.

\texttt{splitDenominator: (LQ, List RF) -> Record(eq:L, rh:List RF)}
++ splitDenominator(op, [g1,...,gm]) returns \spad{op0, [h1,...,hm]} such that the equations \spad{op y = c1 g1 + ... + cm gm} and
++ \spad{op0 y = c1 h1 + ... + cm hm} have the same solutions.

\texttt{Implementation ==> add}
import BoundIntegerRoots(F, UP)
import BalancedFactorisation(F, UP)
import InnerCommonDenominator(UP, RF, List UP, List RF)
import UnivariatePolynomialCategoryFunctions2(F, UP, UP, UP2)

\texttt{tau : (UP, UP, UP, N) -> UP}
\texttt{NPbound : (UP, L, UP) -> N}
\texttt{hdenom : (L, UP, UP) -> UP}
\texttt{denom0 : (Z, L, UP, UP, UP) -> UP}
\texttt{indicialEq : (UP, List N, List UP) -> UP}
\texttt{separateZeros: (UP, UP) -> UP}
\texttt{UPfact : N -> UP}
**PACKAGE ODEPRIM PRIMITIVERATDE**

```
UP2UP2 : UP -> UP2
indeq : (UP, L) -> UP
NPmulambda : (UP, L) -> Record(mu:Z, lambda:List N, func:List UP)

diff := D()$L

UP2UP2 p == map((f1:F):UP +->f1::UP, p)
indicialEquations(op:L) == indicialEquations(op, leadingCoefficient op)
indicialEquation(op:L, a:F) == indeq(monomial(1, 1) - a::UP, op)

splitDenominator(op, lg) ==
  cd := splitDenominator coefficients op
  f := cd.den / gcd(cd.num)
  l:L := 0
  while op ^= 0 repeat
    l := l + monomial(retract(f * leadingCoefficient op), degree op)
    op := reductum op
  [l, [f * g for g in lg]]

tau(p, pp, q, n) ==
  ((pp ** n) * ((q exquo (p ** order(q, p)))::UP)) rem p

indicialEquations(op:LQ) ==
  indicialEquations(splitDenominator(op, empty()).eq)

indicialEquations(op:LQ, p:UP) ==
  indicialEquations(splitDenominator(op, empty()).eq, p)

indicialEquation(op:LQ, a:F) ==
  indeq(monomial(1, 1) - a::UP, splitDenominator(op, empty()).eq)

-- returns z(z-1)...(z-(n-1))
UPfact n ==
  zero? n => 1
  z := monomial(1, 1)$UP
  */[z - i::F::UP for i in 0..(n-1)::N]

indicialEq(c, lamb, lf) ==
  cp := diff c
  cc := UP2UP2 c
  s:UP2 := 0
  for i in lamb for f in lf repeat
    s := s + (UPfact i) * UP2UP2 tau(c, cp, f, i)
  primitivePart resultant(cc, s)

NPmulambda(c, l) ==
  lamb:List(N) := [d := degree l]
  lf:List(UP) := [a := leadingCoefficient l]
  mup := d::Z - order(a, c)
  while (l := reductum l) ^= 0 repeat
```
CHAPTER 17. CHAPTER P

\begin{verbatim}
3510

a := leadingCoefficient l
if (m := (d := degree l)::Z - order(a, c)) > mup then
  mup := m
  lamb := [d]
  lf := [a]
else if (m = mup) then
  lamb := concat(d, lamb)
  lf := concat(a, lf)
[mup, lamb, lf]

-- e = 0 means homogeneous equation
NPbound(c, l, e) ==
  rec := NPmulambda(c, l)
  n := max(0, - integerBound indicialEq(c, rec.lambda, rec.func))
  zero? e => n::N
  max(n, order(e, c)::Z - rec.mu)::N

hdenom(l, d, e) ==
  */[dd.factor ** NPbound(dd.factor, l, e)
    for dd in factors balancedFactorisation(d, coefficients l)]

denom0(n, l, d, e, h) ==
  hdenom(l, d, e) * */[hh.factor ** max(0, order(e, hh.factor) - n)::N
    for hh in factors balancedFactorisation(h, e)]

-- returns a polynomials whose zeros are the zeros of e which are not
-- zeros of d
separateZeros(d, e) ==
  ((g := squareFreePart e) exquo gcd(g, squareFreePart d))::UP

indeq(c, l) ==
  rec := NPmulambda(c, l)
  indicialEq(c, rec.lambda, rec.func)

indicialEquations(op:L, p:UP) ==
  [[dd.factor, indeq(dd.factor, op)]
    for dd in factors balancedFactorisation(p, coefficients op)]

-- cannot return "failed" in the homogeneous case
denomLODE(l:L, g:RF) ==
  d := leadingCoefficient l
  zero? g => hdenom(l, d, 0)
  h := separateZeros(d, e := denom g)
  n := degree l
  (e exquo (h**(n + 1))) case "failed" => "failed"
  denom0(n, l, d, e, h)

denomLODE(l:L, lg:List RF) ==
  empty? lg => denomLODE(l, 0)::UP
  d := leadingCoefficient l
\end{verbatim}
h := separateZeros(d, e := "lcm/[denom g for g in lg]")
denom0(degree l, l, d, e, h)

---

ODOPRIM.dotabb ---

"ODOPRIM" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODOPRIM"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"ODOPRIM" -> "PFECAT"

---

class ODEPRRIC PrimitiveRatRicDE

--- PrimitiveRatRicDE.input ---

)set break resume
)sys rm -f PrimitiveRatRicDE.output
)spool PrimitiveRatRicDE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PrimitiveRatRicDE

--

-- This constructor is not exposed in this frame.

-- Issue )edit bookvol10.4.pamphlet to see algebra source code for ODEPRRIC
--

--R----------------------------- Operations -----------------------------
--R changeVar : (L,UP) -> L
--R denomRicDE : L -> UP
--R constantCoefficientRicDE : (L,(UP -> List(F))) -> List(Record(constant: F, eq: L))
--R leadingCoefficientRicDE : L -> List(Record(deg: NonNegativeInteger, eq: UP))
--R polyRicDE : (L,(UP -> List(F))) -> List(Record(poly: UP, eq: L))
--R singRicDE : (L,(UP,SparseUnivariatePolynomial(UP)) -> List(UP)),(UP -> Factored(UP))) -> List(Record francais)

--E 1

)spool
)lisp (bye)
CHAPTER 17. CHAPTER P

---

PrimitiveRatRicDE (ODEPRRIC)

Exports:
  changeVar  denomRicDE  constantCoefficientRicDE  leadingCoefficientRicDE
  polyRicDE  singRicDE

---

)abbrev package ODEPRRIC PrimitiveRatRicDE
++ Author: Manuel Bronstein
++ Date Created: 22 October 1991
++ Date Last Updated: 2 February 1993
++ Description:
++ In-field solution of Riccati equations, primitive case.

PrimitiveRatRicDE(F, UP, L, LQ):Exports == Implementation where
  F: Join(Field, CharacteristicZero, RetractableTo Fraction Integer)
  UP: UnivariatePolynomialCategory F
L : LinearOrdinaryDifferentialOperatorCategory UP
LQ : LinearOrdinaryDifferentialOperatorCategory Fraction UP

N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
UP2 ==> SparseUnivariatePolynomial UP
REC ==> Record(deg:N, eq:UP)
REC2 ==> Record(deg:N, eq:UP2)
POL ==> Record(poly:UP, eq:L)
FRC ==> Record(frac:RF, eq:L)
CNT ==> Record(constant:F, eq:L)
IJ ==> Record(ij: List Z, deg:N)

Exports ==> with

denomRicDE: L -> UP
++ denomRicDE(op) returns a polynomial \spad{d} such that any rational
++ solution of the associated Riccati equation of \spad{op y = 0} is
++ of the form \spad{p/d + q'/q + r} for some polynomials p and q
++ and a reduced r. Also, \spad{deg(p) < deg(d)} and \{gcd(d,q) = 1\}.

leadingCoefficientRicDE: L -> List REC
++ leadingCoefficientRicDE(op) returns
++ \spad{[[m1, p1], [m2, p2], ... , [mk, pk]]} such that the polynomial
++ part of any rational solution of the associated Riccati equation of
++ \spad{op y = 0} must have degree mj for some j, and its leading
++ coefficient is then a zero of pj. In addition,\spad{m1>m2> ... >mk}.

coefficientCoefficientRicDE: (L, UP -> List F) -> List CNT
++ coefficientCoefficientRicDE(op, ric) returns
++ \spad{[[a1, L1], [a2, L2], ... , [ak, Lk]]} such that any rational
++ solution with no polynomial part of the associated Riccati equation of
++ \spad{op y = 0} must be one of the ai's in which case the equation for
++ \spad{z = y e^{-\operatorname{int} ai}} is \spad{Li z = 0}.
++ \spad{ric} is a Riccati equation solver over \spad{F}, whose input
++ is the associated linear equation.

polyRicDE: (L, UP -> List F) -> List POL
++ polyRicDE(op, zeros) returns
++ \spad{[[p1, L1], [p2, L2], ... , [pk, Lk]]} such that the polynomial
++ part of any rational solution of the associated Riccati equation of
++ \spad{op y=0} must be one of the pi's (up to the constant coefficient),
++ in which case the equation for \spad{z=y e^{-\operatorname{int} p}} is \spad{Li z = 0}.
++ \spad{zeros} is a zero finder in \spad{UP}.

singRicDE: (L, (UP, UP2) -> List UP, UP -> Factored UP) -> List FRC
++ singRicDE(op, zeros, ezfactor) returns
++ \spad{[[f1, L1], [f2, L2], ... , [fk, Lk]]} such that the singular
++ part of any rational solution of the associated Riccati equation of
++ \spad{op y=0} must be one of the fi's (up to the constant coefficient),
++ in which case the equation for \spad{z=y e^{-\operatorname{int} p}} is \spad{Li z=0}.
++ \spad{zeros(C(x),H(x,y))} returns all the \spad{P_i(x)}'s such that
++ \spad{H(x,P_i(x)) = 0} modulo \spad{C(x)}.
++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
++ not necessarily into irreducibles.

\begin{verbatim}
CHAPTER 17. CHAPTER P

+ changeVar: (L, UP) \rightarrow L
+ changeVar(+/[ai D^i], a) returns the operator \texttt{+/[ai (D+a)^i]}.

+ changeVar(+/[ai D^i], a) returns the operator \texttt{+/[ai (D+a)^i]}.

Implementation ==> add
import PrimitiveRatDE(F, UP, L, LQ)
import BalancedFactorisation(F, UP)

bound : (UP, L) \rightarrow \texttt{N}
lambda : (UP, L) \rightarrow \texttt{List IJ}
infmax : (IJ, L) \rightarrow \texttt{List Z}
dmax : (IJ, UP, L) \rightarrow \texttt{List Z}
getPoly : (IJ, L, LQ) \rightarrow \texttt{UP2}
getPol : (IJ, UP, L, List Z) \rightarrow \texttt{UP2}
innerlb : (L, UP \rightarrow \texttt{Z}) \rightarrow \texttt{List IJ}
innermax : (IJ, L, UP \rightarrow \texttt{Z}) \rightarrow \texttt{List Z}
tau0 : (UP, UP) \rightarrow \texttt{UP}
poly1 : (UP, UP, Z) \rightarrow \texttt{UP2}
getPol1 : (IJ, UP, L, List Z) \rightarrow \texttt{UP2}
getIndices : (N, List IJ) \rightarrow \texttt{List Z}
refine : (List UP, UP \rightarrow \texttt{Factored UP}) \rightarrow \texttt{List UP}
polysol : (L, N, Boolean, UP \rightarrow \texttt{List F}) \rightarrow \texttt{List POL}
fracsol : (L, (UP, UP2) \rightarrow \texttt{List UP}, List UP) \rightarrow \texttt{List FRC}
padicsol l : (UP, L, N, Boolean, (UP, UP2) \rightarrow \texttt{List UP}) \rightarrow \texttt{List FRC}
leadingDenomRicDE : (UP, L) \rightarrow \texttt{List REC2}
factoredDenomRicDE: L \rightarrow \texttt{List UP}
constantCoefficientOperator: (L, N) \rightarrow \texttt{UP}

infLambda: L \rightarrow \texttt{List IJ}
-- infLambda(op) returns
-- \texttt{[[i,j], (\deg(a_i)-\deg(a_j))/(i-j)]} for all the pairs
-- of indices \texttt{[i,j]} such that \texttt{\deg(a_i)-\deg(a_j)/(i-j)} is
-- an integer.

diff := D()$L
diffq := D()$LQ

lambda(c, l) == innerlb(l, z +-> order(z, c)::Z)
infLambda l  == innerlb(l, z +-> (degree(z)::Z))
infmax(rec,l) == innermax(rec, l, z +-> degree(z)::Z)
dmax(rec, c,l) == innermax(rec, 1, z +-> - order(z, c)::Z)
tau0(p, q)   == ((q exquo (p ** order(q, p)))::UP) rem p
poly1(c, cp,i) == */[\texttt{monomial(1,1)}$UP2 - (j * cp)::UP2 for j in 0..i-1]
getIndices(n,l) == removeDuplicates_! concat [r.ij for r in l | r.deg=n]
denomRicDE l == */[c ** bound(c, 1) for c in factoredDenomRicDE l]
polyRicDE(l,zeros) == concat([0, l], polysol(l, 0, false, zeros))

-- refine([p1,...,pn], foo) refines the list of factors using foo
refine(l, efactor) ==
\end{verbatim}
concat [[r.factor for r in factors ezfactor p] for p in l]

-- returns [] if the solutions of l have no p-adic component at c
padicsol(c, op, b, finite?, zeros) ==
  ans:List(FRC) := empty()
  finite? and zero? b => ans
  lc := leadingDenomRicDE(c, op)
  if finite? then lc := select_!(z +-> z.deg <= b, lc)
  for rec in lc repeat
    for r in zeros(c, rec.eq) | r ^= 0 repeat
      rcn := r /$RF (c ** rec.deg)
      neweq := changeVar(op, rcn)
      sols := padicsol(c, neweq, (rec.deg-1)::N, true, zeros)
      ans :=
        empty? sols => concat([rcn, neweq], ans)
        concat_!([[rcn + sol.frac, sol.eq] for sol in sols], ans)
        ans

leadingDenomRicDE(c, l) ==
  ind:List(Z) -- to cure the compiler... (won't compile without)
  lb := lambda(c, l)
  done:List(N) := empty()
  ans:List(REC2) := empty()
  for rec in lb | (not member?(rec.deg, done)) and
    not(empty?(ind := dmax(rec, c, l))) repeat
    ans := concat([rec.deg, getPol(rec, c, l, ind)], ans)
    done := concat(rec.deg, done)
  sort_!((z1,z2) +-> z1.deg > z2.deg, ans)

getPol(rec, c, l, ind) ==
  -- one?(rec.deg) => getPol1(ind, c, l)
  (rec.deg = 1) => getPol1(ind, c, l)
  +/[monomial(tau0(c, coefficient(l, i::N)), i::N)$UP2 for i in ind]

getPol1(ind, c, l) ==
  cp := diff c
  +/[tau0(c, coefficient(l, i::N)) * poly1(c, cp, i) for i in ind]

constantCoefficientRicDE(op, ric) ==
  m := "max"/[degree p for p in coefficients op]
  [[a, changeVar(op,a::UP)] for a in ric constantCoefficientOperator(op,m)]

constantCoefficientOperator(op, m) ==
  ans:UP := 0
  while op ^= 0 repeat
    if degree(p := leadingCoefficient op) = m then
      ans := ans + monomial(leadingCoefficient p, degree op)
    op := reductum op
  ans
CHAPTER 17. CHAPTER P

getPoly(rec, 1, ind) ==
+\[\text{monomial}(\text{leadingCoefficient coefficient}(l,i::N),i::N)\]$UP for i in ind

-- returns empty() if rec is does not reach the max,
-- the list of indices (including rec) that reach the max otherwise
innermax(rec, l, nu) ==
n := degree l
i := first(rec.ij)
m := i * (d := rec.deg) + nu coefficient(l, i::N)
ans := List(Z) := empty()
for j in 0..n | (f := coefficient(l, j)) ^= 0 repeat
  if ((k := (j * d + nu f)) > m) then return empty()
  else if (k = m) then ans := concat(j, ans)
ans

leadingCoefficientRicDE l ==
ind := List(Z) -- to cure the compiler... (won't compile without)
lb := infLambda l
done := List(N) := empty()
ans := List(REC) := empty()
for rec in lb | (not member?(rec.deg, done)) and
  not(empty?(ind := infmax(rec, l))) repeat
  ans := concat([rec.deg, getPoly(rec, l, ind)], ans)
done := concat(rec.deg, done)
sort_!((z1,z2) +-> z1.deg > z2.deg, ans)
factoredDenomRicDE l ==
bd := factors balancedFactorisation(leadingCoefficient l, coefficients l)
[dd.factor for dd in bd]

changeVar(l:L, a:UP) ==
dpa := diff + a::L -- the operator (D + a)
dpan := 1 -- will accumulate the powers of (D + a)
op := 0
for i in 0..degree l repeat
  op := op + coefficient(l, i) * dpan
dpan := dpa * dpan
primitivePart op

changeVar(l:L, a:RF) ==
dpa := diffq + a::LQ -- the operator (D + a)
dpan := 1 -- will accumulate the powers of (D + a)
op := 0
for i in 0..degree l repeat
  op := op + coefficient(l, i):RF * dpan
dpan := dpa * dpan
splitDenominator(op, empty()).eq

bound(c, l) ==
empty?(lb := lambda(c, l)) => 1
"max"/[rec.deg for rec in lb]

-- returns all the pairs [[i, j], n] such that
-- n = (nu(i) - nu(j)) / (i - j) is an integer
innerlb(l, nu) ==
  lb:List(IJ) := empty()
  n := degree l
  for i in 0..n | (li := coefficient(l, i)) ^= 0 repeat
    for j in i+1..n | (lj := coefficient(l, j)) ^= 0 repeat
      u := (nu li - nu lj) exquo (i-j)
      if (u case Z) and ((b := u::Z) > 0) then
        lb := concat([[i, j], b::N], lb)
  lb

singRicDE(l, zeros, ezfactor) ==
  concat([0, l], fracsol(l, zeros, refine(factoredDenomRicDE l, ezfactor)))

-- returns [] if the solutions of l have no singular component
fracsol(l, zeros, lc) ==
  ans:List(FRC) := empty()
  empty? lc => ans
  empty?(sols := padicsol(first lc, l, 0, false, zeros)) =>
    fracsol(l, zeros, rest lc)
  for rec in sols repeat
    neweq := changeVar(l, rec.frac)
    sols := fracsol(neweq, zeros, rest lc)
    ans :=
    empty? sols => concat(rec, ans)
    concat_!([[rec.frac + sol.frac, sol.eq] for sol in sols], ans)
  ans

-- returns [] if the solutions of l have no polynomial component
polysol(l, b, finite?, zeros) ==
  ans:List(POL) := empty()
  finite? and zero? b => ans
  lc := leadingCoefficientRicDE l
  if finite? then lc := select_!(z +-> z.deg <= b, lc)
  for rec in lc repeat
    for a in zeros(rec.eq) | a ^= 0 repeat
      atn:UP := monomial(a, rec.deg)
      neweq := changeVar(l, atn)
      sols := polysol(neweq, (rec.deg - 1)::N, true, zeros)
      ans :=
      empty? sols => concat([atn, neweq], ans)
      concat_!([[atn + sol.poly, sol.eq] for sol in sols], ans)
    ans
package PRINT PrintPackage

PrintPackage examples

PrintPackage provides a print function for output forms.
See Also:
- `show PrintPackage`

---

PrintPackage (PRINT)

Exports:
- `print`

---

`package PRINT PrintPackage`

++)abbrev package PRINT PrintPackage
++ Author: Scott Morrison
++ Date Created: Aug. 1, 1990
++ Description:
++ PrintPackage provides a print function for output forms.

PrintPackage(): with
  print : OutputForm -> Void
  ++ print(o) writes the output form o on standard output using the
  ++ two-dimensional formatter.
== add
  print(x) == print(x)$OutputForm

---

- PRINT.dotabb -

"PRINT" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PRINT"]
package PSEUDLIN PseudoLinearNormalForm

--- PseudoLinearNormalForm.input ---

)set break resume
/sys rm -f PseudoLinearNormalForm.output
/spool PseudoLinearNormalForm.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PseudoLinearNormalForm
--R
--R PseudoLinearNormalForm(K: Field) is a package constructor
--R Abbreviation for PseudoLinearNormalForm is PSEUDLIN
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PSEUDLIN
--R
--R------------------------------------------------------------------- Operations -----------------------------------
--R changeBase : (Matrix(K),Matrix(K),Automorphism(K),(K -> K)) -> Matrix(K)
--R companionBlocks : (Matrix(K),Vector(K)) -> List(Record(C: Matrix(K),g: Vector(K)))
--R normalForm : (Matrix(K),Automorphism(K),(K -> K)) -> Record(R: Matrix(K),A: Matrix(K),Ainv:
--R
--E 1

)spool
)lisp (bye)

---

--- PseudoLinearNormalForm.help ---

====================================================================
PseudoLinearNormalForm examples====================================================================

PseudoLinearNormalForm provides a function for computing a block-companion form for pseudo-linear operators.
See Also:
o )show PseudoLinearNormalForm

PseudoLinearNormalForm (PSEUDLIN)

Exports:
  changeBase  companionBlocks  normalForm

--- package PSEUDLIN PseudoLinearNormalForm ---

)abbrev package PSEUDLIN PseudoLinearNormalForm
++ Author: Bruno Zuercher
++ Date Created: November 1993
++ Date Last Updated: 12 April 1994
++ Description:
++ PseudoLinearNormalForm provides a function for computing a block-companion
++ form for pseudo-linear operators.

PseudoLinearNormalForm(K:Field): Exports == Implementation where
  ER ==> Record(C: Matrix K, g: Vector K)
  REC ==> Record(R: Matrix K, A: Matrix K, Ainv: Matrix K)

Exports == with
  normalForm: (Matrix K, Automorphism K, K -> K) -> REC
    ++ normalForm(M, sig, der) returns \spad{[R, A, A^{-1}]} such that
    ++ the pseudo-linear operator whose matrix in the basis \spad{y} is
    ++ \spad{M} had matrix \spad{R} in the basis \spad{z = A y}.
    ++ \spad{der} is a \spad{sig}-derivation.
  changeBase: (Matrix K, Matrix K, Automorphism K, K -> K) -> Matrix K
    ++ changeBase(M, A, sig, der): computes the new matrix of a pseudo-linear
    ++ transform given by the matrix M under the change of base A
companionBlocks: (Matrix K, Vector K) -> List ER
++ companionBlocks(m, v) returns \spad{[[C_1, g_1],..., [C_k, g_k]]}
++ such that each \spad{C_i} is a companion block and
++ \spad{m = diagonal(C_1,...,C_k)}.

Implementation == add

normalForm0: (Matrix K, Automorphism K, Automorphism K, K -> K) -> REC
mulMatrix: (Integer, Integer, K) -> Matrix K
-- mulMatrix(N, i, a): under a change of base with the resulting matrix of
-- size N*N the following operations are performed:
-- D1: column i will be multiplied by sig(a)
-- D2: row i will be multiplied by 1/a
-- D3: addition of der(a)/a to the element at position (i,i)
addMatrix: (Integer, Integer, Integer, K) -> Matrix K
-- addMatrix(N, i, k, a): under a change of base with the resulting matrix
-- of size N*N the following operations are performed:
-- C1: addition of column i multiplied by sig(a) to column k
-- C2: addition of row k multiplied by -a to row i
-- C3: addition of -a*der(a) to the element at position (i,k)
permutationMatrix: (Integer, Integer, Integer) -> Matrix K
-- permutationMatrix(N, i, k): under a change of base with the resulting
-- permutation matrix of size N*N the following operations are performed:
-- P1: columns i and k will be exchanged
-- P2: rows i and k will be exchanged
inv: Matrix K -> Matrix K
-- inv(M): computes the inverse of a invertable matrix M.
-- avoids possible type conflicts

inv m == inverse(m) :: Matrix K
changeBase(M, A, sig, der) ==
  inv(A) * (M * map((k1:K):K +-> sig k1, A) + map(der, A))
normalForm0(M, sig, der) == normalForm0(M, sig, inv sig, der)

companionBlocks(R, w) ==
-- decomposes the rational matrix R into single companion blocks
-- and the inhomogenity w as well
i:Integer := 1
n := nrows R
l:List(ER) := empty()
while i <= n repeat
  j := i
  while j+1 <= n and R(j, j+1) = 1 repeat j := j+1
  -- split block now
  v:Vector K := new((j-i+1)::NonNegativeInteger, 0)
  for k in i..j repeat v(k-i+1) := w k
  l := concat([subMatrix(R, i, j, i, j), v], l)
  i := j+1
l

normalForm0(M, sig, siginv, der) ==
-- the changes of base will be incremented in B and Binv,
-- where B**(-1)=Binv; E defines an elementary matrix
B, Binv, E : Matrix K
recOfMatrices : REC
N := nrows M
B := diagonalMatrix [1 for k in 1..N]
Binv := copy B
-- avoid unnecessary recursion
if diagonal?(M) then return [M, B, Binv]
i : Integer := 1
while i < N repeat
    j := i + 1
    while j <= N and M(i, j) = 0 repeat j := j + 1
    if j <= N then
        -- expand companionblock by lemma 5
        if j ^= i+1 then
            -- perform first a permutation
            E := permutationMatrix(N, i+1, j)
            M := changeBase(M, E, sig, der)
            B := B*E
            Binv := E*Binv
        -- now is M(i, i+1) ^= 0
        E := mulMatrix(N, i+1, siginv inv M(i,i+1))
        M := changeBase(M, E, sig, der)
        B := B*E
        Binv := inv(E)*Binv
        for j in 1..N repeat
            if j ^= i+1 then
                E := addMatrix(N, i+1, j, siginv(-M(i,j)))
                M := changeBase(M, E, sig, der)
                B := B*E
                Binv := inv(E)*Binv
            i := i + 1
        else
            -- apply lemma 6
            for j in i..2 by -1 repeat
                for k in (i+1)..N repeat
                    E := addMatrix(N, k, j-1, M(k,j))
                    M := changeBase(M, E, sig, der)
                    B := B*E
                    Binv := inv(E)*Binv
            j := i + 1
        while j <= N and M(j,1) = 0 repeat j := j + 1
        if j <= N then
            -- expand companionblock by lemma 8
            E := permutationMatrix(N, 1, j)
            M := changeBase(M, E, sig, der)
            B := B*E
            Binv := E*Binv
    -- start again to establish rational form
i := 1
else
  -- split a direct factor
  recOfMatrices :=
    normalForm(subMatrix(M, i+1, N, i+1, N), sig, der)
  setSubMatrix!(M, i+1, i+1, recOfMatrices.R)
  E := diagonalMatrix [1 for k in 1..N]
  setSubMatrix!(E, i+1, i+1, recOfMatrices.A)
  B := B*E
  setSubMatrix!(E, i+1, i+1, recOfMatrices.Ainv)
  Binv := E*Binv
  -- M in blockdiagonalform, stop program
  i := N
  [M, B, Binv]

mulMatrix(N, i, a) ==
  M : Matrix K := diagonalMatrix [1 for j in 1..N]
  M(i, i) := a
  M

addMatrix(N, i, k, a) ==
  A : Matrix K := diagonalMatrix [1 for j in 1..N]
  A(i, k) := a
  A

permutationMatrix(N, i, k) ==
  P : Matrix K := diagonalMatrix [1 for j in 1..N]
  P(i, i) := P(k, k) := 0
  P(i, k) := P(k, i) := 1
  P

package PRS PseudoRemainderSequence

  — PseudoRemainderSequence.input —
PACKAGE PRS PSEUDOREMAINDERSEQUENCE

)set break resume
(sys rm -f PseudoRemainderSequence.output
spool PseudoRemainderSequence.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show PseudoRemainderSequence

--R PseudoRemainderSequence(R: IntegralDomain,polR: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for PseudoRemainderSequence is PRS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for PRS

--R Operations --------------------------------------
--R discriminant : polR -> R  resultanat : (polR,polR) -> R
--R ?*? : (R,Vector(polR)) -> Vector(polR)
--R resultantnaif : (polR,polR) -> R
--R resultantEuclideannaif : (polR,polR) -> Record(coef1: polR,coef2: polR,resultant: R)
--R resultant : (polR,polR) -> R
--R resultantEuclidean : (polR,polR) -> Record(coef1: polR,coef2: polR,resultant: R)
--R resultantEuclideannaif : (polR,polR) -> Record(coef1: polR,coef2: polR,resultant: R)
--R resultantReduit : (polR,polR) -> R if R has GCDDOM
--R resultantReduitEuclidean : (polR,polR) -> Record(coef1: polR,coef2: polR,resultantReduit: R) if R has GCDDOM
--R resultantReduitEuclideannaif : (polR,polR) -> Record(coef1: polR,coef2: polR,resultantReduit: R)
--R resultantReduitEuclideannaif : (polR,polR) -> Record(coef1: polR,coef2: polR,resultantReduit: R)
--R Lazard : (R,R,NonNegativeInteger) -> R
--R Lazard2 : (polR,polR,R,NonNegativeInteger) -> polR
--R chainSubResultants : (polR,polR) -> List(polR)
--R degreeSubResultant : (polR,polR,NonNegativeInteger) -> polR
--R degreeSubResultantEuclidean : (polR,polR,NonNegativeInteger) -> Record(coef1: polR,coef2: polR,subResultant: polR)
--R divide : (polR,polR) -> Record(quotient: polR,remainder: polR)
--R exquo : (Vector(polR),R) -> Vector(polR)
--R gcd : (polR,polR) -> polR if R has GCDDOM
--R indiceSubResultant : (polR,polR,NonNegativeInteger) -> polR
--R indiceSubResultantEuclidean : (polR,polR,NonNegativeInteger) -> Record(coef1: polR,coef2: polR,subResultant: polR)
--R lastSubResultant : (polR,polR) -> polR
--R lastSubResultantEuclidean : (polR,polR) -> Record(coef1: polR,coef2: polR,subResultant: polR)
--R nextsousResultant2 : (polR,polR,polR,R) -> polR
--R pseudoDivide : (polR,polR) -> Record(coef1: R,quotient: polR,remainder: polR)
--R resultantEuclidean : (polR,polR) -> Record(coef1: polR,coef2: polR,resultant: R)
--R resultantEuclideannaif : (polR,polR) -> Record(coef1: polR,coef2: polR,resultant: R)
--R resultantReduit : (polR,polR) -> R if R has GCDDOM
--R resultantReduitEuclidean : (polR,polR) -> Record(coef1: polR,coef2: polR,resultantReduit: R) if R has GCDDOM
--R schema : (polR,polR) -> List(NonNegativeInteger)
--R semiDegreeSubResultantEuclidean : (polR,polR,NonNegativeInteger) -> Record(coef2: polR,subResultant: polR)
--R semiDiscriminantEuclidean : polR -> Record(coef2: polR,discriminant: R)
--R semiIndiceSubResultantEuclidean : (polR,polR,NonNegativeInteger) -> Record(coef2: polR,subResultant: polR)
--R semiLastSubResultantEuclidean : (polR,polR) -> Record(coef2: polR,subResultant: polR)
--R semiResultantEuclidean1 : (polR,polR) -> Record(coef1: polR,resultant: R)
--R semiResultantEuclidean2 : (polR,polR) -> Record(coef2: polR,resultant: R)
--R semiResultantEuclideannaif : (polR,polR) -> Record(coef2: polR,resultant: R)
--R semiResultantReduitEuclidean : (polR,polR) -> Record(coef2: polR,resultantReduit: R) if R has GCDDOM
--R semiSubResultantGcdEuclidean1 : (polR,polR) -> Record(coef1: polR,gcd: polR)
--R semiSubResultantGcdEuclidean2 : (polR,polR) -> Record(coef2: polR,gcd: polR)
--R subResultantGcd : (polR,polR) -> polR
--R subResultantGcdEuclidean : (polR,polR) -> Record(coef1: polR,coef2: polR,gcd: polR)
--R
--E 1

)spool
)lisp (bye)

——

— PseudoRemainderSequence.help —

================================================================================
PseudoRemainderSequence examples
================================================================================

The package constructor PseudoRemainderSequence provides
efficient algorithms by Lionel Ducos (University of Poitiers, France)
for computing sub-resultants. This leads to a speed up in many places
in Axiom where sub-resultants are computed (polynomial system solving,
algebraic factorization, integration).

This package contains some functions from improvements of the
subresultants algorithm.

See Also:
o )show PseudoRemainderSequence

——

PseudoRemainderSequence (PRS)

Exports:
### Package PRS PSEUOREMAINDESEQUENCE

- **discriminant**: Lazard
- **Lazard2**: chainSubResultants
- **degreeSubResultant**: degreeSubResultantsEuclidean
- **discriminantEuclidean**: divide
- **exquo**: gcd
- **indiceSubResultant**: indiceSubResultantsEuclidean
- **lastSubResultant**: lastSubResultantsEuclidean
- **nextsousResultant2**: pseudoDivide
- **resultant**: resultantEuclidean
- **resultantEuclideannaif**: resultantnaif
- **resultantReduit**: resultantReduitEuclidean
- **schema**: semiDegreeSubResultantsEuclidean
- **semiDiscriminantEuclidean**: semiIndiceSubResultantsEuclidean
- **semiLastSubResultantsEuclidean**: semiResultantEuclidean
- **semiResultantEuclidean2**: semiResultantEuclideannaif
- **semiResultantReduitEuclidean**: semiSubResultantGcdEuclideannaif
- **semiSubResultantGcdEuclideannaif**: subResultantGcd
- **subResultantGcdEuclideannaif**: ?

---

```lisp
)abbrev package PRS PseudoRemainderSequence
++ Author: Ducos Lionel (Lionel.Ducos@mathlabo.univ-poitiers.fr)
++ Date Created: january 1995
++ Date Last Updated: 5 february 1999
++ References :
++ Lionel Ducos ”Optimizations of the subresultant algorithm”
++ Description:
++ This package contains some functions: discriminant, resultant,
++ subResultantGcd, chainSubResultants, degreeSubResultants, lastSubResultants,
++ resultantEuclideannaif, semiSubResultantGcdEuclideannaif
++ These procedures come from improvements of the subresultants algorithm.

PseudoRemainderSequence(R, polR) : Specification == Implementation where
R : IntegralDomain
polR : UnivariatePolynomialCategory(R)
NNI == NonNegativeInteger
LC == leadingCoefficient

Specification == with
resultant : (polR, polR) -> R
++ \axiom{(resultant(P, Q))} returns the resultant
++ of \axiom{P} and \axiom{Q}

resultantEuclideannaif : (polR, polR) -> Record(coef1 : polR, coef2 : polR, resultant : R)
++ \axiom{(resultantEuclidean(P, Q))} carries out the equality

---
```
++ \texttt{semiResultantEuclidean2 : (polR, polR) -> Record(coef2 : polR, resultant : R)}
++ \texttt{semiResultantEuclidean2(P,Q)} carries out the equality
++ \texttt{...P + coef2*Q = resultant(P,Q)}.
++ Warning. \texttt{axiom(degree(P) \geq degree(Q)}.

++ \texttt{semiResultantEuclidean1 : (polR, polR) -> Record(coef1 : polR, resultant : R)}
++ \texttt{semiResultantEuclidean1(P,Q)} carries out the equality
++ \texttt{coef1.P + ? Q = resultant(P,Q)}.

++ \texttt{indiceSubResultant : (polR, polR, NNI) -> polR}
++ \texttt{indiceSubResultant(P, Q, i)} returns
++ the subresultant of indice \texttt{axiom(i)}

++ \texttt{indiceSubResultantEuclidean : (polR, polR, NNI) -> Record(coef1 : polR, coef2 : polR, subResultant : polR)}
++ \texttt{indiceSubResultant(P, Q, i) returns}
++ the subresultant \texttt{axiom(S_i(P,Q))} and carries out the equality
++ \texttt{axiom(coef1*P + coef2*Q = S_i(P,Q)}
++ Warning. \texttt{axiom(degree(P) \geq degree(Q)}. 

++ \texttt{indiceSubResultantEuclidean : (polR, polR, NNI) -> Record(coef1 : polR, coef2 : polR, subResultant : polR)}
++ \texttt{indiceSubResultant(P, Q, i) returns}
++ the subresultant \texttt{axiom(S_i(P,Q))} and carries out the equality
++ \texttt{axiom(coef1*P + coef2*Q = S_i(P,Q)}/
++ \axiom{\text{Warning. } \text{axiom(degree(P) \geq degree(Q)}}.

++ \texttt{degreeSubResultant : (polR, polR, NNI) -> polR}
++ \texttt{degreeSubResultant(P, Q, d)} computes
++ a subresultant of degree \texttt{axiom(d)}.

++ \texttt{degreeSubResultantEuclidean : (polR, polR, NNI) -> Record(coef1 : polR, coef2 : polR, subResultant : polR)}
++ \texttt{degreeSubResultant(P, Q, i) returns}
++ a subresultant \texttt{axiom(S)} of degree \texttt{axiom(d)}
++ and carries out the equality \texttt{axiom(coef1*P + coef2*Q = S_i)}.

++ \texttt{degreeSubResultantEuclidean : (polR, polR, NNI) -> Record(coef1 : polR, coef2 : polR, subResultant : polR)}
++ \texttt{degreeSubResultant(P, Q, i) returns}
++ a subresultant \texttt{axiom(S)} of degree \texttt{axiom(d)}
++ and carries out the equality \texttt{axiom(...P + coef2*Q = S_i)}.
++ \texttt{Warning. } \texttt{axiom(degree(P) \geq degree(Q)}}.

++ \texttt{lastSubResultant : (polR, polR) -> polR}
++ \texttt{lastSubResultant(P, Q)} computes
++ the last non zero subresultant of \texttt{axiom(P)} and \texttt{axiom(Q)}
lastSubResultantEuclidean : (polR, polR) ->
    Record(coef1 : polR, coef2 : polR, subResultant : polR)

++ \texttt{lastSubResultantEuclidean(P, Q)} computes
++ the last non zero subresultant \texttt{axiom(S)}
++ and carries out the equality \texttt{axiom(coef1*P + coef2*Q = S)}.

semiLastSubResultantEuclidean : (polR, polR) ->
    Record(coef2 : polR, subResultant : polR)

++ \texttt{semiLastSubResultantEuclidean(P, Q)} computes
++ the last non zero subresultant \texttt{axiom(S)}
++ and carries out the equality \texttt{axiom(P + coef2*Q = S)}.
++ Warning. \texttt{axiom(degree(P) >= degree(Q))}.

subResultantGcd : (polR, polR) -> polR
++ \texttt{axiom(subResultantGcd(P, Q))} returns the gcd
++ of two primitive polynomials \texttt{axiom(P)} and \texttt{axiom(Q)}.

subResultantGcdEuclidean : (polR, polR) -> Record(coef1 : polR, coef2 : polR, gcd : polR)
++ \texttt{axiom(subResultantGcdEuclidean(P,Q))} carries out the equality
++ \texttt{axiom(coef1*P + coef2*Q = +/- S_i(P,Q))}
++ where the degree (not the indice)
++ of the subresultant \texttt{axiom(S_i(P,Q))} is the smaller as possible.

semiSubResultantGcdEuclidean2 : (polR, polR) -> Record(coef2 : polR, gcd : polR)
++ \texttt{semiSubResultantGcdEuclidean2(P,Q)} carries out the equality
++ \texttt{axiom(P + coef2*Q = +/- S_i(P,Q))}
++ where the degree (not the indice)
++ of the subresultant \texttt{axiom(S_i(P,Q))} is the smaller as possible.
++ Warning. \texttt{axiom(degree(P) >= degree(Q))}.

semiSubResultantGcdEuclidean1 : (polR, polR) -> Record(coef1 : polR, gcd : polR)
++ \texttt{semiSubResultantGcdEuclidean1(P,Q)} carries out the equality
++ \texttt{axiom(coef1*P + Q = +/- S_i(P,Q))}
++ where the degree (not the indice)
++ of the subresultant \texttt{axiom(S_i(P,Q))} is the smaller as possible.

discriminant : polR -> R
++ \texttt{axiom(discriminant(P, Q))} returns the discriminant
++ of \texttt{axiom(P)} and \texttt{axiom(Q)}.

discriminantEuclidean : polR ->
    Record(coef1 : polR, coef2 : polR, discriminant : R)

++ \texttt{axiom(discriminantEuclidean(P))} carries out the equality
++ \texttt{axiom(coef1 * P + coef2 * D(P) = discriminant(P))}.

semiDiscriminantEuclidean : polR ->
    Record(coef2 : polR, discriminant : R)
++ \texttt{discriminantEuclidean}(P) \text{ carries out the equality}
++ \texttt{...P + coef2 \ast D(P) = discriminant(P)}.
++ \texttt{Warning. degree(P) \geq degree(Q)}.

\texttt{chainSubResultants : (polR, polR) \to List(polR)}
++ \texttt{chainSubResultants(P, Q)} \text{ computes the list}
++ of non zero subresultants of \texttt{axiom(P)} and \texttt{axiom(Q)}.

\texttt{schema : (polR, polR) \to List(NNI)}
++ \texttt{schema(P,Q)} \text{ returns the list of degrees of}
++ non zero subresultants of \texttt{axiom(P)} and \texttt{axiom(Q)}.

\text{if R has GcdDomain then}
\texttt{resultantReduit : (polR, polR) \to R}
++ \texttt{resultantReduit(P, Q)} \text{ returns the "reduce resultant"}
++ of \texttt{axiom(P)} and \texttt{axiom(Q)}.

\texttt{resultantReduitEuclidean : (polR, polR) \to}
\texttt{Record(coef1 : polR, coef2 : polR, resultantReduit : R)}
++ \texttt{resultantReduitEuclidean(P, Q)} \text{ returns}
++ the "reduce resultant" and carries out the equality
++ \texttt{...P + coef2 \ast Q = resultantReduit(P, Q)}.

\texttt{semiResultantReduitEuclidean : (polR, polR) \to}
\texttt{Record(coef2 : polR, resultantReduit : R)}
++ \texttt{semiResultantReduitEuclidean(P, Q)} \text{ returns}
++ the "reduce resultant" and carries out the equality
++ \texttt{...P + coef2 \ast Q = resultantReduit(P, Q)}.

\texttt{gcd : (polR, polR) \to polR}
++ \texttt{gcd(P, Q)} \text{ returns the gcd of \texttt{axiom(P)} and \texttt{axiom(Q)}}.

\text{-- sub-routines exported for convenience -----------------------------}

\texttt{"*" : (R, Vector(polR)) \to Vector(polR)}
++ \texttt{r \ast v} \text{ computes the product of \texttt{axiom(r)} and \texttt{axiom(v)}

\texttt{"exquo" : (Vector(polR), R) \to Vector(polR)}
++ \texttt{v \ exquo r} \text{ computes}
++ the exact quotient of \texttt{axiom(v)} by \texttt{axiom(r)}

\texttt{pseudoDivide : (polR, polR) \to}
\texttt{Record(coef:R, quotient:polR, remainder:polR)}
++ \texttt{pseudoDivide(P, Q)} \text{ computes the pseudoDivide}
++ of \texttt{axiom(P)} by \texttt{axiom(Q)}.

\texttt{divide : (polR, polR) \to Record(quotient : polR, remainder : polR)}
++ \texttt{divide(F,G)} \text{ computes quotient and rest}
++ of the exact euclidean division of \texttt{axiom(F)} by \texttt{axiom(G)}.
Lazard : (R, R, NNI) -> R
++ \texttt{axiom}(Lazard(x, y, n)) computes \texttt{axiom} \(x^n/y^{(n-1)}\)

Lazard2 : (polR, R, R, NNI) -> polR
++ \texttt{axiom}(Lazard2(F, x, y, n)) computes \texttt{axiom} \((x/y)^{(n-1)} \cdot F\)

next_sousResultant2 : (polR, polR, polR, R) -> polR
++ \texttt{axiom}(nextsousResultant2(P, Q, Z, s)) returns
++ the subresultant \texttt{axiom} \(S_{e-1}\) where
++ \texttt{axiom} \(P = S_d, Q = S_{d-1}, Z = S_e, s = \text{lc}(S_d)\)

resultant_naif : (polR, polR) -> R
++ \texttt{axiom}(resultantEuclidean_naif(P,Q)) returns
++ the resultant of \texttt{axiom} \(P\) and \texttt{axiom} \(Q\) computed
++ by means of the naive algorithm.

resultantEuclidean_naif : (polR, polR) ->
++ \texttt{Record(coef1 : polR, coef2 : polR, resultant : R})
++ \texttt{axiom}(resultantEuclidean_naif(P,Q)) returns
++ the extended resultant of \texttt{axiom} \(P\) and \texttt{axiom} \(Q\) computed
++ by means of the naive algorithm.

semiResultantEuclidean_naif : (polR, polR) ->
++ \texttt{Record(coef2 : polR, resultant : R})
++ \texttt{axiom}(resultantEuclidean_naif(P,Q)) returns
++ the semi-extended resultant of \texttt{axiom} \(P\) and \texttt{axiom} \(Q\) computed
++ by means of the naive algorithm.

Implementation == add
X : polR := monomial(1$R,1)

r : R * v : Vector(polR) == r::polR * v
-- the instruction \texttt{map}(r * \#1, v) is slower !?

v : Vector(polR) exquo r : R ==
map((p1:polR):polR +-> (p1 exquo r)::polR, v)
pseudoDivide(P : polR, Q : polR) :
++ computes the pseudoDivide of \texttt{P} by \texttt{Q}
++ zero?(Q) => error("PseudoDivide$PRS : division by 0")
++ zero?(P) => construct(1, 0, P)
1cQ : R := LC(Q)
(degP, degQ) := (degree(P), degree(Q))
if degP < degQ => construct(1, 0, P)
Q := reductum(Q)
i : NNI := (degP - degQ + 1)::NNI
co : R := 1cQ**i
quot : polR := 0$polR
while (delta : Integer := degree(P) - degQ) >= 0 repeat
divide(F: polR, G: polR): Record(quotient: polR, remainder: polR) ==
-- computes quotient and rest of the exact euclidean division of F by G
lcG : R := LC(G)
degG : NNI := degree(G)
zero?(degG) => (F := (F exquo lcG)::polR; return construct(F, 0))
G : polR := reductum(G)
quot : polR := 0
while (delta := degree(F) - degG) >= 0 repeat
  mon : polR := monomial((LC(F) exquo lcG)::R, delta::NNI)
  quot := quot + mon
  F := reductum(F) - mon * G
return construct(quot, F)

resultant_naif(P: polR, Q: polR): R ==
-- valid over a field
a : R := 1
repeat
  zero?(Q) => return 0
  (degP, degQ) := (degree(P), degree(Q))
  if odd?(degP) and odd?(degQ) then a := -a
  zero?(degQ) => return (a * LC(Q)**degP)
  U : polR := divide(P, Q).remainder
  a := a * LC(Q)**(degP - degree(U))::NNI
  (P, Q) := (Q, U)

-- valid over a field.
  a : R := 1
  old_cf1 : polR := 1 ; cf1 : polR := 0
  old_cf2 : polR := 0 ; cf2 : polR := 1
  repeat
    zero?(Q) => construct(0::polR, 0::polR, 0::R)
    (degP, degQ) := (degree(P), degree(Q))
    if odd?(degP) and odd?(degQ) then a := -a
    if zero?(degQ) then
      a := a * LC(Q)**(degP-1)::NNI
      return construct(a*cf1, a*cf2, a*LC(Q))
    divid := divide(P, Q)
    a := a * LC(Q)**(degP - degree(divid.remainder))::NNI
    (P, Q) := (Q, divid.remainder)
    (old_cf1, old_cf2, cf1, cf2) := (cf1, cf2, old_cf1 - divid.quotient * cf1, old_cf2 - divid.quotient * cf2)
semiResultantEuclidean_naif(P : polR, Q : polR) : Record(coef2 : polR, resultant : R) ==

-- valid over a field
a : R := 1
old_cf2 : polR := 0 ; cf2 : polR := 1
repeat
  zero?(Q) => construct(0::polR, 0::R)
  (degP, degQ) := (degree(P), degree(Q))
  if odd?(degP) and odd?(degQ) then a := -a
  if zero?(degQ) then
    a := a * LC(Q)**(degP-1)::NNI
    return construct(a*cf2, a*LC(Q))
  divid := divide(P,Q)
  a := a * LC(Q)**(degP - degree(divid.remainder))::NNI
  (P, Q) := (Q, divid.remainder)
  (old_cf2, cf2) := (cf2, old_cf2 - divid.quotient * cf2)

Lazard(x : R, y : R, n : NNI) : R ==

zero?(n) => error("Lazard$PRS : n = 0")
-- one?(n) => x
(n = 1) => x
a : NNI := 1
while n >= (b := 2*a) repeat a := b
   c : R := x
   n := (n - a)::NNI
end repeat
   -- c = x**i / y**(i-1), i=n_0 quo a, a=2**?
   -- one?(a) => return c
   (a = 1) => return c
   a := a quo 2
   c := ((c * c) exquo y)::R
   if n >= a then ( c := ((c * x) exquo y)::R ; n := (n - a)::NNI )

Lazard2(F : polR, x : R, y : R, n : NNI) : polR ==

zero?(n) => error("Lazard2$PRS : n = 0")
-- one?(n) => F
(n = 1) => F
x := Lazard(x, y, (n-1)::NNI)
return ((x * F) exquo y)::polR

Lazard3(V : Vector(polR), x : R, y : R, n : NNI) : Vector(polR) ==
-- computes x**(n-1) * V / y**(n-1)
zero?(n) => error("Lazard3$prs : n = 0")
-- one?(n) => V
(n = 1) => V
x := Lazard(x, y, (n-1)::NNI)
return ((x * V) exquo y)

(1cP, c, se) := (LC(P), LC(Q), LC(Z))
(d, e) := (degree(P), degree(Q))
(P, Q, H) := (reductum(P), reductum(Q), - reductum(Z))
A : polR := coefficient(P, e) * H
for i in e+1..d-1 repeat
  H := if degree(H) = e-1 then
    X * reductum(H) - ((LC(H) * Q) exquo c)::polR
  else
    X * H
  -- H = s_e * X^i mod S_d-1
  A := coefficient(P, i) * H + A
while degree(P) >= e repeat P := reductum(P)
A := A + se * P -- A = s_e * reductum(P_0) mod S_d-1
A := (A exquo lcP)::polR -- A = s_e * reductum(S_d) / s_d mod S_d-1
A := if degree(H) = e-1 then
  c * (X * reductum(H) + A) - LC(H) * Q
else
  c * (X * H + A)
A := (A exquo s)::polR -- A = +/- S_e-1
return (if odd?(d-e) then A else - A)

next_sousResultant3(VP : Vector(polR), VQ : Vector(polR), s : R, ss : R) :
  Vector(polR) ==
  -- P ~ S_d, Q = S_d-1, s = lc(S_d), ss = lc(S_e)
  (P, Q) := (VP.1, VQ.1)
  (lcP, c) := (LC(P), LC(Q))
  e : NNI := degree(Q)
  if one?(delta := degree(P) - e) then -- algo_new
    if ((delta := degree(P) - e) = 1) then -- algo_new
      VP := c * VP - coefficient(P, e) * VQ
      VP := VP exquo lcP
      VP := c * (VP - X * VQ) + coefficient(Q, (e-1)::NNI) * VQ
      VP := VP exquo s
    else -- algorithm of Lickteig - Roy
      if odd?(delta) then VP else - VP
  else -- algorithm of Lickteig - Roy
    (r, rr) := (s * lcP, ss * c)
    divid := divide(rr * P, Q)
    VP.1 := (divid.remainder exquo r)::polR
    for i in 2..#VP repeat
      VP.i := rr * VP.i - VQ.i * divid.quotient
      VP.i := (VP.i exquo r)::polR
    return (if odd?(delta) then VP else - VP)

algo_new(P : polR, Q : polR) : R ==
  delta : NNI := (degree(P) - degree(Q))::NNI
  s : R := LC(Q)**delta
  (P, Q) := (Q, pseudoRemainder(P, -Q))
  repeat
    -- P = S_c-1 (except the first turn : P ~ S_c-1),
    -- Q = S_d-1,  s = lc(S_d)
    zero?(Q) => return 0
    delta := (degree(P) - degree(Q))::NNI
Z : polR := Lazard2(Q, LC(Q), s, delta)
-- Z = S_e \sim S_{d-1}
zero?(degree(Z)) => return LC(Z)
(P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

resultant(P : polR, Q : polR) : R ==
zero?(Q) or zero?(P) => 0
if degree(P) < degree(Q) then
  (P, Q) := (Q, P)
if odd?(degree(P)) and odd?(degree(Q)) then Q := - Q
zero?(degree(Q)) => LC(Q)**degree(P)
-- degree(P) >= degree(Q) > 0
R has Finite => resultant_naif(P, Q)
return algo_new(P, Q)

subResultantEuclidean(P : polR, Q : polR) :
  Record(coef1 : polR, coef2 : polR, resultant : R) ==
  s : R := LC(Q)**(degree(P) - degree(Q))::NNI
  VP : Vector(polR) := [Q, 0::polR, 1::polR]
  pdiv := pseudoDivide(P, -Q)
  VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
  repeat
    -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
    -- S_{c-1} = VP.2 P_0 + VP.3 Q_0, S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
    (P, Q) := (VP.1, VQ.1)
    zero?(Q) => return construct(0::polR, 0::polR, 0::R)
    e : NNI := degree(Q)
    delta : NNI := (degree(P) - e)::NNI
    if zero?(e) then
      l : Vector(polR) := Lazard3(VQ, LC(Q), s, delta)
      return construct(1.2, 1.3, LC(1.1))
    ss : R := Lazard(LC(Q), s, delta)
    (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

resultantEuclidean(P : polR, Q : polR) :
  Record(coef1 : polR, coef2 : polR, resultant : R) ==
  zero?(P) or zero?(Q) => construct(0::polR, 0::polR, 0::R)
  if degree(P) < degree(Q) then
    e : Integer := if odd?(degree(P)) and odd?(degree(Q)) then -1 else 1
    l := resultantEuclidean(Q, e * P)
    return construct(e * l.coef2, l.coef1, l.resultant)
  if zero?(degree(Q)) then
    degP : NNI := degree(P)
    zero?(degP) => error("resultantEuclidean$PRS : constant polynomials")
    s := LC(Q)**(degP-1)::NNI
    return construct(0::polR, s::polR, s * LC(Q))
  R has Finite => resultantEuclidean_naif(P, Q)
return subResultantEuclidean(P, Q)
semiSubResultantEuclidean(P : polR, Q : polR) :
    Record(coef2 : polR, resultant : R) ==
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    VP : Vector(polR) := [Q, 1::polR]
    pdiv := pseudoDivide(P, -Q)
    VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
    repeat
        -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
        -- S_{c-1} = ...P_0 + VP.3 Q_0, S_{d-1} = ...P_0 + VQ.3 Q_0
        (P, Q) := (VP.1, VQ.1)
        zero?(Q) => return construct(0::polR, 0::R)
        e : NNI := degree(Q)
        delta : NNI := (degree(P) - e)::NNI
        if zero?(e) then
            l : Vector(polR) := Lazard3(VQ, LC(Q), s, delta)
            return construct(l.2, LC(l.1))
        ss : R := Lazard(LC(Q), s, delta)
        (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
        s := ss

semiResultantEuclidean2(P : polR, Q : polR) :
    Record(coef2 : polR, resultant : R) ==
    zero?(P) or zero?(Q) => construct(0::polR, 0::R)
    degree(P) < degree(Q) => error("semiResultantEuclidean2 : bad degrees")
    if zero?(degree(Q)) then
        degP : NNI := degree(P)
        zero?(degP) => error("semiResultantEuclidean2 : constant polynomials")
        s : R := LC(Q)**(degP-1)::NNI
        return construct(s::polR, s * LC(Q))
    R has Finite => semiResultantEuclidean_naif(P, Q)
    return semiSubResultantEuclidean(P,Q)

semiResultantEuclidean1(P : polR, Q : polR) :
    Record(coef1 : polR, resultant : R) ==
    result := resultantEuclidean(P,Q)
    [result.coef1, result.resultant]

indiceSubResultant(P : polR, Q : polR, i : NNI) : polR ==
    zero?(Q) or zero?(P) => 0
    if degree(P) < degree(Q) then
        (P, Q) := (Q, P)
    if odd?(degree(P)-i) and odd?(degree(Q)-i) then Q := - Q
    if i = degree(Q) then
        delta : NNI := (degree(P)-degree(Q))::NNI
        zero?(delta) => error("indiceSubResultant$PRS : bad degrees")
        s : R := LC(Q)**(delta-1)::NNI
        return s*Q
    i > degree(Q) => 0
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
  -- P = S_{c-1} - S_d, Q = S_{d-1}, s = lc(S_d), i < d
  (degP, degQ) := (degree(P), degree(Q))
  i = degP-1 => return Q
  zero?(Q) or (i > degQ) => return 0
  Z := polR := Lazard2(Q, LC(Q), s, (degP - degQ)::NNI)
  -- Z = S_e ~ S_d-1
  i = degQ => return Z
  (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
  s := LC(Z)

indiceSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
  Record(coef1 : polR, coef2 : polR, subResultant : polR) ==
  zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
  if degree(P) < degree(Q) then
    e := if odd?(degree(P)-i) and odd?(degree(Q)-i) then -1 else 1
    l := indiceSubResultantEuclidean(Q, e * P, i)
    return construct(e * l.coef2, l.coef1, l.subResultant)
  if i = degree(Q) then
    delta : NNI := (degree(P)-degree(Q))::NNI
    zero?(delta) =>
      error("indiceSubResultantEuclidean$PRS : bad degrees")
    s := LC(Q)**(delta-1)::NNI
    return construct(0::polR, s::polR, s * Q)
  if i > degree(Q) => construct(0::polR, 0::polR, 0::polR)
  s := LC(Q)**(degree(P) - degree(Q))::NNI
  VP := Vector(polR) := [Q, 0::polR, 1::polR]
  pdiv := pseudoDivide(P, -Q)
  VQ := Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
  repeat
    -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d), i < d
    -- S_{c-1} = VP.2 P_0 + VP.3 Q_0, S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
    (P, Q) := (VP.1, VQ.1)
    zero?(Q) => return construct(0::polR, 0::polR, 0::polR)
    (degP, degQ) := (degree(P), degree(Q))
    i = degP-1 => return construct(VQ.2, VQ.3, VQ.1)
    (i > degQ) => return construct(0::polR, 0::polR, 0::polR)
    VZ := Lazard3(VQ, LC(Q), s, (degP - degQ)::NNI)
    i = degQ => return construct(VZ.2, VZ.3, VZ.1)
    ss := LC(VZ.1)
    (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
    s := ss

semiIndiceSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
  Record(coef2 : polR, subResultant : polR) ==
  zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
  if degree(P) < degree(Q) then
    error("semiIndiceSubResultantEuclidean$PRS : bad degrees")
delta : NNI := (degree(P)-degree(Q))::NNI
zero?(delta) =>
    error("semiIndiceSubResultantEuclidean$PRS : bad degrees")
s : R := LC(Q)**(delta-1)::NNI
return construct(s::polR, s * Q)
i > degree(Q) => construct(0::polR, 0::polR)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
repeat
    -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s = lc(S_d), i < d
    -- S_{c-1} = ...P_0 + VP.2 Q_0, S_{d-1} = ...P_0 + ...Q_0
    (P, Q) := (VP.1, VQ.1)
    zero?(Q) => return construct(0::polR, 0::polR)
    (degP, degQ) := (degree(P), degree(Q))
i = degP-1 => return construct(VQ.2, VQ.1)
    (i > degQ) => return construct(0::polR, 0::polR)
    VZ := Lazard3(VQ, LC(Q), s, (degP - degQ)::NNI)
i = degQ => return construct(VZ.2, VZ.1)
    ss : R := LC(VZ.1)
    (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

degreeSubResultant(P : polR, Q : polR, i : NNI) : polR ==
    zero?(Q) or zero?(P) => 0
    if degree(P) < degree(Q) then (P, Q) := (Q, P)
    if i = degree(Q) then
        delta : NNI := (degree(P)-degree(Q))::NNI
        zero?(delta) => error("degreeSubResultant$PRS : bad degrees")
s : R := LC(Q)**(delta-1)::NNI
        return s*Q
    i > degree(Q) => 0
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    (P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
    -- P = S_{c-1}, Q = S_{d-1}, s = lc(S_d)
    zero?(Q) or (i > degree(Q)) => return 0
    i = degree(Q) => return Q
    Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
    -- Z = S_e ~ S_d-1
    (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

    zero?(Q) or zero?(P) => construct(0::polR, 0::polR, 0::polR)
    if degree(P) < degree(Q) then
        l := degreeSubResultantEuclidean(Q, P, i)
        return construct(l.coef2, l.coef1, 1.subResultant)
if i = degree(Q) then
  delta : NNI := (degree(P)-degree(Q))::NNI
  zero?(delta) =>
    error("degreeSubResultantEuclidean$PRS : bad degrees")
  s : R := LC(Q)**(delta-1)::NNI
  return construct(0::polR, s::polR, s * Q)
i > degree(Q) => construct(0::polR, 0::polR, 0::polR)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 0::polR, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
  -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
  -- S_{c-1} = ...P_0 + VP.3 Q_0, S_{d-1} = ...P_0 + VQ.3 Q_0
  (P, Q) := (VP.1, VQ.1)
  zero?(Q) or (i > degree(Q)) =>
    return construct(0::polR, 0::polR, 0::polR)
i = degree(Q) => return construct(VQ.2, VQ.3, VQ.1)
ss : R := Lazard(LC(Q), s, (degree(P)-degree(Q))::NNI)
(VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

semiDegreeSubResultantEuclidean(P : polR, Q : polR, i : NNI) :
  Record(coef2 : polR, subResultant : polR) ==
  zero?(Q) or zero?(P) => construct(0::polR, 0::polR)
degree(P) < degree(Q) =>
  error("semiDegreeSubResultantEuclidean$PRS : bad degrees")
if i = degree(Q) then
  delta : NNI := (degree(P)-degree(Q))::NNI
  zero?(delta) =>
    error("semiDegreeSubResultantEuclidean$PRS : bad degrees")
  s : R := LC(Q)**(delta-1)::NNI
  return construct(s::polR, s * Q)
i > degree(Q) => construct(0::polR, 0::polR)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
  -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=lc(S_d)
  -- S_{c-1} = ...P_0 + VP.3 Q_0, S_{d-1} = ...P_0 + VQ.3 Q_0
  (P, Q) := (VP.1, VQ.1)
  zero?(Q) or (i > degree(Q)) =>
    return construct(0::polR, 0::polR)
i = degree(Q) => return construct(VQ.2, VQ.3, VQ.1)
ss : R := Lazard(LC(Q), s, (degree(P)-degree(Q))::NNI)
(VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

lastSubResultant(P : polR, Q : polR) : polR ==
zero?(Q) or zero?(P) \implies 0

if degree(P) < degree(Q) then (P, Q) := (Q, P)

zero?(degree(Q)) \implies (LC(Q)**degree(P))::polR

s : R := LC(Q)**(degree(P) - degree(Q))::NNI

(P, Q) := (Q, pseudoRemainder(P, -Q))

Z : polR := P

repeat

-- Z = S_d (except the first turn : Z = P)
-- P = S_{c-1} - S_d, Q = S_{d-1}, s = lc(S_d)

zero?(Q) \implies return Z

Z := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)

-- Z = S_e - S_{d-1}

zero?(degree(Z)) \implies return Z

(P, Q) := (Q, next_sousResultant2(P, Q, Z, s))

s := LC(Z)

lastSubResultantEuclidean(P : polR, Q : polR) :

Record(coef1 : polR, coef2 : polR, subResultant : polR) ==

zero?(Q) or zero?(P) \implies construct(0::polR, 0::polR, 0::polR)

if degree(P) < degree(Q) then

l := lastSubResultantEuclidean(Q, P)

return construct(l.coef2, l.coef1, l.subResultant)

if zero?(degree(Q)) then

degP : NNI := degree(P)

zero?(degP) =>

error("lastSubResultantEuclidean$PRS : constant polynomials")

s : R := LC(Q)**(degP-1)::NNI

return construct(1.coef2, 1.coef1, 1.subResultant)

if zero?(degree(Q)) then

if degP : NNI := degree(P)

zero?(degP) =>

error("lastSubResultantEuclidean$PRS : constant polynomials")

s : R := LC(Q)**(degP-1)::NNI

return construct(1.coef2, 1.coef1, 1.subResultant)

ss : R := LC(VZ.1)

(VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))

s := ss

semiLastSubResultantEuclidean(P : polR, Q : polR) :

Record(coef2 : polR, subResultant : polR) ==

zero?(Q) or zero?(P) \implies construct(0::polR, 0::polR)

degree(P) < degree(Q) =>
error("semilastSubResultantEuclidean$PRS : bad degrees")
if zero?(degree(Q)) then
degP : NNI := degree(P)
zero?(degP) =>
  error("semilastSubResultantEuclidean$PRS : constant polynomials")
s : R := LC(Q)**(degP-1)::NNI
return construct(s::polR, s * Q)

s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
VZ : Vector(polR) := copy(VP)
repeat
  -- VZ.1 = S_d, VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s = lc(S_d)
  -- S_{c-1} = ... P_0 + VP.2 Q_0
  -- S_{d-1} = ... P_0 + VQ.2 Q_0
  -- S_d = ... P_0 + VZ.2 Q_0
  (Q, Z) := (VQ.1, VZ.1)
  zero?(Q) => return construct(VZ.2, VZ.1)
  VZ := Lazard3(VQ, LC(Q), s, (degree(Z) - degree(Q))::NNI)
  zero?(degree(Q)) => return construct(VZ.2, VZ.1)
  ss : R := LC(VZ.1)
  (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
  s := ss

chainSubResultants(P : polR, Q : polR) : List(polR) ==
  zero?(Q) or zero?(P) => []
  if degree(P) = degree(Q) then
    (P, Q) := (Q, P)
    if odd?(degree(P)) and odd?(degree(Q)) then Q := - Q
    L : List(polR) := []
    zero?(degree(Q)) => return L
    L := [Q]
    s : R := LC(Q)**(degree(P) - degree(Q))::NNI
    (P, Q) := (Q, pseudoRemainder(P, -Q))
    repeat
      -- P = S_{c-1}, Q = S_{d-1}, s = lc(S_d)
      -- L = [S_d, ..., S_{q-1}]
      zero?(Q) => return L
      L := concat(Q, L)
      -- L = [S_{d-1}, ..., S_{q-1}]
      delta : NNI := (degree(P) - degree(Q))::NNI
      Z : polR := Lazard2(Q, LC(Q), s, delta)
      if delta > 1 then L := concat(Z, L)
      -- L = [S_e, ..., S_{q-1}]
      zero?(degree(Z)) => return L
      (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
      s := LC(Z)
  schema(P : polR, Q : polR) : List(NNI) ==
zero?(Q) or zero?(P) => []

if degree(P) < degree(Q) then (P, Q) := (Q, P)
zero?(degree(Q)) => [0]
L : List(NNI) := []
(s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, Q))
repeat
   -- P = S_{c-1} - S_d, Q = S_{d-1}, s = 1c(S_d)
   zero?(Q) => return L
   e : NNI := degree(Q)
   L := concat(e, L)
   delta : NNI := (degree(P) - e)::NNI
   Z : polR := Lazard2(Q, LC(Q), s, delta) -- Z = S_{e} ~ S_{d-1}
   if delta > 1 then L := concat(e, L)
   zero?(e) => return L
   (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

subResultantGcd(P : polR, Q : polR) : polR ==
zero?(P) and zero?(Q) => 0
zero?(P) => Q
zero?(Q) => P
if degree(P) < degree(Q) then (P, Q) := (Q, P)
zero?(degree(Q)) => 1$polR
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
(P, Q) := (Q, pseudoRemainder(P, -Q))
repeat
   -- P = S_{c-1}, Q = S_{d-1}, s = 1c(S_d)
   zero?(Q) => return P
   zero?(degree(Q)) => return 1$polR
   Z : polR := Lazard2(Q, LC(Q), s, (degree(P) - degree(Q))::NNI)
   -- Z = S_{e} ~ S_{d-1}
   (P, Q) := (Q, next_sousResultant2(P, Q, Z, s))
s := LC(Z)

subResultantGcdEuclidean(P : polR, Q : polR) :
Record(coef1 : polR, coef2 : polR, gcd : polR) ==
zero?(P) and zero?(Q) => construct(0::polR, 0::polR, 0::polR)
zero?(P) => construct(0::polR, 1::polR, Q)
zero?(Q) => construct(1::polR, 0::polR, P)
if degree(P) < degree(Q) then
   l := subResultantGcdEuclidean(Q, P)
   return construct(l.coef2, l.coef1, l.gcd)
zero?(degree(Q)) => construct(0::polR, 1::polR, Q)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 0::polR, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.coef::polR, pdiv.quotient]
repeat
   -- VP.1 = S_{c-1}, VQ.1 = S_{d-1}, s=1c(S_d)
-- S_{c-1} = VP.2 P_0 + VP.3 Q_0, S_{d-1} = VQ.2 P_0 + VQ.3 Q_0
(P, Q) := (VP.1, VQ.1)
zero?(Q) => return construct(VP.2, VP.3, P)
e : NNI := degree(Q)
zero?(e) => return construct(VQ.2, VQ.3, Q)
ss := Lazard(LC(Q), s, (degree(P) - e)::NNI)
(VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

semiSubResultantGcdEuclidean2(P : polR, Q : polR) :
  Record(coef2 : polR, gcd : polR) ==
  zero?(P) and zero?(Q) => construct(0::polR, 0::polR)
  zero?(P) => construct(1::polR, Q)
  zero?(Q) => construct(0::polR, P)
  degree(P) < degree(Q) =>
    error("semiSubResultantGcdEuclidean2: bad degrees")
  zero?(degree(Q)) => construct(1::polR, Q)
s : R := LC(Q)**(degree(P) - degree(Q))::NNI
VP : Vector(polR) := [Q, 1::polR]
pdiv := pseudoDivide(P, -Q)
VQ : Vector(polR) := [pdiv.remainder, pdiv.quotient]
repeat
  -- P=S_{c-1}, Q=S_{d-1}, s=lc(S_d)
  -- S_{c-1} = ? P_0 + old_cf2 Q_0, S_{d-1} = ? P_0 + cf2 Q_0
  (P, Q) := (VP.1, VQ.1)
  zero?(Q) => return construct(VP.2, P)
e : NNI := degree(Q)
  zero?(e) => return construct(VQ.2, Q)
  ss := Lazard(LC(Q), s, (degree(P) - e)::NNI)
  (VP, VQ) := (VQ, next_sousResultant3(VP, VQ, s, ss))
s := ss

semiSubResultantGcdEuclidean1(P : polR, Q : polR) :
  Record(coef1 : polR, gcd : polR) ==
  result := subResultantGcdEuclidean(P, Q)
  [result.coef1, result.gcd]

discriminant(P : polR) : R ==
d : Integer := degree(P)
  zero?(d) => error "cannot take discriminant of constants"
a := Integer := (d * (d-1)) quo 2
  a := (-1)**a::NonNegativeInteger
dP : polR := differentiate P
r : R := resultant(P, dP)
d := d - degree(dP) - 1
return (if zero?(d) then a * (r exquo LC(P))::R
  else a * r * LC(P)**(d-1)::NNI)

discriminantEuclidean(P : polR) :
  Record(coef1 : polR, coef2 : polR, discriminant : R) ==
d : Integer := degree(P)
zero?(d) => error "cannot take discriminant of constants"
a : Integer := (d * (d-1)) quo 2
a := (-1)**a::NonNegativeInteger
dP : polR := differentiate P
rE := resultantEuclidean(P, dP)
d := d - degree(dP) - 1
if zero?(d) then
  c1 : polR := a * (rE.coef1 exquo LC(P))::polR
c2 : polR := a * (rE.coef2 exquo LC(P))::polR
cr : R := a * (rE.resultant exquo LC(P))::R
else
  c1 : polR := a * rE.coef1 * LC(P)**(d-1)::NNI
c2 : polR := a * rE.coef2 * LC(P)**(d-1)::NNI
cr : R := a * rE.resultant * LC(P)**(d-1)::NNI
return construct(c1, c2, cr)

semiDiscriminantEuclidean(P : polR) :
  Record(coef2 : polR, discriminant : R) ==
d : Integer := degree(P)
zero?(d) => error "cannot take discriminant of constants"
a : Integer := (d * (d-1)) quo 2
a := (-1)**a::NonNegativeInteger
dP : polR := differentiate P
rE := semiResultantEuclidean2(P, dP)
d := d - degree(dP) - 1
if zero?(d) then
  c2 : polR := a * (rE.coef2 exquo LC(P))::polR
cr : R := a * (rE.resultant exquo LC(P))::R
else
  c2 : polR := a * rE.coef2 * LC(P)**(d-1)::NNI
cr : R := a * rE.resultant * LC(P)**(d-1)::NNI
return construct(c2, cr)

if R has GcdDomain then
  resultantReduit(P : polR, Q : polR) : R ==
    UV := subResultantGcdEuclidean(P, Q)
    UVs : polR := UV.gcd
    degree(UVs) > 0 => 0
    l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
    return (LC(UVs) exquo gcd(l))::R

resultantReduitEuclidean(P : polR, Q : polR) :
  Record(coef1 : polR, coef2 : polR, resultantReduit : R) ==
    UV := subResultantGcdEuclidean(P, Q)
    UVs : polR := UV.gcd
    degree(UVs) > 0 => construct(0::polR, 0::polR, 0::R)
    l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
g1 : R := gcd(l)
c1 : polR := (UV.coef1 exquo g1)::polR
c2 : polR := (UV.coef2 exquo gl)::polR
rr : R := (LC(UVs) exquo gl)::R
return construct(c1, c2, rr)

semiResultantReducedEuclidean(P : polR, Q : polR) :
  Record(coef2 : polR, resultantReduced : R) ==
  UV := subResultantGcdEuclidean(P, Q)
  UVs : polR := UV.gcd
  degree(UVs) > 0 => construct(0::polR, 0::R)
  l : List(R) := concat(coefficients(UV.coef1), coefficients(UV.coef2))
  gl : R := gcd(l)
  c2 : polR := (UV.coef2 exquo gl)::polR
  rr : R := (LC(UVs) exquo gl)::R
  return construct(c2, rr)

gcd_naif(P : polR, Q : polR) : polR ==
  -- valid over a field
  zero?(P) => (Q exquo LC(Q))::polR
  repeat
    zero?(Q) => return (P exquo LC(P))::polR
    zero?(degree(Q)) => return 1$polR
    (P, Q) := (Q, divide(P, Q).remainder)
  end repeat

gcd(P : polR, Q : polR) : polR ==
  R has Finite => gcd_naif(P,Q)
  zero?(P) => Q
  zero?(Q) => P
  cP : R := content(P)
  cQ : R := content(Q)
  P := (P exquo cP)::polR
  Q := (Q exquo cQ)::polR
  G : polR := subResultantGcd(P, Q)
  return gcd(cP,cQ) * primitivePart(G)
package INTPAF PureAlgebraicIntegration

--- PureAlgebraicIntegration.input ---

)set break resume
=sys rm -f PureAlgebraicIntegration.output
=spool PureAlgebraicIntegration.output
=set message test on
=set message auto off
=clear all

--S 1 of 1
=show PureAlgebraicIntegration
  --R
  --R PureAlgebraicIntegration(R: Join(GcdDomain,RetractableTo(Integer),OrderedSet,CharacteristicZero),L: SetCategory) is a package constructor
  --R Abbreviation for PureAlgebraicIntegration is INTPAF
  --R This constructor is not exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTPAF
  --R
  --R------------------------------- Operations --------------------------------
  --R palgLODE : (L,F,Kernel(F),Kernel(F),Symbol) -> Record(particular: Union(F,"failed"),basis: L)
  --R palgRDE : (F,F,F,Kernel(F),Kernel(F),((F,F,Symbol) -> Union(F,"failed"))) -> Union(F,"failed")
  --R palgextint : (F,Kernel(F),Kernel(F),F) -> Union(Record(ratpart: F,coeff: F),"failed")
  --R palgint : (F,Kernel(F),Kernel(F)) -> IntegrationResult(F)
  --R palglimint : (F,Kernel(F),Kernel(F),List(F)) -> Union(Record(mainpart: F,limitedlogs: List(R)),"failed")

=E 1
=spool
=)lisp (bye)

--- PureAlgebraicIntegration.help ---

====================================================================
PureAlgebraicIntegration examples
====================================================================

Integration of pure algebraic functions.

This package provides functions for integration, limited integration, extended integration and the risch differential equation for pure algebraic integrands.

See Also:
  o )show PureAlgebraicIntegration
PureAlgebraicIntegration (INTPAF)

Exports:
  palgLODE  palgRDE  palgextint  palgint  palglimint

— package INTPAF PureAlgebraicIntegration —

)abbrev package INTPAF PureAlgebraicIntegration
++ Author: Manuel Bronstein
++ Date Created: 27 May 1988
++ Date Last Updated: 24 June 1994
++ Description:
  ++ Integration of pure algebraic functions;
  ++ This package provides functions for integration, limited integration,
  ++ extended integration and the risch differential equation for
  ++ pure algebraic integrands;

PureAlgebraicIntegration(R, F, L): Exports == Implementation where
  R: Join(GcdDomain,RetractableTo Integer,OrderedSet, CharacteristicZero,
     LinearlyExplicitRingOver Integer)
  F: Join(FunctionSpace R, AlgebraicallyClosedField,
     TranscendentalFunctionCategory)
  L: SetCategory

SY  ==> Symbol
N  ==> NonNegativeInteger
K  ==> Kernel F
P  ==> SparseMultivariatePolynomial(R, K)
UP  ==> SparseUnivariatePolynomial F
RF  ==> Fraction UP
UPUP  ==> SparseUnivariatePolynomial RF
IR  ==> IntegrationResult F
IR2 ==> IntegrationResultFunctions2(curve, F)
ALG ==> AlgebraicIntegrate(R, F, UP, UPUP, curve)
LDALG ==> LinearOrdinaryDifferentialOperator1 curve
RDALG ==> PureAlgebraicLODE(F, UP, UPUP, curve)
LOG ==> Record(coeff:F, logand:F)
REC ==> Record(particular:U1, basis:List F)
CND ==> Record(left:UP, right:UP)
CHV ==> Record(int:UPUP, left:UP, right:UP, den:RF, deg:N)
U1 ==> Union(F, "failed")
U2 ==> Union(Record(ratpart:F, coeff:F),"failed")
U3 ==> Union(Record(mainpart:F, limitedlogs:List LOG), "failed")
FAIL==> error "failed - cannot handle that integrand"

Exports ==> with
  palgint : (F, K, K) -> IR
    ++ palgint(f, x, y) returns the integral of \( f(x,y)dx \)
    ++ where \( y \) is an algebraic function of \( x \).
  palgextint: (F, K, K, F) -> U2
    ++ palgextint(f, x, y, g) returns functions \( \{h, c\} \) such that
    ++ \( dh/dx = f(x,y) - c g \), where \( y \) is an algebraic function of \( x \);
    ++ returns "failed" if no such functions exist.
  palglimint: (F, K, K, List F) -> U3
    ++ palglimint(f, x, y, \{u1,...,un\}) returns functions
    ++ \( \{h,[[ci, ui]]\} \) such that the ui's are among \( \{u1,...,un\} \)
    ++ and \( d(h + \sum(ci \log(ui)))/dx = f(x,y) \) if such functions exist,
    ++ "failed" otherwise;
    ++ \( y \) is an algebraic function of \( x \).
  palgRDE : (F, F, F, K, K, (F, F, SY) -> U1) -> U1
    ++ palgRDE(nfp, f, g, x, y, foo) returns a function \( z(x,y) \)
    ++ such that \( dz/dx + n * df/dx z(x,y) = g(x,y) \) if such a \( z \) exists,
    ++ "failed" otherwise;
    ++ \( y \) is an algebraic function of \( x \);
    ++ \( \{foo(a, b, x)\} \) is a function that solves
    ++ \( du/dx + n * da/dx u(x) = u(x) \)
    ++ for an unknown \( \{spad{u}(x)\} \) not involving \( y \).
    ++ \( \{spad{nfp}\} \) is \( \{spad{n * df/dx}\} \).

if L has LinearOrdinaryDifferentialOperatorCategory F then
  palgLODE: (L, F, K, K, SY) -> REC
    ++ palgLODE(op, g, kx, y, x) returns the solution of \( \text{spad{op}} f = g \).
    ++ \( y \) is an algebraic function of \( x \).

Implementation ==> add
import IntegrationTools(R, F)
import RationalIntegration(F, UP)
import GenusZeroIntegration(R, F, L)
import ChangeOfVariable(F, UP, UPUP)
import IntegrationResultFunctions2(F, F)
import IntegrationResultFunctions2(RF, F)
import SparseUnivariatePolynomialFunctions2(F, RF)
import UnivariatePolynomialCommonDenominator(UP, RF, UPUP)
import PolynomialCategoryQuotientFunctions(IndexedExponents K, K, R, P, F)

quadIfCan : (K, K) -> Union(Record(coef:F, poly:UP), "failed")
linearInXIfCan : (K, K) -> Union(Record(xsub:F, dxsub:RF), "failed")
prootintegrate : (F, K, K) -> IR
prootintegrate1: (UPUP, K, K, UPUP) -> IR
prootextint : (F, K, K, F) -> U2
prootlimint : (F, K, K, List F) -> U3
prootRDE : (F, F, F, K, K, (F, F, SY) -> U1) -> U1
palgRDE1 : (F, F, K, K) -> U1
palgLODE1 : (List F, F, K, K, SY) -> REC
palgintegrate : (F, K, K) -> IR
palgext : (F, K, K, F) -> U2
palglim : (F, K, K, List F) -> U3
UPUP2F1 : (UPUP, RF, RF, K, K) -> F
UPUP2F0 : (UPUP, K, K) -> F
RF2UPUP : (RF, UPUP) -> UPUP
algaddx : (IR, F) -> IR
chvarIfCan : (UPUP, RF, UP, RF) -> Union(UPUP, "failed")
changeVarIfCan : (UPUP, RF, N) -> Union(CHV, "failed")
rationalInt : (UPUP, N, UP) -> IntegrationResult RF
chv : (UPUP, N, F, F) -> RF
chv0 : (UPUP, N, F, F) -> F
candidates : UP -> List CND

dummy := new()$SY
dumk := kernel(dummy)@K

UPUP2F1(p, t, cf, kx, k) == UPUP2F0(eval(p, t, cf), kx, k)
UPUP2F0(p, kx, k) == multivariate(p, kx, k::F)
chv(f, n, a, b) == univariate(chv0(f, n, a, b), dumk)

RF2UPUP(f, modulus) ==
  bc := extendedEuclidean(map((z1:F):RF+->z1::UP::RF, denom f), modulus, 1)::Record(coef1:UPUP, coef2:UPUP)
  (map((x1:F):RF+->x1::UP::RF, numer f) * bc.coef1) rem modulus

-- returns "failed", or (xx, c) such that f(x, y)dx = f(xx, y) c dy
-- if p(x, y) = 0 is linear in x
linearInXIfCan(x, y) ==
a := b := 0$UP
p := clearDenominator lift(minPoly y, x)
while p "= 0 repeat
  degree(q := numer leadingCoefficient p) > 1 => return "failed"
  a := a + monomial(coefficient(q, 1), d := degree p)
  b := b - monomial(coefficient(q, 0), d)
  p := reductum p
xx:RF := b / a
[xx(dumk::F), differentiate(xx, differentiate)]
-- return \(\int(f(x,y) \, dx)\) where \(y\) is an \(n^{th}\) root of a rational function in \(x\)

\[
\text{prootintegrate}(f, x, y) ==
\]

\[
\text{modulus} := \text{lift}(p := \text{minPoly} \, y, \, x)
\]

\[
\text{rf} := \text{reductum}(\text{ff} := \text{univariate}(f, \, x, \, y, \, p))
\]

\[
\begin{cases}
(r := \text{retractIfCan}(\text{rf})@\text{Union}(\text{RF}, "failed")) \text{ case RF} \land \text{rf} = \not 0 \Rightarrow \\
\text{-- in this case, } \text{ff} := \text{lc}(\text{ff}) \, y^i + r \text{ so we integrate both terms} \\
\text{-- separately to gain time}
\end{cases}
\]

\[
\text{map}(\text{ff}+\rightarrow f(\cdot:x::F), \integrate(r::\text{RF})) +
\]

\[
\text{prootintegrate1}(\text{leadingMonomial ff, x, y, modulus})
\]

\[
\text{prootintegrate1}(ff, x, y, modulus) ==
\]

\[
\text{chv:CHV}
\]

\[
r := \text{radPoly}(\text{modulus})::\text{Record}((\text{radicand}:\text{RF}, \, \text{deg}:N))
\]

\[
(\text{uu := changeVarIfCan}(ff, \, r.\text{radicand}, \, r.\text{deg})) \text{ case CHV =>}
\]

\[
\text{chv := uu::CHV}
\]

\[
\text{newalg} := \text{nthRoot}((\text{chv.left})(\text{dumk}::F), \, \text{chv.deg})
\]

\[
\text{kv := reductum}((\text{chv.int}, \, ku := \text{dumk}, \, \text{newalg})
\]

\[
\text{vu := (chv.right)((x::F))}
\]

\[
\text{vz := (chv.den)(x::F) * (y::F) * denominator(newalg)::F}
\]

\[
\text{map}(x1+\rightarrow \text{eval}(x1, \, [ku, \, kz], \, [vu, \, vz]), \, \text{palgint}(\text{newf}, \, ku, \, kz))
\]

\[
\text{cv} := \text{chvar}(ff, \, \text{modulus})
\]

\[
r := \text{radPoly}(\text{cv.poly})::\text{Record}((\text{radicand}:\text{RF}, \, \text{deg}:N))
\]

\[
\text{qprime := differentiqte}(q := \text{retract}(r.\text{radicand})@\text{UP})::\text{RF}
\]

\[
\text{not zero? qprime and}
\]

\[
((u := \text{chvarIfCan}(\text{cv.func}, \, 1, \, q, \, \text{inv qprime})) \text{ case UPUP) =>}
\]

\[
m := \text{monomial}(1, \, \text{r.deg}$\text{UPUP} - q::\text{UF}::\text{UPUP})
\]

\[
\text{map}(x1+\rightarrow \text{UPUP2F1(RF2UPUP}(x1, \, m), \, \text{cv.c1}, \, \text{cv.c2}, \, x, \, y),
\]

\[
\text{\text{rationalInt}(u::\text{UFUP}, \, \text{r.deg}, \, \text{rationalInt}(1::\text{UPUP}, \, \text{dumk}, \, \text{monomial}(1, \, 1))))}
\]

\[
\text{curve := RadicalFunctionField}(F, \, UP, \, UPUP, \, q::\text{RF}, \, \text{r.deg})
\]

\[
\text{algaddx}(\text{map}(x1+\rightarrow \text{UPUP2F1(lift x1, \, cv.c1}, \, \text{cv.c2}, \, x, \, y),
\]

\[
\text{\text{palgintegrate(reduce(cv.func), \text{differentiate}(\text{UPUP})@\text{ALG}$\text{IR2}, \, x::F))}
\]

-- Do the rationalizing change of variable

-- \(\int((u^n - b)/a) \, (u \, du)\) where

-- \(u^n = y^n = g(x) = a \, x + b\)

-- returns the integral as an integral of a rational function in \(u\)

\[
\text{\text{rationalInt}(f, \, n, \, g) ==}
\]

\[
\begin{cases}
\text{not one? degree g => error "rationalInt: radicand must be linear"} \\
\text{not ((degree g) = 1) => error "rationalInt: radicand must be linear"}
\end{cases}
\]

\[
a := \text{leadingCoefficient} g
\]

\[
\text{integrate}(n * \text{monomial}(\text{inv a}, \, (n-1)::N)$\text{UP}
\]

\[
\text{* chv(f, n, a, \text{leadingCoefficient reductum g})}
\]

-- Do the rationalizing change of variable \(f(x, y) \rightarrow f((u^n - b)/a, \, u)\) where

-- \(u = y = (a \, x + b)^(1/n)\).

-- Returns \(f((u^n - b)/a, \, u)\) as an element of \(F\)

\[
\text{chv0(f, n, a, b) ==}
\]
d := dumk::F
( (f (d::UP::RF)) ((d ** n - b) / a) )

-- candidates(p) returns a list of pairs [g, u] such that p(x) = g(u(x)),
-- those u’s are candidates for change of variables
-- currently uses a dumb heuristic where the candidates u’s are p itself
-- and all the powers x^2, x^3, ..., x^{deg(p)},
-- will use polynomial decomposition in smarter days MB 8/93
candidates p ==
1:List(CND) := empty()
ground? p => l
for i in 2..degree p repeat
  if (u := composite(p, xi := monomial(1, i))) case UP then
    l := concat([u::UP, xi], l)
  concat([monomial(1, 1), p], l)

-- checks whether Int(p(x, y) dx) can be rewritten as
-- Int(r(u, z) du) where u is some polynomial of x,
-- z = d y for some polynomial d, and z^m = g(u)
-- returns either [r(u, z), g, u, d, m] or "failed"
-- we have y^n = radi
changeVarIfCan(p, radi, n) ==
  rec := rootPoly(radi, n)
  for cnd in candidates(rec.radicand) repeat
    if (u := chvarIfCan(p, rec.coef, cnd.right,
      inv(differentiate(cnd.right)::RF))) case UPUP =>
      return [u::UPUP, cnd.left, cnd.right, rec.coef, rec.exponent]
  "failed"

-- checks whether Int(p(x, y) dx) can be rewritten as
-- Int(r(u, z) du) where u is some polynomial of x and z = d y
-- we have y^n = a(x)/d(x)
-- returns either "failed" or r(u, z)
chvarIfCan(p, d, u, u1) ==
  ans:UPUP := 0
  while p ^= 0 repeat
    (v := composite(u1 * leadingCoefficient(p) / d ** degree(p), u))
    case "failed" => return "failed"
    ans := ans + monomial(v::RF, degree p)
    p := reductum p
  ans

algaddx(i, xx) ==
elem? i => i
mkAnswer(ratpart i, logpart i,
  "[- ne.integrand / (xx**2), xx] for ne in notelem i")

prootRDE(nfp, f, g, x, k, rde) ==
  modulus := lift(p := minPoly k, x)
r := radPoly(modulus)::Record(radicand:RF, deg:N)
CHAPTER 17. CHAPTER P

rec := rootPoly(r.radicand, r.deg)
dqdx := inv(differentiate(q := rec.radicand)::RF)
((uf := chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,1)) case UPUP) and
((ug:=chvarIfCan(gg:=univariate(g,x,k,p),rec.coef,q,dqdx)) case UPUP) =>
(u := rde(chv0(uf::UPUP, rec.exponent, 1, 0), rec.exponent *
    (dumk::F) ** (rec.exponent * (rec.exponent - 1))
    * chv0(ug::UPUP, rec.exponent, 1, 0),
    symbolIfCan(dumk)::SY)) case "failed" ⇒ "failed"
eval(u::F, dumk, k::F)
-- one?(rec.coef) =>
((rec.coef) = 1) =>
curve := RadicalFunctionField(F, UP, UPUP, q::RF, rec.exponent)
rc := algDsolve(D()$LDALG + reduce(univariate(nfp, x, k, p))::LDALG,
    reduce univariate(g, x, k, p))$RDALG
rc.particular case "failed" => "failed"
UPUP2F0(lift((rc.particular)::curve), x, k)
palgRDE1(nfp, g, x, k)

prootlimint(f, x, k, lu) ==
modulus := lift(p := minPoly k, x)
r := rootPoly(modulus)::Record(radicand:RF, deg:N)
rec := rootPoly(r.radicand, r.deg)
dqdx := inv(differentiate(q := rec.radicand)::RF)
((uf:=chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,dqdx)) case UPUP) =>
l := empty()$List(RF)
n := rec.exponent * monomial(1, (rec.exponent - 1)::N)$UP
for u in lu repeat
    if ((v:=chvarIfCan(uu:=univariate(u,x,k,p),rec.coef,q,dqdx))case UPUP)
        then l := concat(n * chv(v::UPUP,rec.exponent, 1, 0), l) else FAIL
m := monomial(1, rec.exponent)$UP - q::RF::UPUP
map(x1+->UPUP2F0(RF2UPUP(x1,m), x, k),
    limitedint(n * chv(uf::UPUP, rec.exponent, 1, 0), reverse_! l))
cv := chvar(ff, modulus)
r := rootPoly(cv.poly)::Record(radicand:RF, deg:N)
dqdx := inv(differentiate(q := retract(r.radicand@UP)::RF)
curve := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
(ui := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
case "failed" ⇒ FAIL
[UPUP2F1(lift(ui::curve), cv.c1, cv.c2, x, k), empty()]

prootextint(f, x, k, g) ==
modulus := lift(p := minPoly k, x)
r := rootPoly(modulus)::Record(radicand:RF, deg:N)
rec := rootPoly(r.radicand, r.deg)
dqdx := inv(differentiate(q := rec.radicand)::RF)
((uf:=chvarIfCan(ff := univariate(f,x,k,p),rec.coef,q,dqdx)) case UPUP) and
((ug:=chvarIfCan(gg:=univariate(g,x,k,p),rec.coef,q,dqdx)) case UPUP) =>
m := monomial(1, rec.exponent)$UP - q::RF::UPUP
n := rec.exponent * monomial(1, (rec.exponent - 1)::N)$UP
map(x1+->UPUP2F0(RF2UPUP(x1,m), x, k),
extendedint(n * chv(uf::UPUP, rec.exponent, 1, 0),
    n * chv(ug::UPUP, rec.exponent, 1, 0))

cv := chvar(ff, modulus)
r := radPoly(cv.poly)::Record(radicand:RF, deg:N)
dqdx := inv(differentiate(q := retract(r.radicand)@UP)::RF)
curve := RadicalFunctionField(F, UP, UPUP, q::RF, r.deg)
(u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
    case "failed" => FAIL
    [UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

palgRDE1(nfp, g, x, y) ==
palgLODE1([nfp, 1], g, x, y, simplifyIfCan(x)::SY).particular

palgLODE1(eq, g, kx, y, x) ==
    modulus:= lift(p := minPoly y, kx)
curve := AlgebraicFunctionField(F, UP, UPUP, modulus)
neq:LDALG := 0
    for f in eq for i in 0.. repeat
        neq := neq + monomial(reduce univariate(f, kx, y, p), i)
    empty? remove_!(y, remove_!(kx, varselect(kernels g, x))) =>
    rec := algDsolve(neq, reduce univariate(g, kx, y, p))$RDALG
    bas:List(F) := [UPUP2F0(lift h, kx, y) for h in rec.basis]
    rec.particular case "failed" => ["failed", bas]
    [UPUP2F0(lift((rec.particular)::curve), kx, y), bas]
    rec := algDsolve(neq, 0)
    ["failed", [UPUP2F0(lift h, kx, y) for h in rec.basis]]

palgintegrate(f, x, k) ==
    modulus:= lift(p := minPoly k, x)
cv := chvar(univariate(f, x, k, p), modulus)
curve := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
knownInfBasis(cv.deg)
algaddx(map(x1+->UPUP2F1(lift x1, cv.c1, cv.c2, x, k),
    palgintegrate(reduce(cv.func), differentiate$UP)$ALG)$IR2, x::F)

palglim(f, x, k, lu) ==
    modulus:= lift(p := minPoly k, x)
cv := chvar(univariate(f, x, k, p), modulus)
curve := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
knownInfBasis(cv.deg)
    (u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
    case "failed" => FAIL
    [UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), empty()]

palgext(f, x, k, g) ==
    modulus:= lift(p := minPoly k, x)
cv := chvar(univariate(f, x, k, p), modulus)
curve := AlgebraicFunctionField(F, UP, UPUP, cv.poly)
knownInfBasis(cv.deg)
    (u := palginfieldint(reduce(cv.func), differentiate$UP)$ALG)
case "failed" => FAIL
[UPUP2F1(lift(u::curve), cv.c1, cv.c2, x, k), 0]

palgint(f, x, y) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
    is?(y, "nthRoot"::SY) => prootintegrate(f, x, y)
    is?(y, "rootOf"::SY) => palgintegrate(f, x, y)
    FAIL
    palgint0(f, x, y, u.coef, u.poly)
    palgint0(f, x, y, dumk, v.xsub, v.dxsub)

palgextint(f, x, y, g) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
    is?(y, "nthRoot"::SY) => prootextint(f, x, y, g)
    is?(y, "rootOf"::SY) => palgext(f, x, y, g)
    FAIL
    palgextint0(f, x, y, g, u.coef, u.poly)
    palgextint0(f, x, y, g, dumk, v.xsub, v.dxsub)

palglimint(f, x, y, lu) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
    is?(y, "nthRoot"::SY) => prootlimint(f, x, y, lu)
    is?(y, "rootOf"::SY) => palglim(f, x, y, lu)
    FAIL
    palglimint0(f, x, y, lu, u.coef, u.poly)
    palglimint0(f, x, y, lu, dumk, v.xsub, v.dxsub)

palgRDE(nfp, f, g, x, y, rde) ==
  (v := linearInXIfCan(x, y)) case "failed" =>
  (u := quadIfCan(x, y)) case "failed" =>
    is?(y, "nthRoot"::SY) => prootRDE(nfp, f, g, x, y, rde)
    palgRDE1(nfp, g, x, y)
    palgRDE0(f, g, x, y, rde, u.coef, u.poly)
    palgRDE0(f, g, x, y, rde, dumk, v.xsub, v.dxsub)

-- returns "failed", or (d, P) such that (dy)**2 = P(x)
-- and degree(P) = 2
quadIfCan(x, y) ==
  (degree(p := minPoly y) = 2) and zero?(coefficient(p, 1)) =>
    d := denom(ff :=
               univariate(- coefficient(p, 0) / coefficient(p, 2), x))
    degree(radi := d * numer ff) = 2 => [d(x::F), radi]
    "failed"
    "failed"

if L has LinearOrdinaryDifferentialOperatorCategory F then
  palgLODE(eq, g, kx, y, x) ==
(v := linearInXIfCan(kx, y)) case "failed" =>
(u := quadIfCan(kx, y)) case "failed" =>
palgLODE1([coefficient(eq, i) for i in 0..degree eq], g, kx, y, x)
palgLODE0(eq, g, kx, y, u.coef, u.poly)
palgLODE0(eq, g, kx, y, dumk, v.xsub, v.dxsub)

— INTPAF.dotabb —

"INTPAF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTPAF"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"ACF" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACF"]
"INTPAF" -> "FS"
"INTPAF" -> "ACF"

— package ODEPAL PureAlgebraicLODE —

package ODEPAL PureAlgebraicLODE

— PureAlgebraicLODE.input —

)set break resume
/sys rm -f PureAlgebraicLODE.output
)spool PureAlgebraicLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show PureAlgebraicLODE
--E 1

)spool
)lisp (bye)

— PureAlgebraicLODE.help —

====================================================================
PureAlgebraicLODE examples
====================================================================
In-field solution of an linear ordinary differential equation, 
pure algebraic case.

See Also:
o )show PureAlgebraicLODE

PureAlgebraicLODE (ODEPAL)

Exports:
algDslove

— package ODEPAL PureAlgebraicLODE —

)abbrev package ODEPAL PureAlgebraicLODE
++ Author: Manuel Bronstein
++ Date Created: 21 August 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ In-field solution of an linear ordinary differential equation,
++ pure algebraic case.

PureAlgebraicLODE(F, UP, UPUP, R): Exports == Implementation where
  F : Join(Field, CharacteristicZero,
        RetractableTo Integer, RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F
  UPUP: UnivariatePolynomialCategory Fraction UP
  R : FunctionFieldCategory(F, UP, UPUP)
  RF ==> Fraction UP
  V ==> Vector RF
U ==> Union(R, "failed")
REC ==> Record(particular: Union(RF, "failed"), basis: List RF)
L ==> LinearOrdinaryDifferentialOperator1 R
LQ ==> LinearOrdinaryDifferentialOperator1 RF
Exports ==> with
  algDsolve: (L, R) -> Record(particular: U, basis: List R)
  ++ algDsolve(op, g) returns \spad{["failed", []]} if the equation
  ++ \spad{op y = g} has no solution in \spad{R}. Otherwise, it returns
  ++ \spad{[f, [y_1, \ldots, y_m]]} where \spad{f} is a particular rational
  ++ solution and the \spad{y_i}'s form a basis for the solutions in
  ++ \spad{R} of the homogeneous equation.

Implementation ==> add
  import RationalLODE(F, UP)
  import SystemODESolver(RF, LQ)
  import ReduceLODE(RF, LQ, UPUP, R, L)
algDsolve(l, g) ==
  rec := reduceLODE(l, g)
  sol := solveInField(rec.mat, rec.vec, ratDsolve)
  bas:List(R) := [represents v for v in sol.basis]
  (u := sol.particular) case V => [represents(u::V), bas]
  ["failed", bas]

— ODEPAL.dotabb —

"ODEPAL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODEPAL"]
"FFCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FFCAT"]
"ODEPAL" -> "FFCAT"

package PUSHVAR PushVariables

— PushVariables.input —

)set break resume
)sys rm -f PushVariables.output
)spool PushVariables.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show PushVariables
--R
--R PushVariables(R: Ring,E: OrderedAbelianMonoidSup,OV,PPR: PolynomialCategory(Polynomial(R),E,OV))
--R  OV: OrderedSet with
--R    convert : % -> Symbol
--R    variable : Symbol -> Union(%,"failed") is a package constructor
--R  Abbreviation for PushVariables is PUSHVAR
--R  This constructor is not exposed in this frame.
--R  Issue )edit bookvol10.4.pamphlet to see algebra source code for PUSHVAR
--R
--R---------------------------------------- Operations ----------------------------------------
--R pushdown : (PPR,OV) -> PPR    pushdown : (PPR,List(OV)) -> PPR
--R pushup : (PPR,OV) -> PPR      pushup : (PPR,List(OV)) -> PPR
--R map : ((Polynomial(R) -> PPR),PPR) -> PPR
--R
--E 1
)
spool
)lisp (bye)
PushVariables (PUSHVAR)

Exports:

--- package PUSHVAR PushVariables ---

)abbrev package PUSHVAR PushVariables
++ Description:
++ This package has no description

PushVariables(R,E,OV,PPR):C == T where
E : OrderedAbelianMonoidSup
OV: OrderedSet with
   convert: % -> Symbol
      ++ convert(x) converts x to a symbol
   variable: Symbol -> Union(%, "failed")
      ++ variable(s) makes an element from symbol s or fails
R : Ring
PR ==> Polynomial R
PPR: PolynomialCategory(PR,E,OV)
SUP ==> SparseUnivariatePolynomial
C == with
   pushdown : (PPR, OV) -> PPR
      ++ pushdown(p,v) undocumented{}
   pushdown : (PPR, List OV) -> PPR
      ++ pushdown(p,lv) undocumented{}
   pushup : (PPR, OV) -> PPR
      ++ pushup(p,v) undocumented{}
   pushup : (PPR, List OV) -> PPR
      ++ pushup(p,lv) undocumented{}
   map : ((PR -> PPR), PPR) -> PPR
      ++ map(f,p) undocumented{}

T == add
pushdown(g:PPR,x:OV) : PPR ==
eval(g,x,monomial(1,convert x,1)$PR)
pushdown($g$: PPR, $lv$: List OV) : PPR ==
vals:=[$\text{monomial}(1, \text{convert } x, 1)$ for $x$ in $lv$]
eval($g$, $lv$, $vals$)

$\text{map}(f: (PR \to PPR), p: PPR) : PPR ==$
ground? $p$ => $f($\text{retract} $p$)
$v:=\text{mainVariable}(p)::OV$
multivariate($\text{map}((x: PPR): PPR \to \text{map}(f, x), \text{univariate}(p,v)), v$)

----- push back the variable -----

$\text{pushupCoef}(c: PR, lv: List OV): PPR ==$
ground? $c$ => $c::PPR$
$v:=\text{mainVariable}(c)::Symbol$
v2 := \text{variable}(v)$\$OV$
uc := \text{univariate}(c,v)
ppr : PPR := 0
v2 case OV =>
  while not zero? uc repeat
    ppr := ppr + monomial(1,v2,degree(uc))$PPR *
       pushupCoef(leadingCoefficient uc, lv)
    uc := reductum uc
  ppr
while not zero? uc repeat
  ppr := ppr + monomial(1,v,degree(uc))$PR *
     pushupCoef(leadingCoefficient uc, lv)
  uc := reductum uc
ppr

$\text{pushup}(f: PPR, x: OV) : PPR ==$
map($y \to \text{pushupCoef}(y, [x]), f$)

$\text{pushup}(g: PPR, lv: List OV) : PPR ==$
map($y \to \text{pushupCoef}(y, lv), g$)


— PUSHVAR.dotabb —

"PUSHVAR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=PUSHVAR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"PUSHVAR" -> "PFECAT"
Chapter 18

Chapter Q

package QALGSET2 QuasiAlgebraicSet2

— QuasiAlgebraicSet2.input —

)set break resume
)sys rm -f QuasiAlgebraicSet2.output
)spool QuasiAlgebraicSet2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QuasiAlgebraicSet2
--R
--R QuasiAlgebraicSet2(vl: List(Symbol),nv: NonNegativeInteger) is a package constructor
--R Abbreviation for QuasiAlgebraicSet2 is QALGSET2
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for QALGSET2
--R
--R----------------------------- Operations -----------------------------
--R radicalSimplify : QuasiAlgebraicSet(Fraction(Integer),OrderedVariableList(vl),DirectProduct(nv,NonNegativeInteger))
--R
--E 1

)spool
)lisp (bye)

———

— QuasiAlgebraicSet2.help —

3561
QuasiAlgebraicSet2 examples

QuasiAlgebraicSet2 adds a function radicalSimplify which uses IdealDecompositionPackage to simplify the representation of a quasi-algebraic set.

A quasi-algebraic set is the intersection of a Zariski closed set, defined as the common zeros of a given list of polynomials (the defining polynomials for equations), and a principal Zariski open set, defined as the complement of the common zeros of a polynomial f (the defining polynomial for the inequation).

Quasi-algebraic sets are implemented in the domain QuasiAlgebraicSet, where two simplification routines are provided: idealSimplify and simplify.

The function radicalSimplify is added for comparison study only.

Because the domain IdealDecompositionPackage provides facilities for computing with radical ideals, it is necessary to restrict the ground ring to the domain Fraction Integer, and the polynomial ring to be of type DistributedMultivariatePolynomial.

The routine radicalSimplify uses these to compute groebner basis of radical ideals and is inefficient and restricted when compared to the two in QuasiAlgebraicSet.

See Also:
 o )show QuasiAlgebraicSet2
QuasiAlgebraicSet2 (QALGSET2)

Exports:
radicalSimplify

--- package QALGSET2 QuasiAlgebraicSet2 ---

)abbrev package QALGSET2 QuasiAlgebraicSet2
++ Author: William Sit
++ Date Created: March 13, 1992
++ Date Last Updated: June 12, 1992
++ References: William Sit, "An Algorithm for Parametric Linear Systems"
++ J. Sym. Comp., April, 1992
++ Description:
++ \spadtype{QuasiAlgebraicSet2} adds a function \spadfun{radicalSimplify}
++ which uses \spadtype{IdealDecompositionPackage} to simplify
++ the representation of a quasi-algebraic set. A quasi-algebraic set
++ is the intersection of a Zariski
++ closed set, defined as the common zeros of a given list of
++ polynomials (the defining polynomials for equations), and a principal
++ Zariski open set, defined as the complement of the common
++ zeros of a polynomial f (the defining polynomial for the inequation).
++ Quasi-algebraic sets are implemented in the domain
++ \spadtype{QuasiAlgebraicSet}, where two simplification routines are
++ provided:
++ \spadfun{idealSimplify} and \spadfun{simplify}.
++ The function
++ \spadfun{radicalSimplify} is added
++ for comparison study only. Because the domain
++ \spadtype{IdealDecompositionPackage} provides facilities for
++ computing with radical ideals, it is necessary to restrict
++ the ground ring to the domain \spadtype{Fraction Integer},
++ and the polynomial ring to be of type
++ \spadtype{DistributedMultivariatePolynomial}.
++ The routine \spadfun{radicalSimplify} uses these to compute groebner
++ basis of radical ideals and
++ is inefficient and restricted when compared to the
++ two in \spadtype{QuasiAlgebraicSet}.

QuasiAlgebraicSet2(vl,nv) : C == T where 
  vl : List Symbol
  nv : NonNegativeInteger
  R => Integer
  F => Fraction R
  Var => OrderedVariableList vl
  NNI => NonNegativeInteger
  Expon => DirectProduct(nv,NNI)
  Dpoly => DistributedMultivariatePolynomial(vl,F)
  QALG => QuasiAlgebraicSet(F, Var, Expon, Dpoly)
  newExpon => DirectProduct(#newvl, NNI)
  newPoly => DistributedMultivariatePolynomial(newvl,F)
  newVar => OrderedVariableList newvl
  Status => Union(Boolean, "failed") -- empty or not, or don't know

C == with
radicalSimplify:QALG -> QALG
  ++ radicalSimplify(s) returns a different and presumably simpler
  ++ representation of \( s \) with the defining polynomials for the
  ++ equations
  ++ forming a groebner basis, and the defining polynomial for the
  ++ inequation reduced with respect to the basis, using
  ++ using groebner basis of radical ideals
T == add
  ---- Local Functions ----
  ts:=new()$Symbol
  newvl:=concat(ts, vl)
  tv:newVar:=(variable ts)::newVar
  npoly : Dpoly -> newPoly
  oldpoly : newPoly -> Union(Dpoly,"failed")
  f : Var -> newPoly
  g : newVar -> Dpoly
  import PolynomialIdeals(F,newExpon,newVar,newPoly)
  import GroebnerPackage(F,Expon,Var,Dpoly)
  import GroebnerPackage(F,newExpon,newVar,newPoly)
  import IdealDecompositionPackage(newvl,#newvl)
  import QuasiAlgebraicSet(F, Var, Expon, Dpoly)
  import PolynomialCategoryLifting(Expon,Var,F,Dpoly,newPoly)
  f(v:Var):newPoly ==
    variable((convert v)@Symbol)@Union(newVar,"failed")::newVar:
    :newPoly
  g(v:newVar):Dpoly ==
    v = tv => 0
    variable((convert v)@Symbol)@Union(Var,"failed")::Var::Dpoly
  npoly(p:Dpoly) : newPoly ==
    map(z1 +-> f z1, z2 +-> z2::newPoly, p)
oldpoly(q:newPoly) : Union(Dpoly,"failed") ==
  (x:=mainVariable q) case "failed" => (leadingCoefficient q)::Dpoly
  (x::newVar = tv) => "failed"
  map(z1 +-> g z1, z2 +-> z2::Dpoly, q)

radicalSimplify x ==
  status(x)$QALG = true => x -- x is empty
  z0:=definingEquations x
  n0:=definingInequation x
  t:newPoly:= coerce(tv)$newPoly
  tp:newPoly:= t * (npoly n0) - 1$newPoly
  gen:List newPoly:= concat(tp, [npoly g for g in z0])
  id:=ideal gen
  ngb:=generators radical(id)
  member? (1$newPoly, ngb) => empty()$QALG
  gb:List Dpoly:=nil
  while not empty? ngb repeat
    if ((k:=oldpoly ngb.first) case Dpoly) then gb:=concat(k, gb)
    ngb:=ngb.rest
  y:=quasiAlgebraicSet(gb, primitivePart normalForm(n0, gb))
  setStatus(y,false::Status)

---

QALGSET2.dotabb

"QALGSET2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QALGSET2"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"QALGSET2" -> "ALIST"

---

package QCMPACK QuasiComponentPackage

--- QuasiComponentPackage.input ---

)set break resume
)sys rm -f QuasiComponentPackage.output
)spool QuasiComponentPackage.output
)set message test on
)set message auto off
)clear all
A package for removing redundant quasi-components and redundant branches when decomposing a variety by means of quasi-components of regular triangular sets.

See Also:
  o )show QuasiComponentPackage
QuasiComponentPackage (QCMPACK)

Exports:

- algebraicSort
- branchIfCan
- infRittWu?
- internalInfRittWu?
- internalSubPolSet?
- internalSubQuasiComponent?
- moreAlgebraic?
- prepareDecompose
- removeSuperfluousCases
- removeSuperfluousQuasiComponents
- startTable!
- subCase?
- subPolSet?
- subQuasiComponent?
- subQuasiComponent?
- subTriSet?
- supDimElseRittWu?

— package QCMPACK QuasiComponentPackage —

)abbrev package QCMPACK QuasiComponentPackage
++ Author: Marc Moreno Maza <marc@nag.co.uk>
++ Date Created: 08/30/1998
++ Date Last Updated: 12/16/1998
++ References :
++ [1] D. LAZARD "A new method for solving algebraic systems of
++ d'extensions simples et resolution des systemes d'équations
++ Description:
++ A package for removing redundant quasi-components and redundant
++ branches when decomposing a variety by means of quasi-components
++ of regular triangular sets.

QuasiComponentPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
Key ==> Record(left:TS, right:TS)
Entry ==> Boolean
H ==> TabulatedComputationPackage(Key, Entry)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)

Exports == with
  startTable!: (S,S,S) -> Void
    ++ \texttt{startTableGcd!(s1,s2,s3)}
    ++ is an internal subroutine, exported only for development.
  stopTable!: () -> Void
    ++ \texttt{stopTableGcd!()}
    ++ is an internal subroutine, exported only for development.
  supDimElseRittWu?: (TS,TS) -> Boolean
    ++ \texttt{supDimElseRittWu(ts,us)} returns true iff \texttt{ts}
    ++ has less elements than \texttt{us} otherwise if \texttt{ts}
    ++ has higher rank than \texttt{us} w.r.t. Ritt and Wu ordering.
  algebraicSort: Split -> Split
    ++ \texttt{algebraicSort(lts)} sorts \texttt{lts} w.r.t
    ++ \texttt{supDimElseRittWu}?
  moreAlgebraic?: (TS,TS) -> Boolean
    ++ \texttt{moreAlgebraic?(ts,us)} returns false iff \texttt{ts}
    ++ and \texttt{us} are both empty, or \texttt{ts}
    ++ has less elements than \texttt{us}, or some variable is
    ++ algebraic w.r.t. \texttt{us} and is not w.r.t. \texttt{ts}.
  subTriSet?: (TS,TS) -> Boolean
    ++ \texttt{subTriSet?(ts,us)} returns true iff \texttt{ts}
    ++ a sub-set of \texttt{us}.
  subPolSet?: (LP, LP) -> Boolean
    ++ \texttt{subPolSet?(lp1,lp2)} returns true iff \texttt{lp1}
    ++ a sub-set of \texttt{lp2}.
  internalSubPolSet?: (LP, LP) -> Boolean
    ++ \texttt{internalSubPolSet?(lp1,lp2)} returns true iff \texttt{lp1}
    ++ a sub-set of \texttt{lp2} assuming that these lists are sorted
    ++ increasingly w.r.t.
    ++ \texttt{infRittWu?} from RecursivePolynomialCategory.
internalInfRittWu?: (LP, LP) -> Boolean
++ \texttt{axiom\{internalInfRittWu?(lp1,lp2)\}}
++ is an internal subroutine, exported only for development.
infRittWu?: (LP, LP) -> Boolean
++ \texttt{axiom\{infRittWu?(lp1,lp2)\}}
++ is an internal subroutine, exported only for development.
internalSubQuasiComponent?: (TS,TS) -> Union(Boolean,"failed")
++ \texttt{axiom\{internalSubQuasiComponent?(ts,us)\}} returns a
++ boolean \spad{b} value if the fact that the regular
++ zero set of \texttt{axiom\{us\}} contains that of
++ \texttt{axiom\{ts\}} can be decided (and in that case \texttt{axiom\{b\}} gives this
++ inclusion) otherwise returns \texttt{axiom\{"failed"\}}.
subQuasiComponent?: (TS,TS) -> Boolean
++ \texttt{axiom\{subQuasiComponent?(ts,us)\}} returns true iff
++ internalSubQuasiComponent?
++ returns true.
subQuasiComponent?: (TS,Split) -> Boolean
++ \texttt{axiom\{subQuasiComponent?(ts,lus)\}} returns true iff
++ \texttt{axiom\{subQuasiComponent?(ts,us)\}} holds for one \spad{us}
++ in \texttt{spad\{lus\}}.
removeSuperfluousQuasiComponents: Split -> Split
++ \texttt{axiom\{removeSuperfluousQuasiComponents(lts)\}} removes
++ from \texttt{axiom\{lts\}} any \spad{ts} such that
++ \texttt{axiom\{subQuasiComponent?(ts,us)\}} holds for
++ another \spad{us} in \texttt{axiom\{lts\}}.
subCase?: (LpWT,LpWT) -> Boolean
++ \texttt{axiom\{subCase?(lpwt1,lpwt2)\}}
++ is an internal subroutine, exported only for development.
removeSuperfluousCases: List LpWT -> List LpWT
++ \texttt{axiom\{removeSuperfluousCases(llpwt)\}}
++ is an internal subroutine, exported only for development.
prepareDecompose: (LP, List(TS),B,B) -> List Branch
++ \texttt{axiom\{prepareDecompose(lp,lts,b1,b2)\}}
++ is an internal subroutine, exported only for development.
branchIfCan: (LP,TS,LP,B,B,B,B,B) -> Union(Branch,"failed")
++ \texttt{axiom\{branchIfCan(leq,ts,lineq,b1,b2,b3,b4,b5)\}}
++ is an internal subroutine, exported only for development.

Implementation == add

\begin{verbatim}

squareFreeFactors(lp: LP): LP ==
  lsflp: LP := []
  for p in lp repeat
    lsfp := squareFreeFactors(p)$polsetpack
    lsflp := concat(lsfp,lsflp)
  sort(infRittWu?,removeDuplicates lsflp)

startTable!(ok: S, ko: S, domainName: S): Void ==
  initTable!()$H
  if (not empty? ok) and (not empty? ko) then printInfo!(ok,ko)$H
\end{verbatim}


if (not empty? domainName) then startStats!(domainName)
void()

stopTable!(): Void ==
  if makingStats?() then printStats!()
  clearTable!()

supDimElseRittWu? (ts:TS, us:TS): Boolean ==
  #ts < #us => true
  #ts > #us => false
  lp1 :LP := members(ts)
  lp2 :LP := members(us)
  while (not empty? lp1) and (not infRittWu?(first(lp2), first(lp1))) repeat
    lp1 := rest lp1
    lp2 := rest lp2
  not empty? lp1

algebraicSort (lts:Split): Split ==
  lts := removeDuplicates lts
  sort(supDimElseRittWu?, lts)

moreAlgebraic?(ts:TS, us:TS): Boolean ==
  empty? ts => empty? us
  empty? us => true
  #ts < #us => false
  for p in (members us) repeat
    not algebraic?(mvar(p), ts) => return false
  true

subTriSet?(ts:TS, us:TS): Boolean ==
  empty? ts => true
  empty? us => false
  mvar(ts) > mvar(us) => false
  mvar(ts) < mvar(us) => subTriSet?(ts, rest(us)::TS)
  first(ts)::P = first(us)::P => subTriSet?(rest(ts)::TS, rest(us)::TS)
  false

internalSubPolSet?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => true
  empty? lp2 => false
  associates?(first lp1, first lp2) =>
    internalSubPolSet?(rest lp1, rest lp2)
  infRittWu?(first lp1, first lp2) => false
  internalSubPolSet?(lp1, rest lp2)

subPolSet?(lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalSubPolSet?(lp1, lp2)
infRittWu?(lp1: LP, lp2: LP): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalInfRittWu?(lp1, lp2)

internalInfRittWu?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => not empty? lp2
  empty? lp2 => false
  infRittWu?(first lp1, first lp2)$P => true
  infRittWu?(first lp2, first lp1)$P => false
  infRittWu?(rest lp1, rest lp2)$$

subCase? (lpwt1:LpWT, lpwt2:LpWT): Boolean ==
  -- ASSUME lpwt.{1,2}.val is sorted w.r.t. infRittWu?
  not internalSubPolSet?(lpwt2.val, lpwt1.val) => false
  subQuasiComponent?(lpwt1.tower, lpwt2.tower)

internalSubQuasiComponent?(ts:TS, us:TS): Union(Boolean,"failed") ==
  -- "failed" is false iff saturate(us) is radical
  subTriSet?(us, ts) => true
  not moreAlgebraic?(ts, us) => false::Union(Boolean,"failed")
  for p in (members us) repeat
    mdeg(p) < mdeg(select(ts, mvar(p))::P) =>
      return("failed"::Union(Boolean,"failed"))
  for p in (members us) repeat
    not zero? initiallyReduce(p, ts) =>
      return("failed"::Union(Boolean,"failed"))
  lsfp := squareFreeFactors(initials us)
  for p in lsfp repeat
    not invertible?(p, ts)@B =>
      return(false::Union(Boolean,"failed"))
  true::Union(Boolean,"failed")

subQuasiComponent?(ts:TS, us:TS): Boolean ==
  k: Key := [ts, us]
  e := extractIfCan(k)$H
  e case Entry => e::Entry
  ubf: Union(Boolean,"failed") := internalSubQuasiComponent?(ts, us)
  b: Boolean := (ubf case Boolean) and (ubf::Boolean)
  insert!(k, b)$H
  b

subQuasiComponent?(ts:TS, lus:Split): Boolean ==
  for us in lus repeat
    subQuasiComponent?(ts, us)$B => return true
  false

removeSuperfluousCases (cases:List LpWT) ==
  #cases < 2 => cases
\[\text{toSee} := \\text{sort}(\langle x: LpWT, y: LpWT \rangle : \text{Boolean} -> \\
\quad \text{supDimElseRittWu?(x.tower, y.tower), cases})\]

\[lptv1, lptv2 : \text{LpWT}\]
\[\text{toSave}, \text{headmaxcases}, \text{maxcases}, \text{copymaxcases} : \text{List LpWT}\]
\[\text{while not empty? toSee repeat}\]
\[lptv1 := \text{first toSee}\]
\[\text{toSee} := \text{rest toSee}\]
\[\text{toSave} := []\]
\[\text{for lptv2 in toSee repeat}\]
\[\text{if subCase?(lptv1, lptv2)}\]
\[\text{then}\]
\[\quad \text{lptv1} := \text{lptv2}\]
\[\text{else}\]
\[\quad \text{if not subCase?(lptv2, lptv1)}\]
\[\quad \text{then}\]
\[\quad \quad \text{toSave} := \text{cons(lptv2, toSave)}\]
\[\text{if empty? maxcases}\]
\[\text{then}\]
\[\quad \text{headmaxcases} := [\text{lptv1}]\]
\[\quad \text{maxcases} := \text{headmaxcases}\]
\[\text{else}\]
\[\quad \text{copymaxcases} := \text{maxcases}\]
\[\quad \text{while (not empty? copymaxcases) and }\]
\[\quad \quad \text{(not subCase?(lptv1, first(copymaxcases)))} \text{ repeat}\]
\[\quad \quad \text{copymaxcases} := \text{rest copymaxcases}\]
\[\quad \text{if empty? copymaxcases}\]
\[\quad \text{then}\]
\[\quad \quad \text{setrest!(headmaxcases, [lptv1])}\]
\[\quad \quad \text{headmaxcases} := \text{rest headmaxcases}\]
\[\text{toSee} := \text{reverse toSave}\]
\[\text{maxcases}\]

\[\text{removeSuperfluousQuasiComponents(lts: Split): Split} =\]
\[\text{lts} := \text{removeDuplicates lts}\]
\[#\text{lts} < 2 => \text{lts}\]
\[\text{toSee} := \text{algebraicSort lts}\]
\[\text{toSave}, \text{headmaxlts}, \text{maxlts}, \text{copymaxlts} : \text{Split}\]
\[\text{while not empty? toSee repeat}\]
\[\text{ts} := \text{first toSee}\]
\[\text{toSee} := \text{rest toSee}\]
\[\text{toSave} := []\]
\[\text{for us in toSee repeat}\]
\[\text{if subQuasiComponent?(ts, us)@B}\]
\[\text{then}\]
\[\quad \text{ts} := \text{us}\]
\[\text{else}\]
\[\quad \text{if not subQuasiComponent?(us, ts)@B}\]
\[\quad \text{then}\]
\[\quad \quad \text{toSave} := \text{cons(us, toSave)}\]
if empty? maxlts
  then
    headmaxlts := [ts]
    maxlts := headmaxlts
  else
    copymaxlts := maxlts
    while (not empty? copymaxlts) and _
       (not subQuasiComponent?(ts,first(copymaxlts))@B) repeat
      copymaxlts := rest copymaxlts
    if empty? copymaxlts
      then
        setrest!(headmaxlts,[ts])
        headmaxlts := rest headmaxlts
    toSee := reverse toSave
    algebraicSort maxlts

removeAssociates (lp:LP):LP ==
  removeDuplicates [primitivePart(p) for p in lp]

branchIfCan(leq: LP, ts, lineq: LP, b1:B,b2:B,b3:B,b4:B,b5:B):UBF ==
  -- ASSUME pols in leq are squarefree and mainly primitive
  -- if b1 then CLEAN UP leq
  -- if b2 then CLEAN UP lineq
  -- if b3 then SEARCH for ZERO in lineq with leq
  -- if b4 then SEARCH for ZERO in lineq with ts
  -- if b5 then SEARCH for ONE in leq with lineq
  if b1
    then
      leq := removeAssociates(leq)
      leq := remove(zero?,leq)
      any?(ground?,leq) =>
        return("failed":Union(Branch,"failed"))
  if b2
    then
      any?(zero?,lineq) =>
        return("failed":Union(Branch,"failed"))
      lineq := removeRedundantFactors(lineq)$polsetpack
  if b3
    then
      ps: PS := construct(leq)$PS
      for q in lineq repeat
        zero? remainder(q,ps).polnum =>
          return("failed":Union(Branch,"failed"))
      (empty? leq) or (empty? lineq) => ([leq, ts, lineq]$Branch)::UBF
  if b4
    then
      for q in lineq repeat
        zero? initiallyReduce(q,ts) =>
          return("failed":Union(Branch,"failed"))
  if b5
newleq: LP := []
for p in leq repeat
  for q in lineq repeat
    if mvar(p) = mvar(q)
      then
        g := gcd(p,q)
        newp := (p exquo g)::P
        ground? newp =>
          return("failed":::Union(Branch,"failed"))
        newleq := cons(newp,newleq)
      else
        newleq := cons(p,newleq)
  leq := newleq
leq := sort(infRittWu?, removeDuplicates leq)
([leq, ts, lineq]$Branch)::UBF

prepareDecompose(lp: LP, lts: List(TS), b1: B, b2: B): List Branch ==
  -- if b1 then REMOVE REDUNDANT COMPONENTS in lts
  -- if b2 then SPLIT the input system with squareFree
  lp := sort(infRittWu?, remove(zero?,removeAssociates(lp)))
  any?(ground?,lp) => []
  empty? lts => []
  if b1 then lts := removeSuperfluousQuasiComponents lts
  not b2 =>
    [[lp,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
toSee: List Branch
lq: LP := []
toSee := [[lq,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
empty? lp => toSee
for p in lp repeat
  lsfp := squareFreeFactors(p)$polsetpack
  branches: List Branch := []
  lq := []
  for f in lsfp repeat
    for branch in toSee repeat
      leq := branch.eq
      ts := branch.tower
      lineq := branch.ineq
      ubf1: UBF := branchIfCan(leq,ts,lq,false,false,true,true,true,true)@UBF
      ubf1 case "failed" => "leave"
      ubf2: UBF :=
        branchIfCan([f],ts,lineq,false,false,true,true,true,true)@UBF
      ubf2 case "failed" => "leave"
      leq := sort(infRittWu?,removeDuplicates concat(ubf1.eq,ubf2.eq))
      lineq :=
        sort(infRittWu?,removeDuplicates concat(ubf1.ineq,ubf2.ineq))
      newBranch :=
        branchIfCan(leq,ts,lineq,false,false,true,false,false)
      branches:= cons(newBranch::Branch,branches)
---

--- QCMPACK.dotabb ---

"QCMPACK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QCMPACK"]
"RSETCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RSETCAT"]
"QCMPACK" -> "RSETCAT"

---

package QFCAT2 QuotientFieldCategoryFunctions2

--- QuotientFieldCategoryFunctions2.input ---

)set break resume
)sys rm -f QuotientFieldCategoryFunctions2.output
)spool QuotientFieldCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show QuotientFieldCategoryFunctions2
--R
--R QuotientFieldCategoryFunctions2(A: IntegralDomain,B: IntegralDomain,R: QuotientFieldCategory(A),S: QuotientFieldCategory(B)) is a package constructor
--R Abbreviation for QuotientFieldCategoryFunctions2 is QFCAT2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for QFCAT2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((A -> B),R) -> S
--R
--E 1

)spool
)lisp (bye)

---

--- QuotientFieldCategoryFunctions2.help ---
This package extends a function between integral domains to a mapping between their quotient fields.

See Also:
- )show QuotientFieldCategoryFunctions2

---

**QuotientFieldCategoryFunctions2 (QFCAT2)**

Exports:
- map

---

)abbrev package QFCAT2 QuotientFieldCategoryFunctions2
++ Description:
++ This package extends a function between integral domains to a mapping between their quotient fields.

QuotientFieldCategoryFunctions2(A, B, R, S): Exports == Impl where
  A, B: IntegralDomain
  R : QuotientFieldCategory(A)
  S : QuotientFieldCategory(B)

Exports => with
  map: (A -> B, R) -> S
  ++ map(func,frac) applies the function func to the numerator and denominator of frac.
package QUATCT2 QuaternionCategoryFunctions2

Impl ==> add
map(f, r) == f(numer r) / f(denom r)

package QUATCT2 QuaternionCategoryFunctions2

)set break resume
)spool QuaternionCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 4
q := quatern(2/11,-8,3/4,1)
--R
--R
--R 2 3
--R (1) -- - 8i + - j + k
--R 11 4
--R
--R Type: Quaternion(Fraction(Integer))
--E 1

--S 2 of 4
f(a:Fraction Integer):Complex Fraction Integer == a::Complex Fraction Integer
--R
--R Function declaration f : Fraction(Integer) -> Complex(Fraction(Integer)) has been added to workspace.
--R
--R Type: Void
--E 3

--S 3 of 4
map(f,q)
Compiling function f with type Fraction(Integer) -> Complex(Fraction Integer)

2 3
(3) -- 8i + j + k
11 4

Type: Quaternion(Complex(Fraction(Integer)))

---

--- QuaternionCategoryFunctions2.help ---

QuaternionCategoryFunctions2 examples

QuaternionCategoryFunctions2 implements functions between two quaternion domains. The function map is used by the system interpreter to coerce between quaternion types.

The package QuaternionCategoryFunctions2 provides a function map to convert an expression in Quaternion(R) to an expression in Quaternion(S) using the function f.

q := quatern(2/11,-8,3/4,1)

2 3
-- 8i + j + k
11 4

Type: Quaternion Fraction Integer
f(a:Fraction Integer):Complex Fraction Integer == a::Complex Fraction Integer
   Function declaration f : Fraction Integer -> Complex Fraction Integer has been added to workspace.
       Type: Void

map(f,q)
   Compiling function f with type Fraction Integer -> Complex Fraction Integer

\[
\begin{array}{c}
\frac{2}{11} \quad \frac{3}{4} \\
\frac{-8i + - j + k}{11}
\end{array}
\]

Type: Quaternion Complex Fraction Integer

See Also:
   o )show QuaternionCategoryFunctions2

---

**QuaternionCategoryFunctions2 (QUATCT2)**

Exports:
   map

— package QUATCT2 QuaternionCategoryFunctions2 —

)abbrev package QUATCT2 QuaternionCategoryFunctions2
   ++ Author: Robert S. Sutor
   ++ Date Created: 23 May 1990
   ++ Date Last Updated: 23 May 1990
   ++ Description:
   ++ \spadtype{QuaternionCategoryFunctions2} implements functions between
++ two quaternion domains. The function \spadfun{map} is used by
++ the system interpreter to coerce between quaternion types.

QuaternionCategoryFunctions2(QR,R,QS,S) : Exports ==
  Implementation where
    R : CommutativeRing
    S : CommutativeRing
    QR : QuaternionCategory R
    QS : QuaternionCategory S
  Exports == with
    map: (R -> S, QR) -> QS
    ++ map(f,u) maps f onto the component parts of the quaternion u.
    ++ to convert an expression in Quaternion(R) to Quaternion(S)
    ++
    ++X f(a:FRAC(INT)):COMPLEX(FRAC(INT)) == a:COMPLEX(FRAC(INT))
    ++X q:=quatern(2/11,-8,3/4,1)
    ++X map(f,q)
  Implementation == add
    map(fn : R -> S, u : QR): QS ==
      quatern(fn real u, fn imagI u, fn imagJ u, fn imagK u)$QS

"QUATCT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=QUATCT2"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=QUATCAT"]
"QUATCT2" -> "QUATCAT"
Chapter 19

Chapter R

package REP RadicalEigenPackage

— RadicalEigenPackage.input —

)set break resume
)sys rm -f RadicalEigenPackage.output
)spool RadicalEigenPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RadicalEigenPackage
--R
--R RadicalEigenPackage is a package constructor
--R Abbreviation for RadicalEigenPackage is REP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for REP
--R
--R------------------------------- Operations --------------------------------
--R eigenMatrix : Matrix(Fraction(Polynomial(Integer))) -> Union(Matrix(Expression(Integer)),"failed")
--R gramschmidt : List(Matrix(Expression(Integer))) -> List(Matrix(Expression(Integer)))
--R normalise : Matrix(Expression(Integer)) -> Matrix(Expression(Integer))
--R orthonormalBasis : Matrix(Fraction(Polynomial(Integer))) -> List(Matrix(Expression(Integer)))
--R radicalEigenvalues : Matrix(Fraction(Polynomial(Integer))) -> List(Expression(Integer))
--R radicalEigenvector : (Expression(Integer),Matrix(Fraction(Polynomial(Integer)))) -> List(Matrix(Expression(Integer)))
--R radicalEigenvectors : Matrix(Fraction(Polynomial(Integer))) -> List(Record(radval: Expression(Integer),radmult: Integer,radvect: List(Matrix(Expression(Integer))))
--R
--E 1

)spool

3581
CHAPTER 19. CHAPTER R

)lisp (bye)

— RadicalEigenPackage.help —

====================================================================
RadicalEigenPackage examples
====================================================================

Package for the computation of eigenvalues and eigenvectors. This package works for matrices with coefficients which are rational functions over the integers.

The eigenvalues and eigenvectors are expressed in terms of radicals.

See Also:
o )show RadicalEigenPackage
o )show Fraction(Polynomial(Integer))

— RadicalEigenPackage (REP) —

Exports:
eigenMatrix gramschmidt normalise orthonormalBasis
radicalEigenvalues radicalEigenvectors

— package REP RadicalEigenPackage —

)abbrev package REP RadicalEigenPackage
++ Author: P.Gianni
++ Date Created: Summer 1987
++ Date Last Updated: October 1992
++ Description:
++ Package for the computation of eigenvalues and eigenvectors.
++ This package works for matrices with coefficients which are
++ rational functions over the integers.
++ (see \spadtype{Fraction Polynomial Integer}).
++ The eigenvalues and eigenvectors are expressed in terms of radicals.

RadicalEigenPackage() : C == T

where

R ==> Integer
P ==> Polynomial R
F ==> Fraction P
RE ==> Expression R
SE ==> Symbol()
M ==> Matrix(F)
MRE ==> Matrix(RE)
ST ==> SuchThat(SE,P)
NNI ==> NonNegativeInteger

EigenForm ==> Record(eigval:Union(F,ST),eigmult:NNI,eigvec:List(M))
RadicalForm ==> Record(radval:RE,radmult:Integer,radvect:List(MRE))

C == with

radicalEigenvectors : M -> List(RadicalForm)

++ radicalEigenvectors(m) computes
++ the eigenvalues and the corresponding eigenvectors of the
++ matrix m;
++ when possible, values are expressed in terms of radicals.

radicalEigenvector : (RE,M) -> List(MRE)

++ radicalEigenvector(c,m) computes the eigenvector(s) of the
++ matrix m corresponding to the eigenvalue c;
++ when possible, values are
++ expressed in terms of radicals.

radicalEigenvalues : M -> List RE

++ radicalEigenvalues(m) computes the eigenvalues of the matrix m;
++ when possible, the eigenvalues are expressed in terms of radicals.

eigenMatrix : M -> Union(MRE,"failed")

++ eigenMatrix(m) returns the matrix b
++ such that \spad{b*m*(inverse b)} is diagonal,
++ or "failed" if no such b exists.

normalise : MRE -> MRE

++ normalise(v) returns the column
++ vector v
++ divided by its euclidean norm;
++ when possible, the vector \( v \) is expressed in terms of radicals.

\[ \text{gramschmidt} : \text{List(MRE)} \rightarrow \text{List(MRE)} \]
++ \text{gramschmidt(lv)} converts the list of column vectors \( \text{lv} \) into
++ a set of orthogonal column vectors
++ of euclidean length 1 using the Gram-Schmidt algorithm.

\[ \text{orthonormalBasis} : \text{M} \rightarrow \text{List(MRE)} \]
++ \text{orthonormalBasis(m)} returns the orthogonal matrix \( b \) such that
++ \( \text{spad}(b*m*(\text{inverse} \ b)) \) is diagonal.
++ Error: if \( m \) is not a symmetric matrix.

\[ T == \text{add} \]
\[ \text{PI} \quad \Rightarrow \quad \text{PositiveInteger} \]
\[ \text{RSP} := \text{RadicalSolvePackage R} \]
\[ \text{import} \ \text{EigenPackage R} \]

\[ \text{---- Local Functions ----} \]
\[ \text{evalvect} : (\text{M},\text{RE},\text{SE}) \rightarrow \text{MRE} \]
\[ \text{innerprod} : (\text{MRE},\text{MRE}) \rightarrow \text{RE} \]

\[ \text{---- eval a vector of } F \text{ in a radical expression ----} \]
\[ \text{evalvect(vect:M,alg:RE,x:SE)} : \text{MRE} == \]
\[ n:=\text{nrows vect} \]
\[ xx:=\text{kernel(x)}\%\text{Kernel(RE)} \]
\[ w:\text{MRE}:=\text{zero}(n,1)\%\text{MRE} \]
\[ \text{for} \ i \ \text{in} \ 1..n \ \text{repeat} \]
\[ v:=\text{eval(vect(i,1)} :: \text{RE},xx,\text{alg}) \]
\[ \text{setelt}(w,i,1,v) \]
\[ w \]

\[ \text{---- inner product ----} \]
\[ \text{innerprod(v1:MRE,v2:MRE)}: \text{RE} == ((\text{transpose v1)}* \ v2)::\text{MRE})(1,1) \]

\[ \text{---- normalization of a vector ----} \]
\[ \text{normalise(v:MRE)} : \text{MRE} == \]
\[ \text{normv:RE} := \text{sqrt(innerprod(v,v))} \]
\[ \text{normv} = 0\%\text{RE} \Rightarrow \text{v} \]
\[ (1/\text{normv})*\text{v} \]

\[ \text{---- Eigenvalues of the matrix } A \text{ ----} \]
\[ \text{radicalEigenvalues(A:M)}: \text{List(RE)} == \]
\[ x:SE := \text{new()}\%\text{SE} \]
\[ \text{pol}:= \text{characteristicPolynomial(A,x)} :: \text{F} \]
\[ \text{radicalRoots(pol,x)}\%\text{RSP} \]

\[ \text{---- Eigenvectors belonging to a given eigenvalue ----} \]
\[ \text{---- expressed in terms of radicals ----} \]
\[ \text{radicalEigenvector(alpha:RE,A:M)} : \text{List(MRE)} == \]
\[ n:=\text{nrows A} \]
B := zero(n,n)
for i in 1..n repeat
    for j in 1..n repeat B(i,j) := A(i,j) :: RE
    B(i,i) := B(i,i) - alpha
[v :: MRE for v in nullSpace B]

---- eigenvectors and eigenvalues ----
radicalEigenvectors(A : M) : List(RadicalForm) ==
    leig := eigenvectors A
    n := nrows A
    sln := List RadicalForm := empty()
    veclist := List MRE
    for eig in leig repeat
        eig.eigval case F =>
            veclist := empty()
            for ll in eig.eigvec repeat
                m := zero(n,1)
                for i in 1..n repeat m(i,1) := (ll(i,1)) :: RE
                veclist := cons(m, veclist)
            sln := cons([eig.eigval :: F :: RE, eig.eigmult, veclist]$RadicalForm, sln)
        sym := eig.eigval :: ST
        xx := lhs sym
        lval := List RE := radicalRoots((rhs sym) :: F, xx)$RSP
        for alg in lval repeat
            nsl := [alg, eig.eigmult, [evalvect(ep, alg, xx) for ep in eig.eigvec]]$RadicalForm
            sln := cons(nsl, sln)
    sln

---- orthonormalization of a list of vectors ----
    -- Graham - Schmidt process ----
gramschmidt(lvect : List(MRE)) : List(MRE) ==
    lvect = [] => []
    v := lvect.first
    n := nrows v
    RMR := RectangularMatrix(n:PI, 1, RE)
    orth := List(MRE) : [normalise v]
    for v in lvect.rest repeat
        pol := ((v: RMR) - (+/[innerprod(w, v) * w: RMR for w in orth]))$MRE
        orth := cons(normalise pol, orth)
    orth

---- The matrix of eigenvectors ----
eigenMatrix(A : M) : Union(MRE, "failed") ==
    lef := List(MRE) := [eiv.radvect for eiv in radicalEigenvectors(A)]
    n := nrows A
    #lef < n => "failed"
d:MRE:=copy(lef.first)
for v in lef.rest repeat d:=(horizConcat(d,v))::MRE
d

---- orthogonal basis for a symmetric matrix ----

orthonormalBasis(A:M):List(MRE) ==
  symmetric?(A) => error "the matrix is not symmetric"
  basis:List(MRE):=[]
  lvec:List(MRE):=[]
  alglist:List(RadicalForm):=radicalEigenvectors(A)
  n:=nrows A
  for alterm in alglist repeat
    if (lvec:=alterm.radvect)=[] then error "sorry "
    if #(lvec)>1 then
      lvec:= gramschmidt(lvec)
      basis:=[lvec,basis]
    else basis:=[normalise(lvec.first),basis]
  basis

package SOLVERAD RadicalSolvePackage

  -- RadicalSolvePackage.input --

)set break resume
)sys rm -f RadicalSolvePackage.output
)spool RadicalSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 21
b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
--R
\[ \begin{align*}
\text{Type: } & \text{Fraction(Polynomial(Integer))} \\
\end{align*} \]

\[ \begin{align*}
\text{radicalSolve(b,x)} \\
\text{radicalSolve(b)} \\
\text{radicalSolve(b=0,x)} \\
\text{radicalSolve(b=0)}
\end{align*} \]
\[ x = \text{radicalRoots}(b,x) \]
\[ x = \text{contractSolve}(b=0,x) \]
\[ c : \frac{y^2 + 4}{y + 1} \]
radicalSolve([b,c],[x,y])

```
3+---+ +---+ +---+ 3+---+ 3+---+ 3+---+
- \|- 7 \|- 1 \|- 7 \|- 1 \|- 7 \|- 7
\[x= \------------------------------------------------------------------, y= - 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= - 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= - 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= - 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= - 2\|- 1 \],
3++
2\|- 1
\[x= \------------------------------------------------------------------, y= 2\|- 1 \],
3++
2\|- 1
```

Type: List(List(Equation(Expression(Expression(Integer)))))
\text{Type: List(List(Equation(Expression(Integer))))}

\text{radicalSolve([b=0,c=0],[x,y])}

\text{(12)}

\text{radicalSolve([b=0,c=0])}

\text{(13)}
radicalRoots([b,c],[x,y])

Type: List(List(Expression(Integer)))

)clear all

t1:=radicalSolve(x^3 + x^2 - 7 = 0,x)

Type: List(List(Expression(Integer)))
\[ x = \frac{(-9\|1295 + 187\|13)}{3\|54\|3} \]
\[ \text{Type: List(Equation(Expression(Integer))))} \]
```
--R + \ | 54\13
--R +------------------+
--R | +----+ +-+
--R \ |1295 + 187\13
--R (9\- 3\- 3 + 3) |------------------ + 2
--R \ | +----+ +-+
--R \ | 54\13
--R Type: Expression(Integer)
--E 17

--S 18 of 21
t4:=rhs(t1.3)
--R +------------------+
--R | +----+ +-+
--R \ |1295 + 187\13
--R (9\- 3 - 9) |------------------ - 3
--R \ | +----+ +-+
--R \ | 54\13
--R Type: Expression(Integer)
--E 18

--S 19 of 21
t2^3+t2^2-7
--R +------------------+
--R | +----+ +-+
--R \ |1295 + 187\13
--R Type: Expression(Integer)
--E 19

--S 20 of 21
t3^3+t3^2-7
--R +------------------+
--R | +----+ +-+
--R \ |1295 + 187\13
--R Type: Expression(Integer)
--E 20
```
This package tries to find solutions expressed in terms of radicals for systems of equations of rational functions with coefficients in an integral domain \( R \).

\( b : \text{Fraction}(\text{Polynomial}(\text{Integer})):=(3\times x^3+7)/(5\times x^2-13) \)
\[
\frac{3x + 7}{2} = \frac{5x - 13}{2}
\]

\[
\text{radicalSolve}(b,x)
\]

\[
3x + 7 + \frac{3x}{2} = 5x - 13 + \frac{5x}{2}
\]

\[
\begin{align*}
3x + 7 &= 5x - 13 \\
3x + 7 &= 5x - 13
\end{align*}
\]

\[
[x= \frac{10}{2}, x= \frac{3}{2}]
\]

\[
\text{radicalSolve}(b=0,x)
\]

\[
3x + 7 + \frac{3x}{2} = 5x - 13 + \frac{5x}{2}
\]

\[
\begin{align*}
3x + 7 &= 5x - 13 \\
3x + 7 &= 5x - 13
\end{align*}
\]

\[
[x= \frac{10}{2}, x= \frac{3}{2}]
\]

\[
\text{contractSolve}(b=0,x)
\]

\[
\frac{3x + 7}{2} = \frac{5x - 13}{2}
\]

\[
\begin{align*}
3x + 7 &= 5x - 13 \\
3x + 7 &= 5x - 13
\end{align*}
\]

\[
[x= \frac{10}{2}, x= \frac{3}{2}]
\]
\begin{verbatim}
contractSolve(b,x)
  \begin{align*}
  \pm 1 & \pm 3 \pm 3 \\
  \pm 1 & \pm 1 \pm 3 \\
  2 & 2 & 3 \\
  \end{align*}
\end{verbatim}

\begin{verbatim}
c:Fraction(Polynomial(Integer)):=((y^2+4)/(y+1))
  \begin{align*}
  2 & \\
  y & + 4 \\
  \hline \\
  y & + 1 \\
  \end{align*}
\end{verbatim}

\begin{verbatim}
radicalSolve([b,c],[x,y])
  \begin{align*}
  \pm 1 & \pm 1 \pm 3 \\
  \pm 1 & \pm 7 \pm 3 \\
  2 & 3 & 3 \\
  \end{align*}
\end{verbatim}

\begin{verbatim}
radicalSolve([b,c])
  \begin{align*}
  \pm 1 & \pm 7 \pm 3 \\
  \pm 1 & \pm 1 \pm 3 \\
  2 & 3 & 3 \\
  \end{align*}
\end{verbatim}
\[
\begin{align*}
\text{radicalSolve}([b=0,c=0],[x,y]) \\
3+++++++ ++++++++ 3+++++++ \\
- \begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\begin{array}{l}
\text{x= \ldots, } y= -2\mid\text{-1 }
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{align*}
\]
radicalRoots([b,c],[x,y])

See Also:
  o )show RadicalSolvePackage
  o )show RealSolvePackage
RadicalSolvePackage (SOLVERAD)

Exports:
contractSolve radicalRoots radicalSolve

— package SOLVERAD RadicalSolvePackage —

)abbrev package SOLVERAD RadicalSolvePackage
++ Author: P.Gianni
++ Date Created: Summer 1990
++ Date Last Updated: October 1991
++ References:
++ Description:
++ This package tries to find solutions
++ expressed in terms of radicals for systems of equations
++ of rational functions with coefficients in an integral domain R.

RadicalSolvePackage(R): Cat == Capsule where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero)
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  Z ==> Integer
  B ==> Boolean
  ST ==> String
  PR ==> Polynomial R
  UP ==> SparseUnivariatePolynomial PR
  LA ==> LocalAlgebra(PR, Z, Z)
  RF ==> Fraction PR
  RE ==> Expression R
  EQ ==> Equation
  SY ==> Symbol
  SU ==> SuchThat(List RE, List Equation RE)
  SUP ==> SparseUnivariatePolynomial
  L ==> List
  P ==> Polynomial

  SOLVEFOR ==> PolynomialSolveByFormulas(SUP RE, RE)
radicalSolve: (RF, SY) -> L EQ RE
++radicalSolve(rf,x) finds the solutions expressed in terms of
++radicals of the equation rf = 0 with respect to the symbol x,
++where rf is a rational function.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X radicalSolve(b,x)

radicalSolve: RF -> L EQ RE
++radicalSolve(rf) finds the solutions expressed in terms of
++radicals of the equation rf = 0, where rf is a
++univariate rational function.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X radicalSolve(b)

radicalSolve: (EQ RF, SY) -> L EQ RE
++radicalSolve(eq,x) finds the solutions expressed in terms of
++radicals of the equation of rational functions eq
++with respect to the symbol x.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X radicalSolve(b=0,x)

radicalSolve: EQ RF -> L EQ RE
++radicalSolve(eq) finds the solutions expressed in terms of
++radicals of the equation of rational functions eq
++with respect to the unique symbol x appearing in eq.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X radicalSolve(b=0)

radicalSolve: (L RF, L SY) -> L L EQ RE
++radicalSolve(lrf,lvar) finds the solutions expressed in terms of
++radicals of the system of equations lrf = 0 with
++respect to the list of symbols lvar,
++where lrf is a list of rational functions.
++
++X b:Fraction(Polynomial(Integer)):=((3*x^3+7)/(5*x^2-13))
++X c:Fraction(Polynomial(Integer)):=((y^2+4)/(y+1))
++X radicalSolve([b,c],[x,y])

radicalSolve: L RF -> L L EQ RE
++radicalSolve(lrf) finds the solutions expressed in terms of
++radicals of the system of equations lrf = 0, where lrf is a
++system of univariate rational functions.
++ b: Fraction(Polynomial(Integer)) := (3\times x^3 + 7)/(5\times x^2 - 13)  
++ c: Fraction(Polynomial(Integer)) := (y^2 + 4)/(y + 1)  
++ radicalSolve([b, c])

radicalSolve : (L EQ RF, L SY) -> L L EQ RE  
++ radicalSolve(leq, lvar) finds the solutions expressed in terms of  
++ radicals of the system of equations of rational functions leq  
++ with respect to the list of symbols lvar.  
++ b: Fraction(Polynomial(Integer)) := (3\times x^3 + 7)/(5\times x^2 - 13)  
++ c: Fraction(Polynomial(Integer)) := (y^2 + 4)/(y + 1)  
++ radicalSolve([b = 0, c = 0], [x, y])

radicalSolve : L EQ RF -> L L EQ RE  
++ radicalSolve(leq) finds the solutions expressed in terms of  
++ radicals of the system of equations of rational functions leq  
++ with respect to the unique symbol x appearing in leq.  
++ b: Fraction(Polynomial(Integer)) := (3\times x^3 + 7)/(5\times x^2 - 13)  
++ c: Fraction(Polynomial(Integer)) := (y^2 + 4)/(y + 1)  
++ radicalSolve([b = 0, c = 0])

radicalRoots : (RF, SY) -> L RE  
++ radicalRoots(rf, x) finds the roots expressed in terms of radicals  
++ of the rational function rf with respect to the symbol x.  
++ b: Fraction(Polynomial(Integer)) := (3\times x^3 + 7)/(5\times x^2 - 13)  
++ radicalRoots(b, x)

radicalRoots : (L RF, L SY) -> L L RE  
++ radicalRoots(lrf, lvar) finds the roots expressed in terms of  
++ radicals of the list of rational functions lrf  
++ with respect to the list of symbols lvar.  
++ b: Fraction(Polynomial(Integer)) := (3\times x^3 + 7)/(5\times x^2 - 13)  
++ c: Fraction(Polynomial(Integer)) := (y^2 + 4)/(y + 1)  
++ radicalRoots([b, c], [x, y])

contractSolve : (EQ RF, SY) -> SU  
++ contractSolve(eq, x) finds the solutions expressed in terms of  
++ radicals of the equation of rational functions eq  
++ with respect to the symbol x. The result contains new  
++ symbols for common subexpressions in order to reduce the  
++ size of the output.  
++ b: Fraction(Polynomial(Integer)) := (3\times x^3 + 7)/(5\times x^2 - 13)  
++ contractSolve(b = 0, x)

contractSolve : (RF, SY) -> SU
++contractSolve(rf,x) finds the solutions expressed in terms of
++radicals of the equation rf = 0 with respect to the symbol x,
++where rf is a rational function. The result contains new
++symbols for common subexpressions in order to reduce the
++size of the output.
++
++X b:Fraction(Polynomial(Integer)):=\frac{(3x^3+7)}{(5x^2-13)}
++X contractSolve(b,x)

Capsule ==> add
  import DegreeReductionPackage(PR, R)
  import SOLVEFOR

SideEquations: List EQ RE := []
ContractSoln: B := false

---- Local Function Declarations ----
solveInner:(PR, SY, B) -> SU
linear: UP -> List RE
quadratic: UP -> List RE
cubic: UP -> List RE
quartic: UP -> List RE
rad: PI -> RE
wrap: RE -> RE
New: RE -> RE
makeEq : (List RE,L SY) -> L EQ RE
select : L L RE -> L L RE
isGeneric? : (L PR,L SY) -> Boolean
findGenZeros : (L PR,L SY) -> L L RE
findZeros : (L PR,L SY) -> L L RE

New s ==
  s = 0 => 0
  S := new()$Symbol ::PR::RF::RE
  SideEquations := append([S = s], SideEquations)
  S

linear u == [(-coefficient(u,0))::RE /(coefficient(u,1))::RE]
quadratic u == quadratic(map(coerce,u)$UPF2)$SOLVEFOR
  cubic u == cubic(map(coerce,u)$UPF2)$SOLVEFOR
  quartic u == quartic(map(coerce,u)$UPF2)$SOLVEFOR
  rad n == n::Z::RE
  wrap s == (ContractSoln => New s; s)

---- Exported Functions ----

-- find the zeros of components in "generic" position --
findGenZeros(rlp:L PR,rlv:L SY) : L L RE ==
  pp:=rlp.first
  v:=first rlv
  rlv:=rest rlv
  res:L L RE:=[[]
    res:=append([reverse cons(r,[eval(
      (-coefficient(univariate(p,vv),0)::RE)/
      (leadingCoefficient univariate(p,vv))::RE,
      kernel(v)@Kernel(RE),r) for vv in rlv for p in rlp.rest])
      for r in radicalRoots(pp::RF,v)],res)
  res

findZeros(rlp:L PR,rlv:L SY) : L L RE ==
  parRes:=[radicalRoots(p::RF,v) for p in rlp for v in rlv]
  parRes:=select parRes
  res:L L RE:=[[]
    res1:L RE
    for par in parRes repeat
      res1:=[par.first]
      lv1:L Kernel(RE):=[kernel rlv.first]
      rlv1:=rlv.rest
      p1:=par.rest
      while p1!=[] repeat
        res1:=cons(eval(p1.first,lv1,res1),res1)
        p1:=p1.rest
        lv1:=cons(kernel rlv1.first,lv1)
        rlv1:=rlv1.rest
      res:=cons(res1,res)
  res

radicalSolve(pol:RF,v:SY) ==
  [equation(v::RE,r) for r in radicalRoots(pol,v)]

radicalSolve(p:RF) ==
  zero? p =>
    error "equation is always satisfied"
  lv:=removeDuplicates
    concat(variables numer p, variables denom p)
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  radicalSolve(p,lv.first)

radicalSolve(eq: EQ RF) ==
  radicalSolve(lhs eq -rhs eq)

radicalSolve(eq: EQ RF,v:SY) ==
  radicalSolve(lhs eq - rhs eq,v)

radicalRoots(lp: L RF,lv: L SY) ==
parRes := triangularSystems(lp, lv)$SystemSolvePackage(R)
parRes = list [] => []
-- select the components in "generic" form
rlv := reverse lv
rpRes := [reverse res for res in parRes]
listGen := [res for res in rpRes|isGeneric?(res, rlv)]
result := L L RE := []
if listGen = [] then
  result := "append" /[findGenZeros(res, rlv) for res in listGen]
  for res in listGen repeat
    rpRes := delete(rpRes, position(res, rpRes))
-- non-generic components
  rpRes = [] => result
  append("append" /[findZeros(res, rlv) for res in rpRes],
    result)

radicalSolve(lp: L RF, lv: L SY) ==
  [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

radicalSolve(lp: L RF) ==
  lv := "setUnion" / [setUnion(variables numer p, variables denom p)
    for p in lp]
  [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

radicalSolve(le: L EQ RF, lv: L SY) ==
  lp := [rhs p - lhs p for p in le]
  [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

radicalSolve(le: L EQ RF) ==
  lp := [rhs p - lhs p for p in le]
  lv := "setUnion" / [setUnion(variables numer p, variables denom p)
    for p in lp]
  [makeEq(lres, lv) for lres in radicalRoots(lp, lv)]

contractSolve(eq: EQ RF, v: SY) ==
  solveInner(numer(lhs eq - rhs eq), v, true)

contractSolve(pq: RF, v: SY) == solveInner(numer pq, v, true)

radicalRoots(pq: RF, v: SY) == lhs solveInner(numer pq, v, false)

-- test if the ideal is radical in generic position --
isGeneric?(rlp: L PR, rlv: L SY) : Boolean ==
  "and" /[degree(f, x) = 1 for f in rest rlp and x in rest rlv]

----- select the univariate factors
select(lp: L L RE) : L L RE ==
  lp = [] => list []
  [:[cons(f, lsel) for lsel in select lp.rest] for f in lp.first]
---- Local Functions ----

-- construct the equation
makeEq(nres:L RE,lv:L SY) : L EQ RE ==
[equation(x :: RE,r) for x in lv for r in nres]

solveInner(pq:PR,v:SY,contractFlag:B) ==
SideEquations ::= []
ContractSoln ::= contractFlag

factors:= factors
(factor pq)$MultivariateFactorize(SY,IndexedExponents SY,R,PR)

constants: List PR := []
unsolved: List PR := []
solutions: List RE := []

for f in factors repeat
  ff:=f.factor
  ^ member?(v, variables (ff)) =>
    constants := cons(ff, constants)
  u := univariate(ff, v)
  t := reduce u
  u := t.pol
  n := degree u
  l: List RE :=
  n = 1 => linear u
  n = 2 => quadratic u
  n = 3 => cubic u
  n = 4 => quartic u
  unsolved := cons(ff, unsolved)
  []
  for s in l repeat
    if t.deg > 1 then s := wrap s
    T0 := expand(s, t.deg)
    for i in 1..f.exponent repeat
      solutions := append(T0, solutions)
    re := SideEquations
    [solutions, SideEquations]$SU

--- SOLVERAD.dotabb ---

"SOLVERAD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVERAD"]
"ACFS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=ACFS"]
"SOLVERAD" -> "ACFS"
package RADUTIL RadixUtilities

— RadixUtilities.input —

)set break resume
/sys rm -f RadixUtilities.output
/spool RadixUtilities.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RadixUtilities
--R
--R RadixUtilities is a package constructor
--R Abbreviation for RadixUtilities is RADUTIL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RADUTIL
--R
--R----------------------------------- Operations ----------------------------------
--R radix : (Fraction(Integer),Integer) -> Any
--R
--E 1

)spool
)lisp (bye)

— RadixUtilities.help —

================================================================================
RadixUtilities examples
================================================================================

This package provides tools for creating radix expansions.

See Also:
  o )show RadixUtilities
RadixUtilities (RADUTIL)

Exports:
radix

— package RADUTIL RadixUtilities —

)abbrev package RADUTIL RadixUtilities
++ Author: Stephen M. Watt
++ Date Created: October 1986
++ Date Last Updated: May 15, 1991
++ Description:
++ This package provides tools for creating radix expansions.

RadixUtilities: Exports == Implementation where
Exports ==> with
  radix: (Fraction Integer,Integer) -> Any
  ++ radix(x,b) converts x to a radix expansion in base b.
Implementation ==> add
  radix(q, b) ==
    coerce(q :: RadixExpansion(b))$AnyFunctions1(RadixExpansion b)
package RDIST RandomDistributions

--- RandomDistributions.input ---

g) set break resume
) sys rm -f RandomDistributions.output
) spool RandomDistributions.output
) set message test on
) set message auto off
) clear all

-- S 1 of 1
) show RandomDistributions
-- R
-- R RandomDistributions(S: SetCategory) is a package constructor
-- R Abbreviation for RandomDistributions is RDIST
-- R This constructor is not exposed in this frame.
-- R Issue )edit bookvol10.4.pamphlet to see algebra source code for RDIST
-- R
-- R ----------------------------- Operations -----------------------------
-- R uniform : Set(S) -> (() -> S)
-- R rdHack1 : (Vector(S),Vector(Integer),Integer) -> (() -> S)
-- R weighted : List(Record(value: S,weight: Integer)) -> (() -> S)
-- R
-- E 1

) spool
) lisp (bye)

---

--- RandomDistributions.help ---

====================================================================
RandomDistributions examples
====================================================================

This package exports random distributions

See Also:
 o ) show RandomDistributions
RandomDistributions (RDIST)

Exports:
  uniform  rdHack1  weighted

— package RDIST RandomDistributions —

)abbrev package RDIST RandomDistributions
++ Description:
++ This package exports random distributions

RandomDistributions(S: SetCategory): with
  uniform: Set S -> (() -> S)
    ++ uniform(s) undocumented
  weighted: List Record(value: S, weight: Integer) -> (() -> S)
    ++ weighted(l) undocumented
  rdHack1: (Vector S, Vector Integer, Integer) -> (() -> S)
    ++ rdHack1(v, u, n) undocumented
== add
  import RandomNumberSource()

weighted lvw ==
  -- Collapse duplicates, adding weights.
  t: Table(S, Integer) := table()
  for r in lvw repeat
    u := search(r.value, t)
    w := (u case "failed" => 0; u::Integer)
    t.r.value := w + r.weight

  -- Construct vectors of values and cumulative weights.
  kl := keys t
  n := (#kl)::NonNegativeInteger
  n = 0 => error "Cannot select from empty set"
  kv: Vector(S) := new(n, kl.0)
  wv: Vector(Integer) := new(n, 0)

  totwt: Integer := 0
for k in kl for i in 1..n repeat
  kv.i := k
  totwt := totwt + t k
  wv.i := totwt

-- Function to generate an integer and lookup.
rdHack1(kv, wv, totwt)

rdHack1(kv, wv, totwt) ==
  w := randnum totwt
  -- do binary search in wv
  kv.1

uniform fset ==
  l := members fset
  n := #l
  l.(randnum(n)+1)

package RFDIST RandomFloatDistributions

-- RandomFloatDistributions.input

)set break resume
)sys rm -f RandomFloatDistributions.output
)spool RandomFloatDistributions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RandomFloatDistributions
--R
--R RandomFloatDistributions is a package constructor
--R Abbreviation for RandomFloatDistributions is RFDIST
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RFDIST
--R
--R------------------------------- Operations ------------------------------
--R exponential : Float -> (() -> Float) exponential1 : () -> Float
--R normal01 : () -> Float uniform01 : () -> Float
--R Beta : (NonNegativeInteger,NonNegativeInteger) -> (() -> Float)
--R F : (NonNegativeInteger,NonNegativeInteger) -> (() -> Float)
--R chiSquare : NonNegativeInteger -> (() -> Float)
--R chiSquare1 : NonNegativeInteger -> Float
--R normal : (Float,Float) -> (() -> Float)
--R t : (NonNegativeInteger) -> (() -> Float)
--R uniform : (Float,Float) -> (() -> Float)
--R
--E 1

)spool
)lisp (bye)

——

— RandomFloatDistributions.help —

====================================================================
| RandomFloatDistributions examples |
====================================================================

This package exports random floating-point distributions

See Also:
 o )show RandomFloatDistributions
RandomFloatDistributions (RFDIST)

Exports:

\begin{verbatim}
Beta  chiSquare  chiSquare1  exponential  exponential1
F    normal    t          normal01    uniform01
uniform
\end{verbatim}

--- package RFDIST RandomFloatDistributions ---

\begin{verbatim}
)abbrev package RFDIST RandomFloatDistributions
++ Description:
++ This package exports random floating-point distributions

RationalNumber==> Fraction Integer
RandomFloatDistributions(): Cat == Body where
  NNI ==> NonNegativeInteger

  Cat ==> with
  uniform01: () -> Float
    ++ uniform01() \ undocumented
  normal01: () -> Float
    ++ normal01() \ undocumented
  exponential1(): -> Float
    ++ exponential1() \ undocumented
  chiSquare1: NNI -> Float
    ++ chiSquare1(n) \ undocumented

  uniform: (Float, Float) -> (() -> Float)
    ++ uniform(f,g) \ undocumented
  normal:     (Float, Float) -> (() -> Float)
    ++ normal(f,g) \ undocumented
  exponential: (Float) -> (() -> Float)
    ++ exponential(f) \ undocumented
  chiSquare:  (NNI) -> (() -> Float)
    ++ chiSquare(n) \ undocumented
  Beta:      (NNI, NNI) -> (() -> Float)
    ++ Beta(n,m) \ undocumented
\end{verbatim}
F: (NNI, NNI) -> (() -> Float)
++ F(n,m) undocumented

t: (NNI) -> (() -> Float)
++ t(n) undocumented

Body ==> add
import RandomNumberSource()
-- FloatPackage0()

-- random() generates numbers in 0..rnmax
rnmax := (size()$RandomNumberSource() - 1)::Float

uniform01() ==
  randnum()::Float/rnmax
uniform(a,b) ==
  a + uniform01()*(b-a)

exponential1() ==
  u: Float := 0
  -- This test should really be u < m where m is
  -- the minimum acceptable argument to log.
  while u = 0 repeat u := uniform01()
  - log u
exponential(mean) ==
  mean*exponential1()

-- This method is correct but slow.
normal01() ==
  s := 2::Float
  while s >= 1 repeat
    v1 := 2 * uniform01() - 1
    v2 := 2 * uniform01() - 1
    s := v1**2 + v2**2
  v1 * sqrt(-2 * log s/s)
normal(mean, stdev) ==
  mean + stdev*normal01()

chiSquare1 dgfree ==
  x: Float := 0
  for i in 1..dgfree quo 2 repeat
    x := x + 2*exponential1()
  if odd? dgfree then
    x := x + normal01()**2
  x
chiSquare dgfree ==
  chiSquare1 dgfree

Beta(dgfree1, dgfree2) ==
  y1 := chiSquare1 dgfree1
\begin{dmath*}
y_2 := \text{chiSquare1} \ d_{\text{free}2} \\
y_1/(y_1 + y_2)
\end{dmath*}

\begin{dmath*}
F(d_{\text{free}1}, \ d_{\text{free}2}) ==
\begin{align*}
y_1 & := \text{chiSquare1} \ d_{\text{free}1} \\
y_2 & := \text{chiSquare1} \ d_{\text{free}2} \\
(d_{\text{free}2} \times y_1)/(d_{\text{free}1} \times y_2)
\end{align*}
\end{dmath*}

\begin{dmath*}
t \ d_{\text{free}} ==
\begin{align*}
n & := \text{normal01}() \\
d & := \text{chiSquare1}(d_{\text{free}}) / (d_{\text{free}}::\text{Float}) \\
n / \sqrt{d}
\end{align*}
\end{dmath*}

---

\textit{RFDIST.dotabb} ---

"RFDIST" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFDIST"]
"Package" [color="#FF4488"]
"RFDIST" -> "Package"

---

package RIDIST RandomIntegerDistributions

--- RandomIntegerDistributions.input ---

\texttt{)set break resume}
\texttt{)sys rm -f RandomIntegerDistributions.output}
\texttt{)spool RandomIntegerDistributions.output}
\texttt{)set message test on}
\texttt{)set message auto off}
\texttt{)clear all}

\texttt{--S 1 of 1}
\texttt{)show RandomIntegerDistributions}
\texttt{--R}
\texttt{--R RandomIntegerDistributions is a package constructor}
\texttt{--R Abbreviation for RandomIntegerDistributions is RIDIST}
\texttt{--R This constructor is not exposed in this frame.}
\texttt{--R Issue \texttt{)edit bookvol10.4.pamphlet} to see algebra source code for RIDIST}
\texttt{--R}
\texttt{--R----------------------------- Operations -----------------------------}
\texttt{--R binomial : (Integer,\text{RationalNumber}) \to (() \to \text{Integer})}
--R geometric : RationalNumber -> (() -> Integer)
--R poisson : RationalNumber -> (() -> Integer)
--R ridHack1 : (Integer,Integer,Integer,Integer) -> Integer
--R uniform : Segment(Integer) -> (() -> Integer)

)spool
)lisp (bye)

RandomIntegerDistributions (RIDIST)

Exports:
  binomial geometric poisson ridHack1 uniform

--- package RIDIST RandomIntegerDistributions ---

)abbrev package RIDIST RandomIntegerDistributions
++ Description:
++ This package exports integer distributions

RandomIntegerDistributions(): with
  uniform: Segment Integer -> (() -> Integer)
    ++ uniform(s) as
    ++ 1 + u0 + w*u1 + w**2*u2 +...+ w**(n-1)*u-1 + w**n*m
    ++ where
    ++  s = a..b
    ++  l = min(a,b)
    ++  m = abs(b-a) + 1
    ++  w**n < m < w**(n+1)
    ++  u0,...,un-1 are uniform on 0..w-1
    ++  m is uniform on 0..(m quo w**n)-1
  binomial: (Integer, RationalNumber) -> (() -> Integer)
    ++ binomial(n,f) \undocumented
  poisson: RationalNumber -> (() -> Integer)
    ++ poisson(f) \undocumented
  geometric: RationalNumber -> (() -> Integer)
    ++ geometric(f) \undocumented
  ridHack1: (Integer,Integer,Integer,Integer) -> Integer
    ++ ridHack1(i,j,k,l) \undocumented

== add
import RandomNumberSource()
import IntegerBits()

uniform aTob ==
  a := lo aTob; b := hi aTob
  l := min(a,b); m := abs(a-b) + 1
  w := 2**(bitLength size() quo 2)::NonNegativeInteger

  n := 0
  mq := m -- m quo w**n
  while (mqnext := mq quo w) > 0 repeat
    n := n + 1
    mq := mqnext
    ridHack1(mq, n, w, l)

  ridHack1(mq, n, w, l) ==
    r := randnum mq
    for i in 1..n repeat r := r*w + randnum w
    r + 1

— RIDIST.dotabb —
package RANDSRC RandomNumberSource

-- RandomNumberSource.input --

)set break resume
)sys rm -f RandomNumberSource.output
)spool RandomNumberSource.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show RandomNumberSource
--R
--R RandomNumberSource is a package constructor
--R Abbreviation for RandomNumberSource is RANDSRC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RANDSRC
--R
--R-------------------------------- Operations --------------------------------
--R randnum : () -> Integer       randnum : Integer -> Integer
--R reseed : Integer -> Void      seed : () -> Integer
--R size : () -> Integer
--R
--E 1

)spool
)lisp (bye)

-- RandomNumberSource.help --

====================================================================
RandomNumberSource examples
====================================================================

All random numbers used in the system should originate from
the same generator. This package is intended to be the source.
RandomNumberSource (RANDSRC)

Exports:
randnum  reseed  seed  size

--- package RANDSRC RandomNumberSource ---

)abbrev package RANDSRC RandomNumberSource
++ Author:S.M.Watt
++ Date Created: April 87
++ Date Last Updated:Jan 92, May 1995 (MCD)
++ Description:
++ Random number generators.
++ All random numbers used in the system should originate from
++ the same generator. This package is intended to be the source.
--
-- Possible improvements:
-- 1) Start where the user left off
-- 2) Be able to switch between methods in the random number source.

RandomNumberSource(): with
  -- If r := randnum() then 0 <= r < size().
  randnum: () -> Integer
    ++ randnum() is a random number between 0 and size().
  -- If r := randnum() then 0 <= r < size().
  size: () -> Integer
    ++ size() is the base of the random number generator
-- If r := randnum n and n <= size() then 0 <= r < n.

randnum: Integer -> Integer
  ++ randnum(n) is a random number between 0 and n.

reseed: Integer -> Void
  ++ reseed(n) restarts the random number generator at n.

seed : () -> Integer
  ++ seed() returns the current seed value.

== add
-- This random number generator passes the spectral test
-- with flying colours. [Knuth vol2, 2nd ed, p105]

ranbase: Integer := 2**31-1
x0: Integer := 1231231231
x1: Integer := 3243232987

randnum() ==
t := (271828183 * x1 - 314159269 * x0) rem ranbase
if t < 0 then t := t + ranbase
x0:= x1
x1:= t

size() == ranbase
reseed n ==
x0 := n rem ranbase
  -- x1 := (n quo ranbase) rem ranbase
  x1 := n quo ranbase

seed() == x1*ranbase + x0

-- Compute an integer in 0..n-1.
randnum n ==
  (n * randnum()) quo ranbase

---
--- RANDSRC.dotabb ---

"RANDSRC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RANDSRC"]
"ALGEBRA-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALGEBRA"]
"RANDSRC" -> "ALGEBRA-"
package RATFACT RationalFactorize

--- RationalFactorize.input ---

)set break resume
)sys rm -f RationalFactorize.output
)spool RationalFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFactorize
--R
--R RationalFactorize(RP: UnivariatePolynomialCategory(Fraction(Integer))) is a package constructor
--R Abbreviation for RationalFactorize is RATFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RATFACT
--R
--R----------------------------- Operations -----------------------------
--R factor : RP -> Factored(RP)
--R factorSquareFree : RP -> Factored(RP)
--R
--E 1

)spool
)lisp (bye)

---

--- RationalFactorize.help ---

====================================================================
RationalFactorize examples
====================================================================

Factorization of extended polynomials with rational coefficients.
This package implements factorization of extended polynomials whose
coefficients are rational numbers. It does this by taking the lcm of
the coefficients of the polynomial and creating a polynomial with
integer coefficients. The algorithm in GaloisGroupFactorizer is then
used to factor the integer polynomial. The result is normalized with
respect to the original lcm of the denominators.

See Also:
  o )show RationalFactorize
RationalFactorize (RATFACT)

Exports:
  factor  factorSquareFree

— package RATFACT RationalFactorize —

)abbrev package RATFACT RationalFactorize
++ Author: P. Gianni
++ Date last updated: December 1993
++ Description:
  ++ Factorization of extended polynomials with rational coefficients.
  ++ This package implements factorization of extended polynomials
  ++ whose coefficients are rational numbers. It does this by taking the
  ++ lcm of the coefficients of the polynomial and creating a polynomial
  ++ with integer coefficients. The algorithm in
  ++ \spadtype{GaloisGroupFactorizer} is then
  ++ used to factor the integer polynomial. The result is normalized
  ++ with respect to the original lcm of the denominators.

RationalFactorize(RP) : public == private where
I  ===> Integer
RN ===> Fraction Integer
BP ===> SparseUnivariatePolynomial(I)
RP : UnivariatePolynomialCategory RN

public  ===> with

  factor       : RP ->  Factored RP
     ++ factor(p) factors an extended polynomial p over the rational numbers.
  factorSquareFree : RP -> Factored RP
++ factorSquareFree(p) factors an extended squareFree
++ polynomial p over the rational numbers.

private ==> add
import GaloisGroupFactorizer (BP)
ParFact ==> Record(irr:BP,pow:I)
FinalFact ==> Record(contp:I,factors:List(ParFact))
URNI ==> UnivariatePolynomialCategoryFunctions2(RN,RP,I,BP)
UIRN ==> UnivariatePolynomialCategoryFunctions2(I,BP,RN,RP)
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FFE ==> Record(flg:fUnion, fctr:RP, xpnt:I)

factor(p:RP) : Factored(RP) ==
pden: I := lcm([denom c for c in coefficients p])
pol : RP := pden*p
ipol: BP := map(numer,pol)$URNI
ffact: FinalFact := henselFact(ipol,false)
makeFR(((ffact.contp)/pden)::RP,
    [["prime",map(coerce,u.irr)$UIRN,u.pow]$FFE
    for u in ffact.factors])

factorSquareFree(p:RP) : Factored(RP) ==
pden: I := lcm([denom c for c in coefficients p])
pol : RP := pden*p
ipol: BP := map(numer,pol)$URNI
ffact: FinalFact := henselFact(ipol,true)
makeFR(((ffact.contp)/pden)::RP,
    [["prime",map(coerce,u.irr)$UIRN,u.pow]$FFE
    for u in ffact.factors])

package RF RationalFunction

— RationalFunction.input —
(set break resume)
(sys rm -f RationalFunction.output)
(spool RationalFunction.output)
(set message test on)
(set message auto off)
clear all

(--S 1 of 1)
(show RationalFunction)
(--R)
|--R RationalFunction(R: IntegralDomain) is a package constructor
|--R Abbreviation for RationalFunction is RF
|--R This constructor is exposed in this frame.
|--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RF
|--R

|--R--------------------------------------------------------------- Operations ---------------------------------------------------------------
|--R coerce : R -> Fraction(Polynomial(R))
|--R eval : (Fraction(Polynomial(R)),Symbol,Fraction(Polynomial(R)))) -> Fraction(Polynomial(R))
|--R eval : (Fraction(Polynomial(R)),List(Symbol),List(Fraction(Polynomial(R)))) -> Fraction(Polynomial(R))
|--R eval : (Fraction(Polynomial(R)),Equation(Fraction(Polynomial(R)))) -> Fraction(Polynomial(R))
|--R eval : (Fraction(Polynomial(R)),List(Equation(Fraction(Polynomial(R))))) -> Fraction(Polynomial(R))
|--R mainVariable : Fraction(Polynomial(R)) -> Union(Symbol,”failed”)
|--R multivariate : (Fraction(SparseUnivariatePolynomial(Fraction(Polynomial(R)))),Symbol) -> Fraction(Polynomial(R))
|--R univariate : (Fraction(Polynomial(R)),Symbol) -> Fraction(SparseUnivariatePolynomial(Fraction(Polynomial(R))))
|--R variables : Fraction(Polynomial(R)) -> List(Symbol)
|--R

--)E 1

(spool)
(lisp (bye)

---

--- RationalFunction.help ---

====================================================================
RationalFunction examples
====================================================================

Utilities that provide the same top-level manipulations on fractions than on polynomials.

See Also:
- )show RationalFunction

---
RationalFunction (RF)

Exports:
coerce eval mainVariable multivariate univariate variables

--- package RF RationalFunction ---

)abbrev package RF RationalFunction
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 18 April 1991
++ Description:
++ Utilities that provide the same top-level manipulations on
++ fractions than on polynomials.
++ Do not make into a domain!

RationalFunction(R: IntegralDomain): Exports == Implementation where
V ==> Symbol
P ==> Polynomial R
Q ==> Fraction P
QF ==> PolynomialCategoryQuotientFunctions(IndexedExponents Symbol,
Symbol, R, P, Q)

Exports ==> with
variables : Q -> List V
++ variables(f) returns the list of variables appearing
++ in the numerator or the denominator of f.
mainVariable: Q -> Union(V, "failed")
++ mainVariable(f) returns the highest variable appearing
++ in the numerator or the denominator of f, "failed" if
++ f has no variables.
univariate : (Q, V) -> Fraction SparseUnivariatePolynomial Q
++ univariate(f, v) returns f viewed as a univariate
++ rational function in v.
multivariate: (Fraction SparseUnivariatePolynomial Q, V) -> Q
++ multivariate(f, v) applies both the numerator and
++ denominator of f to v.
eval : (Q, V, Q) -> Q
++ eval(f, v, g) returns f with v replaced by g.
eval : (Q, List V, List Q) -> Q
++ eval(f, [v1, ..., vn], [g1, ..., gn]) returns f with
++ each vi replaced by gi in parallel, i.e. vi’s appearing
++ inside the gi’s are not replaced.
eval : (Q, Equation Q) -> Q
++ eval(f, v = g) returns f with v replaced by g.
++ Error: if v is not a symbol.
eval : (Q, List Equation Q) -> Q
++ eval(f, [v1 = g1, ..., vn = gn]) returns f with
++ each vi replaced by gi in parallel, i.e. vi’s appearing
++ inside the gi’s are not replaced.
++ Error: if any vi is not a symbol.
coerce : R -> Q
++ coerce(r) returns r viewed as a rational function over R.

Implementation ==> add
foo : (List V, List Q, V) -> Q
peval: (P, List V, List Q) -> Q
coerce(r:R):Q == r::P::Q
variables f == variables(f)$QF
mainVariable f == mainVariable(f)$QF
univariate(f, x) == univariate(f, x)$QF
multivariate(f, x) == multivariate(f, x)$QF
eval(x:Q, s:V, y:Q) == eval(x, [s], [y])
eval(x:Q, eq:Equation Q) == eval(x, [eq])
foo(ls, lv, x) == match(ls, lv, x, x::Q)$ListToMap(V, Q)
eval(x:Q, l:List Equation Q) ==
eval(x, [retract(lhs eq)@V for eq in l]$List(V),
[rhs eq for eq in l]$List(Q))
eval(x:Q, ls:List V, lv:List Q) ==
peval(numer x, ls, lv) / peval(denom x, ls, lv)
peval(p, ls, lv) ==
map(z1 ++-> foo(ls, lv, z1), z2 ++-> z2::Q,p)
$PolynomialCategoryLifting(IndexedExponents V,V,R,P,Q)

—— RF.dotabb ——
package DEFINTRF RationalFunctionDefiniteIntegration

--- RationalFunctionDefiniteIntegration.input ---

)set break resume
)sys rm -f RationalFunctionDefiniteIntegration.output
)spool RationalFunctionDefiniteIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFunctionDefiniteIntegration

--R RationalFunctionDefiniteIntegration(R: Join(EuclideanDomain,OrderedSet,CharacteristicZero,RetractableTo(Integer),LinearlyExplicitRingOver(Integer))) is a package constructor

--R Abbreviation for RationalFunctionDefiniteIntegration is DEFINTRF

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DEFINTRF

--R

--R------------------------------------------------- Operations -------------------------------

--R integrate : (Fraction(Polynomial(R)),SegmentBinding(OrderedCompletion(Expression(R)))) -> Union(f1: OrderedCompletion(Expression(R)),f2: List(OrderedCompletion(Expression(R))),fail: failed,pole: potentialPole)

--R integrate : (Fraction(Polynomial(R)),SegmentBinding(OrderedCompletion(Expression(R))),String) -> Union(f1: OrderedCompletion(Expression(R)),f2: List(OrderedCompletion(Expression(R))),fail: failed,pole: potentialPole)

--R integrate : (Fraction(Polynomial(R)),SegmentBinding(OrderedCompletion(Fraction(Polynomial(R))))) -> Union(f1: OrderedCompletion(Expression(R)),f2: List(OrderedCompletion(Expression(R))),fail: failed,pole: potentialPole)

--R integrate : (Fraction(Polynomial(R)),SegmentBinding(OrderedCompletion(Fraction(Polynomial(R)))),String) -> Union(f1: OrderedCompletion(Expression(R)),f2: List(OrderedCompletion(Expression(R))),fail: failed,pole: potentialPole)

--R

--)spool
)lisp (bye)

---

--- RationalFunctionDefiniteIntegration.help ---

====================================================================
RationalFunctionDefiniteIntegration examples
====================================================================

Definite integration of rational functions.
RationalFunctionDefiniteIntegration provides functions to compute
definite integrals of rational functions.
See Also:
  o )show RationalFunctionDefiniteIntegration

---

RationalFunctionDefiniteIntegration (DEFINTRF)

Exports:
integrate

--- package DEFINTRF RationalFunctionDefiniteIntegration ---

)abbrev package DEFINTRF RationalFunctionDefiniteIntegration
++ Author: Manuel Bronstein
++ Date Created: 2 October 1989
++ Date Last Updated: 2 February 1993
++ Description:
++ Definite integration of rational functions.
++ \spadtype{RationalFunctionDefiniteIntegration} provides functions to
++ compute definite integrals of rational functions.

RationalFunctionDefiniteIntegration(R):Exports == Implementation where
  R : Join(EuclideanDomain, OrderedSet, CharacteristicZero,
           RetractableTo Integer, LinearlyExplicitRingOver Integer)

SE ==> Symbol
RF ==> Fraction Polynomial R
FE ==> Expression R
ORF ==> OrderedCompletion RF
OFE ==> OrderedCompletion FE
U ==> Union(f1:OFE, f2:List OFE, fail:"failed", pole:"potentialPole")

Exports ==> with
integrate: (RF, SegmentBinding OFE) -> U
++ integrate(f, x = a .. b) returns the integral of
++ integral of \( \int_{a}^{b} f(x) \, dx \) from a to b.
++ Error: if f has a pole for x between a and b.
integrate: (RF, SegmentBinding OFE, String) -> U
++ integrate(f, x = a .. b, "noPole") returns the
++ integral of \( \int_{a}^{b} f(x) \, dx \) from a to b.
++ If it is not possible to check whether f has a pole for x
++ between a and b (because of parameters), then this function
++ will assume that f has no such pole.
++ Error: if f has a pole for x between a and b or
++ if the last argument is not "noPole".

-- the following two are contained in the above, but they are for the
-- interpreter... DO NOT COMMENT OUT UNTIL THE INTERPRETER IS BETTER!

integrate: (RF, SegmentBinding ORF) -> U
++ integrate(f, x = a .. b) returns the integral of
++ integral of \( \int_{a}^{b} f(x) \, dx \) from a to b.
++ Error: if f has a pole for x between a and b.

Implementation ==> add
import DefiniteIntegrationTools(R, FE)
import IntegrationResultRFToFunction(R)
import OrderedCompletionFunctions2(RF, FE)

int : (RF, SE, OFE, OFE, Boolean) -> U
nopole: (RF, SE, OFE, OFE) -> U

integrate(f:RF, s:SegmentBinding OFE) ==
int(f, variable s, lo segment s, hi segment s, false)

nopole(f, x, a, b) ==
k := kernel(x)@Kernel(FE)
(u := integrate(f, x)) case FE =>
  (v := computeInt(k, u::FE, a, b, true)) case "failed" => ["failed"]
[v::OFE]
ans := empty()$List(OFE)
for g in u::List(FE) repeat
  (v := computeInt(k, g, a, b, true)) case "failed" => return ["failed"]
  ans := concat!(ans, [v::OFE])
[ans]

integrate(f:RF, s:SegmentBinding ORF) ==
int(f, variable s, map(x +-> x::FE, lo segment s),
    map(x +-> x::FE, hi segment s), false)

integrate(f:RF, s:SegmentBinding ORF, str:String) ==
int(f, variable s, map(x +-> x::FE, lo segment s),
    map(x +-> x::FE, hi segment s), ignore? str)

integrate(f:RF, s:SegmentBinding OVE, str:String) ==
int(f, variable s, lo segment s, hi segment s, ignore? str)

int(f, x, a, b, ignor?) ==
a = b => [0::OFE]
(z := checkForZero(denom f, x, a, b, true)) case "failed" =>
    ignor? => nopole(f, x, a, b)
["potentialPole"]
z::Boolean => error "integrate: pole in path of integration"
nopole(f, x, a, b)

package RFFACT RationalFunctionFactor

--S 1 of 1
)show RationalFunctionFactor
--R
--R RationalFunctionFactor(UP: UnivariatePolynomialCategory(Fraction(Polynomial(Integer)))) is
--R Abbreviation for RationalFunctionFactor is RFACT
--- R This constructor is exposed in this frame.
--- R Issue )edit bookvol10.4.pamphlet to see algebra source code for RFFACT
--- R
--- R--------------------------- Operations ----------------------------
--- R factor : UP -> Factored(UP)
--- R
--- E 1

)spool
)lisp (bye)

---

RationalFunctionFactor (RFFACT)

Exports:
factor

--- package RFFACT RationalFunctionFactor ---
RationalFunctionFactor(UP): Exports == Implementation where
    UP: UnivariatePolynomialCategory Fraction Polynomial Integer

    SE ==> Symbol
    P ==> Polynomial Integer
    RF ==> Fraction P
    UPCF2 ==> UnivariatePolynomialCategoryFunctions2

    Exports == with
        factor: UP -> Factored UP
            ++ factor(p) returns a prime factorisation of p.

    Implementation == add
        likuniv: (P, SE, P) -> UP

        dummy := new()$SE

        likuniv(p, x, d) ==
            map(y +-> y/d, univariate(p, x))$UPCF2(P, SparseUnivariatePolynomial P, RF, UP)

        factor p ==
            d := denom(q := elt(p, dummy::P :: RF))
            map(x +-> likuniv(x, dummy, d),
                factor(numer q)$MultivariateFactorize(SE, IndexedExponents SE, Integer, P))$FactoredFunctions2(P, UP)

package RFFACTOR RationalFunctionFactorizer
RATIONALFUNCTIONFACTORIZER

--- RationalFunctionFactorizer.input ---

)set break resume
)sys rm -f RationalFunctionFactorizer.output
)spool RationalFunctionFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFunctionFactorizer

--R RationalFunctionFactorizer(R: EuclideanDomain) is a package constructor
--R Abbreviation for RationalFunctionFactorizer is RFFACTOR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RFFACTOR
--R
--R------------------------------- Operations --------------------------------
--R factorFraction : Fraction(Polynomial(R)) -> Fraction(Factored(Polynomial(R)))
--R
--E 1

)spool
)lisp (bye)

--- RationalFunctionFactorizer.help ---

====================================================================
RationalFunctionFactorizer examples
====================================================================

RationalFunctionFactorizer contains the factor function (called
factorFraction) which factors fractions of polynomials by factoring
the numerator and denominator. Since any non zero fraction is a unit
the usual factor operation will just return the original fraction.

See Also:
o )show RationalFunctionFactorizer
RationalFunctionFactorizer (RFFACTOR)

Exports:
factor

— package RFFACTOR RationalFunctionFactorizer —

)abbrev package RFFACTOR RationalFunctionFactorizer
++ Author: P. Gianni
++ Date Last Updated: March 1995
++ Description:
++ \spadtype{RationalFunctionFactorizer} contains the factor function
++ (called factorFraction) which factors fractions of polynomials by factoring
++ the numerator and denominator. Since any non zero fraction is a unit
++ the usual factor operation will just return the original fraction.

RationalFunctionFactorizer(R) : C == T
where
  R : EuclideanDomain -- R with factor for R[X]
  P == Polynomial R
  FP == Fraction P
  SE == Symbol

C == with
  factorFraction : FP -> Fraction Factored(P)
  ++ factorFraction(r) factors the numerator and the denominator of
  ++ the polynomial fraction r.

T == add

factorFraction(p:FP) : Fraction Factored(P) ==
  R is Fraction Integer =>
    MR:=MRationalFactorize(IndexedExponents SE,SE, Integer,P)
    (factor(numer p)$MR)/ (factor(denom p)$MR)
  R has FiniteFieldCategory =>
    FF:=MultFiniteFactorize(SE,IndexedExponents SE,R,P)
PACKAGE INTRF RATIONALFUNCTIONINTEGRATION

3635

(factor(numer p))$FF/(factor(denom p))$FF
R has CharacteristicZero =>
MFF:=MultivariateFactorize(SE,IndexedExponents SE,R,P)
(factor(numer p))$MFF/(factor(denom p))$MFF
error "case not handled"

———— RFFACTOR.dotabb —
"RFFACTOR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RFFACTOR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RFFACTOR" -> "PFECAT"

———-

package INTRF RationalFunctionIntegration
— RationalFunctionIntegration.input —
)set break resume
)sys rm -f RationalFunctionIntegration.output
)spool RationalFunctionIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFunctionIntegration
--R
--R RationalFunctionIntegration(F: Join(IntegralDomain,RetractableTo(Integer),CharacteristicZero)) is a pac
--R Abbreviation for RationalFunctionIntegration is INTRF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTRF
--R
--R------------------------------- Operations ---------------------------------R extendedIntegrate : (Fraction(Polynomial(F)),Symbol,Fraction(Polynomial(F))) -> Union(Record(ratpart: Fr
--R infieldIntegrate : (Fraction(Polynomial(F)),Symbol) -> Union(Fraction(Polynomial(F)),"failed")
--R internalIntegrate : (Fraction(Polynomial(F)),Symbol) -> IntegrationResult(Fraction(Polynomial(F)))
--R limitedIntegrate : (Fraction(Polynomial(F)),Symbol,List(Fraction(Polynomial(F)))) -> Union(Record(mainpa
--R
--E 1


RationalFunctionIntegration examples

This package provides functions for the integration of rational functions.

See Also:
   o )show RationalFunctionIntegration

RationalFunctionIntegration (INTRF)

Exports:
   extendedIntegrate infieldIntegrate internalIntegrate limitedIntegrate

   — package INTRF RationalFunctionIntegration —

   )abbrev package INTRF RationalFunctionIntegration
   ++ Author: Manuel Bronstein
   ++ Date Created: 1987
   ++ Date Last Updated: 29 Mar 1990
   ++ Description:
   ++ This package provides functions for the integration of rational functions.
RationalFunctionIntegration(F): Exports == Implementation where
F: Join(IntegralDomain, RetractableTo Integer, CharacteristicZero)

SE ==> Symbol
P ==> Polynomial F
Q ==> Fraction P
UP ==> SparseUnivariatePolynomial Q
QF ==> Fraction UP
LGQ ==> List Record(coeff:Q, logand:Q)
UQ ==> Union(Record(ratpart:Q, coeff:Q), "failed")
ULQ ==> Union(Record(mainpart:Q, limitedlogs:LGQ), "failed")

Exports ==> with
internalIntegrate: (Q, SE) -> IntegrationResult Q
++ internalIntegrate(f, x) returns g such that \( \frac{dg}{dx} = f \).
infieldIntegrate : (Q, SE) -> Union(Q, "failed")
++ infieldIntegrate(f, x) returns a fraction
++ g such that \( \frac{dg}{dx} = f \)
++ if g exists, "failed" otherwise.
limitedIntegrate : (Q, SE, List Q) -> ULQ
++ \( \text{limitedIntegrate(f, x, [g_1,...,g_n])} \) returns fractions
++ \( \text{[h, [c_i, g_i]]} \) such that the g_i's are among
++ \( \text{[g_1,...,g_n]}, \)
++ \( \frac{d(c_i)}{dx} = 0 \), and \( \frac{dh + \text{sum}(c_i \log(g_i))}{dx} = f \)
++ if possible, "failed" otherwise.
extendedIntegrate: (Q, SE, Q) -> UQ
++ extendedIntegrate(f, x, g) returns fractions \( \text{[h, c]} \) such that
++ \( \frac{dc}{dx} = 0 \) and \( \frac{dh}{dx} = f - cg \), if \( \text{[h, c]} \) exist,
++ "failed" otherwise.

Implementation ==> add
import RationalIntegration(Q, UP)
import IntegrationResultFunctions2(QF, Q)
import PolynomialCategoryQuotientFunctions(IndexedExponents SE, SE, F, P, Q)

infieldIntegrate(f, x) ==
map(x1 +-> multivariate(x1, x), infieldint univariate(f, x))

internalIntegrate(f, x) ==
map(x1 +-> multivariate(x1, x), integrate univariate(f, x))

extendedIntegrate(f, x, g) ==
map(x1 +-> multivariate(x1, x),
extendedint(univariate(f, x), univariate(g, x)))

limitedIntegrate(f, x, lu) ==
map(x1 +-> multivariate(x1, x),
limitedint(univariate(f, x), [univariate(u, x) for u in lu]))
3638

CHAPTER 19. CHAPTER R
———— INTRF.dotabb —

"INTRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTRF" -> "PFECAT"

———-

package LIMITRF RationalFunctionLimitPackage
— RationalFunctionLimitPackage.input —
)set break resume
)sys rm -f RationalFunctionLimitPackage.output
)spool RationalFunctionLimitPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalFunctionLimitPackage
--R
--R RationalFunctionLimitPackage(R: GcdDomain) is a package constructor
--R Abbreviation for RationalFunctionLimitPackage is LIMITRF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for LIMITRF
--R
--R------------------------------- Operations ---------------------------------R complexLimit : (Fraction(Polynomial(R)),Equation(OnePointCompletion(Polynomial(R)))) -> OneP
--R complexLimit : (Fraction(Polynomial(R)),Equation(Fraction(Polynomial(R)))) -> OnePointComple
--R limit : (Fraction(Polynomial(R)),Equation(OrderedCompletion(Polynomial(R)))) -> Union(Ordere
--R limit : (Fraction(Polynomial(R)),Equation(Fraction(Polynomial(R)))) -> Union(OrderedCompleti
--R limit : (Fraction(Polynomial(R)),Equation(Fraction(Polynomial(R))),String) -> Union(OrderedC
--R
--E 1
)spool
)lisp (bye)

———— RationalFunctionLimitPackage.help —


RationalFunctionLimitPackage examples

Computation of limits for rational functions.

See Also:
- )show RationalFunctionLimitPackage

---

RationalFunctionLimitPackage (LIMITRF)

Exports:
- complexLimit
- limit

--- package LIMITRF RationalFunctionLimitPackage ---

)abbrev package LIMITRF RationalFunctionLimitPackage
++ Author: Manuel Bronstein
++ Date Created: 4 October 1989
++ Date Last Updated: 26 November 1991
++ Description:
++ Computation of limits for rational functions.

RationalFunctionLimitPackage(R:GcdDomain):Exports==Implementation where
- Z ==> Integer
- P ==> Polynomial R
- RF ==> Fraction P
- EQ ==> Equation
- ORF ==> OrderedCompletion RF
- OPF ==> OnePointCompletion RF
- UP ==> SparseUnivariatePolynomial RF
SE ==> Symbol
QF ==> Fraction SparseUnivariatePolynomial RF
Result ==> Union(ORF, "failed")
TwoSide ==> Record(leftHandLimit:Result, rightHandLimit:Result)
U ==> Union(ORF, TwoSide, "failed")
RFSGN ==> RationalFunctionSign(R)

Exports ==> with
-- The following are the one we really want, but the interpreter cannot
-- handle them...
-- limit: (RF,EQ ORF) -> U
-- ++ limit(f(x),x,a) computes the real two-sided limit lim(x -> a,f(x))
-- complexLimit: (RF,EQ OPF) -> OPF
-- ++ complexLimit(f(x),x,a) computes the complex limit lim(x -> a,f(x))

-- ... so we replace them by the following 4:
limit: (RF,EQ OrderedCompletion P) -> U
++ limit(f(x),x,a) computes the real two-sided limit
++ of f as its argument x approaches \spad{a}.
limit: (RF,EQ RF) -> U
++ limit(f(x),x,a) computes the real two-sided limit
++ of f as its argument x approaches \spad{a}.
complexLimit: (RF,EQ OnePointCompletion P) -> OPF
++ \spad{complexLimit(f(x),x,a)} computes the complex limit
++ of \spad{f} as its argument x approaches \spad{a}.
complexLimit: (RF,EQ RF) -> OPF
++ complexLimit(f(x),x,a) computes the complex limit
++ of f as its argument x approaches \spad{a}.
limit: (RF,EQ RF,String) -> Result
++ limit(f(x),x,a,"left") computes the real limit
++ of f as its argument x approaches \spad{a} from the left;
++ limit(f(x),x,a,"right") computes the corresponding limit as x
++ approaches \spad{a} from the right.

Implementation ==> add
import ToolsForSign R
import InnerPolySign(RF, UP)
import RFSGN
import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
SE, R, P, RF)

finiteComplexLimit: (QF, RF) -> OPF
finiteLimit : (QF, RF) -> U
fLimit : (Z, UP, RF, Z) -> Result

-- These 2 should be exported, see comment above
locallimit : (RF, SE, ORF) -> U
locallimitcomplex: (RF, SE, OPF) -> OPF
limit(f:RF, eq: EQ RF) ==
  (xx := retractIfCan(lhs eq) @ Union(SE, "failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE; a := rhs eq
  locallimit(f, x, a :: ORF)

complexLimit(f:RF, eq: EQ RF) ==
  (xx := retractIfCan(lhs eq) @ Union(SE, "failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE; a := rhs eq
  locallimitcomplex(f, x, a :: OPF)

limit(f:RF, eq: EQ OrderedCompletion P) ==
  (p := retractIfCan(lhs eq) @ Union(P, "failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  (xx := retractIfCan(p) @ Union(SE, "failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE
  a := map(y +-> y :: RF, rhs eq) @ OrderedCompletionFunctions2(P, RF)
  locallimit(f, x, a)

complexLimit(f:RF, eq: EQ OnePointCompletion P) ==
  (p := retractIfCan(lhs eq) @ Union(P, "failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  (xx := retractIfCan(p) @ Union(SE, "failed")) case "failed" =>
    error "limit: left hand side must be a variable"
  x := xx :: SE
  a := map(y +-> y :: RF, rhs eq) @ OnePointCompletionFunctions2(P, RF)
  locallimitcomplex(f, x, a)

fLimit(n, d, a, dir) ==
  (s := signAround(d, a, dir, sign$RFSGN)) case "failed" => "failed"
  n * (s::Z) * plusInfinity()

finiteComplexLimit(f, a) ==
  zero?(n := (numer f) a) => 0
  zero?(d := (denom f) a) => infinity()
  (n / d)::OPF

finiteLimit(f, a) ==
  zero?(n := (numer f) a) => 0
  zero?(d := (denom f) a) =>
    (a := sign(n)$RFSGN) case "failed" => "failed"
    rhs1 := fLimit(s::Z, denom f, a, 1)
    lhs1 := fLimit(s::Z, denom f, a, -1)
    rhs1 case "failed" =>
      lhs1 case "failed" => "failed"
      [lhs1, rhs1]
    lhs1 case "failed" => [lhs1, rhs1]
    rhs1::ORF = lhs1::ORF => lhs1::ORF
locallimit(f,x,a) ==
g := univariate(f, x)
zero?(n := whatInfinity a) => finiteLimit(g, retract a)
(dn := degree numer g) > (dd := degree denom g) =>
  (sn := signAround(numer g, n, sign$RFSGN)) case "failed" => "failed"
  (sd := signAround(denom g, n, sign$RFSGN)) case "failed" => "failed"
  (sn::Z) * (sd::Z) * plusInfinity()
(dn < dd => 0
  ((leadingCoefficient numer g) / (leadingCoefficient denom g))::ORF
limit(f,eq,st) ==
(xx := retractIfCan(lhs eq)@Union(SE,"failed")) case "failed" =>
  error "limit: left hand side must be a variable"
x := xx :: SE; a := rhs eq
zero?(n := (numer(g := univariate(f, x))) a) => 0
zero?(d := (denom g) a) =>
  (s := sign(n)$RFSGN) case "failed" => "failed"
  fLimit(s::Z, denom g, a, direction st)
  (n / d)::ORF
locallimitcomplex(f,x,a) ==
g := univariate(f, x)
(r := retractIfCan(a)@Union(RF,"failed")) case RF =>
  finiteComplexLimit(g, r::RF)
  (dn := degree numer g) > (dd := degree denom g) => infinity()
  dn < dd => 0
  ((leadingCoefficient numer g) / (leadingCoefficient denom g))::OPF

package SIGNRF RationalFunctionSign

— RationalFunctionSign.input —


```lisp
(set break resume)
(sys rm -f RationalFunctionSign.output)
(spool RationalFunctionSign.output)
(set message test on)
(set message auto off)
clear all

--S 1 of 1
(>show RationalFunctionSign)
--R
--R RationalFunctionSign(R: GcdDomain) is a package constructor
--R Abbreviation for RationalFunctionSign is SIGNRF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SIGNRF
--R
--R----------------------------------- Operations -----------------------------------
--R sign : Fraction(Polynomial(R)) -> Union(Integer,"failed")
--R sign : (Fraction(Polynomial(R)),Symbol,OrderedCompletion(Fraction(Polynomial(R)))) -> Union(Integer,"failed")
--R sign : (Fraction(Polynomial(R)),Symbol,Fraction(Polynomial(R)),String) -> Union(Integer,"failed")
--R
--E 1

(spool)
lisp (bye)
```

---

--- RationalFunctionSign.help ---

=====================================
RationalFunctionSign examples
=====================================

Find the sign of a rational function around a point or infinity.

See Also:
- )show RationalFunctionSign

---
RationalFunctionSign (SIGNRF)

Exports:

sign

— package SIGNRF RationalFunctionSign —

)abbrev package SIGNRF RationalFunctionSign
++ Author: Manuel Bronstein
++ Date Created: 23 August 1989
++ Date Last Updated: 26 November 1991
++ Description:
++ Find the sign of a rational function around a point or infinity.

RationalFunctionSign(R:GcdDomain): Exports == Implementation where
    SE ==> Symbol
    P ==> Polynomial R
    RF ==> Fraction P
    ORF ==> OrderedCompletion RF
    UP ==> SparseUnivariatePolynomial RF
    U ==> Union(Integer, "failed")
    SGN ==> ToolsForSign(R)

Exports ==> with
    sign: RF -> U
    ++ sign f returns the sign of f if it is constant everywhere.
    sign: (RF, SE, ORF) -> U
    ++ sign(f, x, a) returns the sign of f as x approaches \spad{a},
        ++ from both sides if \spad{a} is finite.
    sign: (RF, SE, RF, String) -> U
    ++ sign(f, x, a, s) returns the sign of f as x nears \spad{a} from
        ++ the left (below) if s is the string \spad{"left"},
        ++ or from the right (above) if s is the string \spad{"right"}.

Implementation ==> add
    import SGN
    import InnerPolySign(RF, UP)
import PolynomialCategoryQuotientFunctions(IndexedExponents SE,
SE, R, P, RF)

psign : P -> U
sqfrSign : P -> U
termSign : P -> U
listSign : (List P, Integer) -> U
finiteSign: (Fraction UP, RF) -> U

sign f ==
  (un := psign numer f) case "failed" => "failed"
  (ud := psign denom f) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

finiteSign(g, a) ==
  (ud := signAround(denom g, a, sign$%)) case "failed" => "failed"
  (un := signAround(numer g, a, sign$%)) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

sign(f, x, a) ==
  g := univariate(f, x)
  zero?(n := whatInfinity a) => finiteSign(g, retract a)
  (ud := signAround(denom g, n, sign$%)) case "failed" => "failed"
  (un := signAround(numer g, n, sign$%)) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

sign(f, x, a, st) ==
  (ud := signAround(denom(g := univariate(f, x)), a,
  d := direction st, sign$%)) case "failed" => "failed"
  (un := signAround(numer g, a, d, sign$%)) case "failed" => "failed"
  (un::Integer) * (ud::Integer)

psign p ==
  (r := retractIfCan(p)@Union(R, "failed")) case R => sign(r::R)$SGN
  (u := sign(retract(unit(s := squareFree p))@R)$SGN) case "failed" => "failed"
  ans := u::Integer
  for term in factors s | odd?(term.exponent) repeat
    (u := sqfrSign(term.factor)) case "failed" => return "failed"
    ans := ans * (u::Integer)
  ans

sqfrSign p ==
  (u := termSign first(l := monomials p)) case "failed" => "failed"
  listSign(rest l, u::Integer)

listSign(l, s) ==
  for term in l repeat
    (u := termSign term) case "failed" => return "failed"
    u::Integer ^= s => return "failed"
s

termSign term ==
  for var in variables term repeat
    odd? degree(term, var) => return "failed"
    sign(leadingCoefficient term)$SGN

---

-- SIGNRF.dotabb --

"SIGNRF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIGNRF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SIGNRF" -> "PFECAT"

---

package SUMRF RationalFunctionSum

--- RationalFunctionSum.input ---

)set break resume
)sys rm -f RationalFunctionSum.output
)spool RationalFunctionSum.output
)set message test on
)set message auto off
)clear all

--S 1 of 14
sum(i::Polynomial(Integer),variable(i=1..n))
  --R
  --R
  --R 2
  --R 6i - 6i + 1
  --R (1) ---------
  --R 12
  --R Type: Fraction(Polynomial(Integer))
  --E 1

--S 2 of 14
sum(i::Fraction(Polynomial(Integer)),i::Symbol)
  --R
  --R
  --R 2
---R \[6i - 6i + 1\]
---R (2) ---------------
---R 12
---R Type: Union(Fraction(Polynomial(Integer)), ...)
---E 3

--S 3 of 14
sum(i, i=1..n)
--R
--R
--R 2
--R n + n
--R (3) ------
--R 2
--R Type: Fraction(Polynomial(Integer))
--E 3

--S 4 of 14
sum(i::Fraction(Polynomial(Integer)), i=1..n)
--R
--R
--R 2
--R n + n
--R (4) ------
--R 2
--R Type: Union(Fraction(Polynomial(Integer)), ...)
--E 4

--S 5 of 14
s := i=1..n
--R
--R
--R (5) i = 1..n
--R Type: SegmentBinding(Polynomial(Integer))
--E 5

--S 6 of 14
hiseg := high(segment(s))
--R
--R
--R (6) n
--R Type: Polynomial(Integer)
--E 6

--S 7 of 14
loseg := low(segment(s))
--R
--R
--R (7) 1
--R Type: Polynomial(Integer)
---E 7

---S 8 of 14
v:=variable s
---R
---R
---R (8) i
---R Type: Symbol
---E 8

---S 9 of 14
p:=i::Polynomial(Integer)
---R
---R
---R (9) i
---R Type: Polynomial(Integer)
---E 9

---S 10 of 14
f:=sum(p,v)
---R
---R
---R 2
---R 6i - 6i + 1
---R (10) ------------
---R 12
---R Type: Fraction(Polynomial(Integer))
---E 10

---S 11 of 14
t1:=eval(f,v,(1+hiseg))
---R
---R
---R 2
---R 6n + 6n + 1
---R (11) ------------
---R 12
---R Type: Fraction(Polynomial(Integer))
---E 11

---S 12 of 14
t2:=eval(f,v,loseg)
---R
---R
---R 1
---R (12) --
---R 12
---R Type: Fraction(Polynomial(Integer))
---E 12
RationalFunctionSum examples
====================================================================
Computes sums of rational functions.
There are 4 different forms of the sum operator in this domain.
They are variations of the types of the input
sum(i::Polynomial(Integer),variable(i=1..n))

\[
\frac{2}{12} \left( \frac{6i - 6i + 1}{12} \right)
\]
We can compute the sum form on part at a time to see what happens. First, we make a SegmentBinding(Polynomial(Integer)): 

\[ s := i = 1..n \]

\[ i = 1..n \]

We pick out the upper and lower bounds of the segment:

\[ \text{hiseg} := \text{high(segment(s))} \]

\[ n \]

\[ \text{loseg} := \text{low(segment(s))} \]

\[ 1 \]

We pick out the variable from the segment:

\[ v := \text{variable s} \]
We create a polynomial we wish to sum:

\[ p := i::\text{Polynomial}(\text{Integer}) \]

And we create the sum for that polynomial:

\[ f := \text{sum}(p, v) \]

\[ \frac{2}{12} \frac{6i - 6i + 1}{12} \]

The new evaluate it at the upper endpoint:

\[ t1 := \text{eval}(f, v, (1 + \text{hiseg})) \]

\[ \frac{2}{12} \frac{6n + 6n + 1}{12} \]

And the lower endpoint:

\[ t2 := \text{eval}(f, v, \text{loseg}) \]

\[ \frac{1}{12} \]

And we take the difference of the endpoints:

\[ t1 - t2 \]

\[ \frac{2}{2} \frac{n + n}{2} \]

See Also:

\[ \text{show RationalFunctionSum} \]
RationalFunctionSum (SUMRF)

Exports:

sum

— package SUMRF RationalFunctionSum —

)abbrev package SUMRF RationalFunctionSum
++ Author: Manuel Bronstein
++ Date Created: ???
++ Date Last Updated: 19 April 1991
++ Description:
++ Computes sums of rational functions;

RationalFunctionSum(R): Exports == Impl where
  R: Join(IntegralDomain, OrderedSet, RetractableTo Integer)
  P ==> Polynomial R
  RF ==> Fraction P
  FE ==> Expression R
  SE ==> Symbol

Exports ==> with

sum: (P, SE) -> RF
++ sum(a(n), n) returns \spad{A} which
++ is the indefinite sum of \spad{a} with respect to
++ upward difference on \spad{n}, i.e. \spad{A(n+1) - A(n) = a(n)}.
++
++X sum(i::Polynomial(Integer),variable(i=1..n))
sum: (RF, SE) -> Union(RF, FE)
++ sum(a(n), n) returns \spad{A} which
++ is the indefinite sum of \spad{a} with respect to
++ upward difference on \spad{n}, i.e. \spad{A(n+1) - A(n) = a(n)}.
++
++\X sum(i::\text{Fraction}(\text{Polynomial}(\text{Integer})),i::\text{Symbol})
sum: (P, SegmentBinding P) -> RF
++ sum(f(n), n = a..b) returns \spad{f(a) + f(a+1) + \ldots f(b)}.
++
++\X sum(i,i=1..n)
sum: (RF, SegmentBinding RF) -> \text{Union}(RF, FE)
++ sum(f(n), n = a..b) returns \spad{f(a) + f(a+1) + \ldots f(b)}.
++
++\X sum(i::\text{Fraction}(\text{Polynomial}(\text{Integer})),i=1..n)

Impl ==> add
import \text{RationalFunction R}
import \text{GosperSummationMethod}($\text{IndexedExponents}\ SE, SE, R, P, RF$)
innersum : (RF, SE) -> \text{Union}(RF, "failed")
innerpolysum: (P, SE) -> RF

sum(f:RF, s:SegmentBinding RF) ==
  (indef := innersum(f, v := \text{variable} s)) case "failed" =>
    summation(f::FE,map((z:RF)::FE +->z::FE,s)
      $\text{SegmentBindingFunctions2}(RF,FE)$)
  eval(indef::RF, v, 1 + hi segment s)
  - eval(indef::RF, v,lo segment s)

sum(an:RF, n:SE) ==
  (u := innersum(an, n)) case "failed" => summation(an::FE, n)
u::RF

sum(p:P, s:SegmentBinding P) ==
  f := sum(p, v := \text{variable} s)
  eval(f, v, (1 + hi segment s)::RF) - eval(f,v,lo(segment s)::RF)

innersum(an, n) ==
  (r := retractIfCan(an)@Union(P, "failed")) case "failed" =>
    an1 := eval(an, n, -1 + n::RF)
    (u := GospersMethod(an/an1, n, new\$SE)) case "failed" =>
      "failed"
    an1 * eval(u::RF, n, -1 + n::RF)
  sum(r::P, n)

sum(p:P, n:SE) ==
  rec := sum(p, n)$\text{InnerPolySum}($\text{IndexedExponents}\ SE, SE, R, P)
  rec.num / (rec.den :: P)
package INTRAT RationalIntegration

— RationalIntegration.input —

)set break resume
)sys rm -f RationalIntegration.output
)spool RationalIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalIntegration
--R
--R RationalIntegration(F: Join(Field,CharacteristicZero,RetractableTo(Integer)),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for RationalIntegration is INTRAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTRAT
--R
--R---------------------------------------------------------- Operations --------------------------------
--R extendedint : (Fraction(UP),Fraction(UP)) -> Union(Record(ratpart: Fraction(UP),coeff: Fraction(UP)),"failed")
--R infieldint : Fraction(UP) -> Union(Fraction(UP),"failed")
--R integrate : Fraction(UP) -> IntegrationResult(Fraction(UP))
--R limitedint : (Fraction(UP),List(Fraction(UP))) -> Union(Record(mainpart: Fraction(UP),limitedlogs: List(Record(coeff: Fraction(UP)),logand: Fraction(UP)))),"failed")

)spool
)lisp (bye)

— RationalIntegration.help —

====================================================================
RationalIntegration examples
Rational function integration. This package provides functions for the base case of the Risch algorithm.

See Also:
o )show RationalIntegration

---

RationalIntegration (INTRAT)

Exports:
extendedint infieldint integrate limitedint

--- package INTRAT RationalIntegration ---

)abbrev package INTRAT RationalIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 24 October 1995
++ Description:
++ Rational function integration
++ This package provides functions for the base case of the Risch algorithm.
-- Used internally by the integration packages

RationalIntegration(F, UP):Exports == Implementation where
  F : Join(Field, CharacteristicZero, RetractableTo Integer)
  UP: UnivariatePolynomialCategory F

  RF ==> Fraction UP
  IR ==> IntegrationResult RF
  LLG ==> List Record(coef:RF, logand:RF)
URF ==> Union(Record(ratpart:RF, coeff:RF), "failed")
U ==> Union(Record(mainpart:RF, limitedlogs:LLG), "failed")

Exports ==> with
  integrate : RF -> IR
    ++ integrate(f) returns g such that \spad{g' = f}.
  infieldint : RF -> Union(RF, "failed")
    ++ infieldint(f) returns g such that \spad{g' = f} or "failed"
    ++ if the integral of f is not a rational function.
  extendedint: (RF, RF) -> URF
    ++ extendedint(f, g) returns fractions \spad{[h, c]} such that
      ++ \spad{c' = 0} and \spad{h' = f - cg},
      ++ if \spad{[h, c]} exist, "failed" otherwise.
  limitedint : (RF, List RF) -> U
    ++ \spad{limitedint(f, [g1,...,gn])} returns
      ++ fractions \spad{[h,[[ci, gi]]]} such that the gi's are among \spad{[g1,...,gn]}, \spad{ci' = 0}, and
      ++ \spad{[h+sum(ci log(gi))]}' = f, if possible, "failed" otherwise.

Implementation ==> add
import TranscendentalIntegration(F, UP)

infieldint f ==
rec := baseRDE(0, f)$TranscendentalRischDE(F, UP)
rec.nosol => "failed"
rec.ans

integrate f ==
rec := monomialIntegrate(f, differentiate)
integrate(rec.polypart)::RF::IR + rec.ir

limitedint(f, lu) ==
quorem := divide(numer f, denom f)
   (u := primlimintfrac(quorem.remainder / (denom f), differentiate,
   lu)) case "failed" => "failed"
   [u.mainpart + integrate(quorem.quotient)::RF, u.limitedlogs]

extendedint(f, g) ==
fqr := divide(numer f, denom f)
gqr := divide(numer g, denom g)
   (i1 := primextintfrac(fqr.remainder / (denom f), differentiate,
   gqr.remainder / (denom g))) case "failed" => "failed"
i2:=integrate({fqr.quotient-retract(i1.coeff)@UP *gqr.quotient)::RF
   [i2 + i1.ratpart, i1.coeff]

—— INTRAT.dotabb ——
package **RINTERP** RationalInterpolation

**Introduction**

This file contains a crude naïve implementation of rational interpolation, where the coefficients of the rational function are in any given field.

**Questions and Outlook**

- Maybe this file should be joined with pinterp.spad, where polynomial Lagrange interpolation is implemented. I have a second version that parallels the structure of pinterp.spad closely.

- There are probably better ways to implement rational interpolation. Maybe [http://www.cs.ucsb.edu/~omer/personal/abstracts/rational.html](http://www.cs.ucsb.edu/~omer/personal/abstracts/rational.html) contains something useful, but I don’t know.

- Comments welcome!

— **RationalInterpolation.input** —

```lisp
)set break resume
)sys rm -f RationalInterpolation.output
)spool RationalInterpolation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalInterpolation
--R
--R RationalInterpolation(xx: Symbol, F: Field) is a package constructor
--R Abbreviation for RationalInterpolation is RINTERP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RINTERP
--R
--R----------------------------- Operations -----------------------------
--R interpolate : (List(F), List(F), NonNegativeInteger, NonNegativeInteger) -> Fraction(Polynomial(F))
--R
```
CHAPTER 19. CHAPTER R

---

RationalInterpolation examples

This package exports rational interpolation algorithms

See Also:
- `)show RationalInterpolation`

---

### RationalInterpolation (RINTERP)

Exports:
- `interpolate`

---

`RATIONALINTERPOLATION(RINTERP)`

`PFECAT`
xx: Symbol
F: Field
Exports == with
    interpolate: (List F, List F, NonNegativeInteger, NonNegativeInteger) -> Fraction Polynomial F

The implementation sets up a system of linear equations and solves it.

— package RINTERP RationalInterpolation —

Implementation == add
    interpolate(xlist, ylist, m, k) ==

First we check whether we have the right number of points and values. Clearly the number of points and the number of values must be identical. Note that we want to determine the numerator and denominator polynomials only up to a factor. Thus, we want to determine \( m + k + 1 \) coefficients, where \( m \) is the degree of the polynomial in the numerator and \( k \) is the degree of the polynomial in the denominator.

In fact, we could also leave, for example, \( k \) unspecified and determine it as \( k = \#\text{xlist} - m - 1 \). I don’t know whether this would be better.

— package RINTERP RationalInterpolation —

\[
\#\text{xlist} ^= \#\text{ylist} =>
\]
\[
\text{error "Different number of points and values."}
\]
\[
\#\text{xlist} ^= m + k + 1 =>
\]
\[
\text{error "Wrong number of points"}
\]

The next step is to set up the matrix. Suppose that our numerator polynomial is \( p(x) = a_0 + a_1 x + \ldots + a_m x^m \) and that our denominator polynomial is \( q(x) = b_0 + b_1 x + \ldots + b_m x^m \). Then we have the following equations, writing \( n \) for \( m + k + 1 \):

\[
p(x_1) - y_1 q(x_1) = a_0 + a_1 x_1 + \ldots + a_m x_1^m - y_1 (b_0 + b_1 x_1 + \ldots + b_m x_1^m) = 0
\]
\[
p(x_2) - y_2 q(x_2) = a_0 + a_1 x_2 + \ldots + a_m x_2^m - y_2 (b_0 + b_1 x_2 + \ldots + b_m x_2^m) = 0
\]
\[
\vdots
\]
\[
p(x_n) - y_n q(x_n) = a_0 + a_1 x_n + \ldots + a_m x_n^m - y_n (b_0 + b_1 x_n + \ldots + b_m x_n^m) = 0
\]
This can be written as

\[
\begin{bmatrix}
1 & x_1 & \cdots & x_1^m & -y_1 & -y_1 x_1 & \cdots & -y_1 x_1^k \\
1 & x_2 & \cdots & x_2^m & -y_2 & -y_2 x_2 & \cdots & -y_2 x_2^k \\
\vdots \\
1 & x_n & \cdots & x_n^m & -y_n & -y_n x_n & \cdots & -y_n x_n^k \\
\end{bmatrix}
\begin{bmatrix}
a_0 \\
a_1 \\
\vdots \\
a_m \\
b_0 \\
b_1 \\
\vdots \\
b_k \\
\end{bmatrix}
= 0
\]

We generate this matrix columnwise:

```
--- package RINTERP RationalInterpolation ---

tempvec: List F := [1 for i in 1..(m+k+1)]
collist: List List F := cons(tempvec,
   [[tempvec := [tempvec.i * xlist.i _
   for i in 1..(m+k+1)] _
   for j in 1..max(m,k)]]
collist := append([collist.j for j in 1..(m+1)], _
   [[- collist.j.i * ylist.i for i in 1..(m+k+1)] _
   for j in 1..(k+1)]
```

Now we can solve the system:

```
--- package RINTERP RationalInterpolation ---

res: List Vector F := nullSpace((transpose matrix collist) _
   ::Matrix F)
```

Note that it may happen that the system has several solutions. In this case, some of the
data points may not be interpolated correctly. However, the solution is often still useful,
thus we do not signal an error.

```
--- package RINTERP RationalInterpolation ---

if #res=1 then output("Warning: unattainable points!" _
   ::OutputForm)$OutputPackage
```

In this situation, all the solutions will be equivalent, thus we can always simply take the first one:
Finally, we generate the rational function:

— package RINTERP RationalInterpolation —

\[
\frac{\text{reduce}(\_+), \text{reslist}.1}{\text{reduce}(\_+), \text{reslist}.2}
\]

— RINTERP.dotabb —

"RINTERP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RINTERP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RINTERP" -> "PFECAT"

package ODERAT RationalLODE

— RationalLODE.input —

)set break resume
)sys rm -f RationalLODE.output
)spool RationalLODE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalLODE
--R
--R RationalLODE(F: Join(Field,CharacteristicZero,RetractableTo(Integer),RetractableTo(Fraction(Integer))),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for RationalLODE is ODERAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODERAT
--R
--R----------------------------------- Operations -----------------------------------
--R indicialEquationAtInfinity : LinearOrdinaryDifferentialOperator1(Fraction(UP)) -> UP
RationalLODE (ODERAT)

Exports:
indicialEquationAtInfinity    ratDsolve

— package ODERAT RationalLODE —
\texttt{P\textsc{ackage}} \texttt{O\textsc{derat R\textsc{ationalLODE}}}

\texttt{\texttt{\textbackslash{abrev}} package \texttt{O\textsc{derat R\textsc{ationalLODE}}}}
++ \texttt{Author: Manuel Bronstein}
++ \texttt{Date Created: 13 March 1991}
++ \texttt{Date Last Updated: 13 April 1994}
++ \texttt{Description:}
++ \texttt{\spad{RationalLODE} provides functions for in-field solutions of linear}
++ \texttt{ordinary differential equations, in the rational case.}

\texttt{RationalLODE(F, UP): Exports == Implementation where}
\hspace{1em} \texttt{F : Join(Field, CharacteristicZero, RetractableTo Integer,}
\hspace{2em} \texttt{RetractableTo Fraction Integer)}
\texttt{UP : UnivariatePolynomialCategory F}

\texttt{N} \hspace{1em} \texttt{\Rightarrow} \hspace{1em} \texttt{NonNegativeInteger}
\texttt{Z} \hspace{1em} \texttt{\Rightarrow} \hspace{1em} \texttt{Integer}
\texttt{RF} \hspace{1em} \texttt{\Rightarrow} \hspace{1em} \texttt{Fraction UP}
\texttt{U} \hspace{1em} \texttt{\Rightarrow} \hspace{1em} \texttt{Union(RF, "failed")}
\texttt{V} \hspace{1em} \texttt{\Rightarrow} \hspace{1em} \texttt{Vector F}
\texttt{M} \hspace{1em} \texttt{\Rightarrow} \hspace{1em} \texttt{Matrix F}
\texttt{LODO \Rightarrow} \texttt{LinearOrdinaryDifferentialOperator1 RF}
\texttt{LODO2\Rightarrow} \texttt{LinearOrdinaryDifferentialOperator2(UP, RF)}

\texttt{Exports} \hspace{1em} \texttt{==} \hspace{1em} \texttt{with}
\hspace{1em} \texttt{ratDsolve: (LODO, RF) \rightarrow \texttt{Record(particular: U, basis: List RF)}}
\hspace{1em} ++ \spad{ratDsolve(op, g)} returns \texttt{\{"failed", []\}} if the equation
\hspace{2em} ++ \spad{op y = g} has no rational solution. Otherwise, it returns
\hspace{3em} ++ \spad{[f, [y_1, ..., y_m]]} where \texttt{f} is a particular rational solution
\hspace{3em} ++ and the \texttt{y_i}'s form a basis for the rational solutions of the
\hspace{3em} ++ homogeneous equation.
\hspace{1em} \texttt{ratDsolve: (LODO, List RF) \rightarrow \texttt{Record(basis:List RF, mat:Matrix F)}}
\hspace{1em} ++ \spad{ratDsolve(op, [g_1, ..., g_m])} returns \texttt{\{[h_1, ..., h_q], M\}} such
\hspace{2em} ++ that any rational solution of \spad{\texttt{op y = c_1 g_1 + ... + c_m g_m}}
\hspace{3em} ++ is of the form \spad{\texttt{d_1 h_1 + ... + d_q h_q}} where
\hspace{3em} ++ \spad{M \{d_1, ..., d_q, c_1, ..., c_m\} = 0}.
\hspace{1em} \texttt{ratDsolve: (LODO2, RF) \rightarrow \texttt{Record(particular: U, basis: List RF)}}
\hspace{1em} ++ \spad{ratDsolve(op, g)} returns \texttt{\{"failed", []\}} if the equation
\hspace{2em} ++ \spad{op y = g} has no rational solution. Otherwise, it returns
\hspace{3em} ++ \spad{[f, [y_1, ..., y_m]]} where \texttt{f} is a particular rational solution
\hspace{3em} ++ and the \texttt{y_i}'s form a basis for the rational solutions of the
\hspace{3em} ++ homogeneous equation.
\hspace{1em} \texttt{ratDsolve: (LODO2, List RF) \rightarrow \texttt{Record(basis:List RF, mat:Matrix F)}}
\hspace{1em} ++ \spad{ratDsolve(op, [g_1, ..., g_m])} returns \texttt{\{[h_1, ..., h_q], M\}} such
\hspace{2em} ++ that any rational solution of \spad{\texttt{op y = c_1 g_1 + ... + c_m g_m}}
\hspace{3em} ++ is of the form \spad{\texttt{d_1 h_1 + ... + d_q h_q}} where
\hspace{3em} ++ \spad{M \{d_1, ..., d_q, c_1, ..., c_m\} = 0}.
\hspace{1em} \texttt{indicialEquationAtInfinity: LODO \rightarrow UP}
\hspace{1em} ++ \texttt{indicialEquationAtInfinity op} returns the indicial equation of
\hspace{2em} ++ \spad{\texttt{op}} at infinity.
\hspace{1em} \texttt{indicialEquationAtInfinity: LODO2 \rightarrow UP}
\hspace{1em} ++ \texttt{indicialEquationAtInfinity op} returns the indicial equation of
++ \spad{op} at infinity.

Implementation ==> add
import BoundIntegerRoots(F, UP)
import RationalIntegration(F, UP)
import PrimitiveRatDE(F, UP, LDO2, LDO)
import LinearSystemMatrixPackage(F, V, V, M)
import InnerCommonDenominator(UP, RF, List UP, List RF)

nzero? : V -> Boolean
evenodd : N -> F
UPfact : N -> UP
infOrder : RF -> Z
infTau : (UP, N) -> F
infBound : (LODO2, List RF) -> N
regularPoint : (LODO2, List RF) -> Z
infIndicialEquation: (List N, List UP) -> UP
makeDot : (Vector F, List RF) -> RF
unitlist : (N, N) -> List F
infMuLambda: LDO2 -> Record(mu:Z, lambda:List N, func:List UP)
ratDsolve0: (LODO2, RF) -> Record(particular: U, basis: List RF)
ratDsolve1: (LODO2, List RF) -> Record(basis:List RF, mat:Matrix F)
candidates: (LODO2,List RF,UP) -> Record(basis:List RF,particular:List RF)

dummy := new()$Symbol

infOrder f == (degree denom f) - (degree numer f)
evenodd n == (even? n => 1; -1)

ratDsolve1(op, lg) ==
d := denomLODE(op, lg)
rec := candidates(op, lg, d)
l := concat([op q for q in rec.basis],
            [op(rec.particular.i) - lg.i for i in 1..#(rec.particular)])
sys1 := reducedSystem(matrix [l])@Matrix(UP)
    [rec.basis, reducedSystem sys1]

ratDsolve0(op, g) ==
zero? degree op => [inv(leadingCoefficient(op)::RF) * g, empty()]
minimumDegree op > 0 =>
sol := ratDsolve0(monicRightDivide(op, monomial(1, 1)).quotient, g)
b:List(RF) := [1]
for f in sol.basis repeat
  if (uu := infieldint f) case RF then b := concat(uu::RF, b)
sol.particular case "failed" => ["failed", b]
  [infieldint(sol.particular::RF), b]
(u := denomLODE(op, g)) case "failed" => ["failed", empty()]
rec := candidates(op, [g], u::UP)
l := 1b := lsol := empty()$List(RF)
for q in rec.basis repeat
if zero?(opq := op q) then lsol := concat(q, lsol)
else (l := concat(opq, l); lb := concat(q, lb))
h:RF := (zero? g => 0; first(rec.particular))
empty? l =>
zero? g => [0, lsol]
[(g = op h => h; "failed"), lsol]
m:M
v:V
if zero? g then
m := reducedSystem(reducedSystem(matrix [l])@Matrix(UP))@M
v := new(ncols m, 0)$V
else
sys1 := reducedSystem(matrix [l], vector [g - op h]
)@Record(mat: Matrix UP, vec: Vector UP)
sys2 := reducedSystem(sys1.mat, sys1.vec)@Record(mat:M, vec:V)
m := sys2.mat
v := sys2.vec
sol := solve(m, v)
part:U :=
zero? g => 0
sol.particular case "failed" => "failed"
makeDot(sol.particular::V, lb) + first(rec.particular)
[part,
concat_!(lsol, [makeDot(v, lb) for v in sol.basis | nzero? v])]
CHAPTER 19. CHAPTER R

solver := UnivariateTaylorSeriesODESolver(F, uts)
dd := UP2UTS(d)$tools
f := LDD2FNU(op)$tools
q := degree op
e := unitlist(1, q)
hom := [UTS2UP(dd * ode(f, unitlist(i, q))$solver, n)$tools /$RF d
     for i in 1..q]$List(RF)
a1 := inv(leadingCoefficient(op)::RF)
part :=
[UTS2UP(dd *
    ode(l1:List(uts))$:uts +->
    RF2UTS(a1 * g)$tools + f l1, e)$solver, n)$tools
    /$RF d for g in lg | g ^= 0]$List(RF)
[hom, part]

nzero? v ==
  for i in minIndex v .. maxIndex v repeat
    not zero? qelt(v, i) => return true
  false

-- returns z(z+1)...(z+(n-1))
UPfact n ==
  zero? n => 1
  z := monomial(1, 1)$UP
  */[z + i::F::UP for i in 0..(n-1)::N]

infMuLambda l ==
lamb:List(N) := [d := degree l]
lf:List(UP) := [a := leadingCoefficient l]
mup := degree(a)::Z - d
while (l := reductum l) ^= 0 repeat
  a := leadingCoefficient l
  if (m := degree(a)::Z - (d := degree l)) > mup then
    mup := m
    lamb := [d]
    lf := [a]
  else if (m = mup) then
    lamb := concat(d, lamb)
    lf := concat(a, lf)
  [mup, lamb, lf]

infIndicialEquation(lambda, lf) ==
ans:UP := 0
for i in lambda for f in lf repeat
  ans := ans + evenodd i * leadingCoefficient f * UPfact i
ans

infBound(l, lg) ==
rec := infMuLambda l
n := min(- degree(l)::Z - 1,
integerBound infIndicialEquation(rec.lambda, rec.func))
while not(empty? lg) and zero? first lg repeat lg := rest lg
    empty? lg => (-n)::N
    m := infOrder first lg
    for g in rest lg repeat
        if not(zero? g) and (mm := infOrder g) < m then m := mm
    (-min(n, rec.mu - degree(leadingCoefficient l)::Z + m))::N

makeDot(v, bas) ==
    ans:RF := 0
    for i in 1.. for b in bas repeat ans := ans + v.i::UP * b
    ans

ratDsolve(op:LODO, g:RF) ==
    rec := splitDenominator(op, [g])
    ratDsolve0(rec.eq, first(rec.rh))

ratDsolve(op:LODO, lg:List RF) ==
    rec := splitDenominator(op, lg)
    ratDsolve1(rec.eq, rec.rh)

ratDsolve(op:LODO2, g:RF) ==
    unit?(c := content op) => ratDsolve0(op, g)
    ratDsolve0((op exquo c)::LODO2, inv(c::RF) * g)

ratDsolve(op:LODO2, lg:List RF) ==
    unit?(c := content op) => ratDsolve1(op, lg)
    ratDsolve1((op exquo c)::LODO2, [inv(c::RF) * g for g in lg])

package RATRET RationalRetractions

— RationalRetractions.input —

)}set break resume
(sys rm -f RationalRetractions.output
(spool RationalRetractions.output
(set message test on
(set message auto off
(clear all

--S 1 of 1
(show RationalRetractions
  --R
  --R RationalRetractions(S: RetractableTo(Fraction(Integer))) is a package constructor
  --R Abbreviation for RationalRetractions is RATRET
  --R This constructor is exposed in this frame.
  --R Issue )edit bookvol10.4.pamphlet to see algebra source code for RATRET
  --R
  --R------------------------------------------- Operations -------------------------------------------
  --R rational : S -> Fraction(Integer)      rational? : S -> Boolean
  --R rationalIfCan : S -> Union(Fraction(Integer),"failed")
  --R
  --E 1

(spool
(lisp (bye)

— RationalRetractions.help —

====================================================================
RationalRetractions examples
====================================================================

Rational number testing and retraction functions.

See Also:
  o )show RationalRetractions

—
RationalRetractions (RATRET)

Exports:
  rational  rational?  rationalIfCan

--- package RATRET RationalRetractions ---

)abbrev package RATRET RationalRetractions
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Rational number testing and retraction functions.

RationalRetractions(S:RetractableTo(Fraction Integer)): with
  rational : S -> Fraction Integer
    ++ rational(x) returns x as a rational number;
    ++ error if x is not a rational number;
  rational? : S -> Boolean
    ++ rational?(x) returns true if x is a rational number,
    ++ false otherwise;
  rationalIfCan: S -> Union(Fraction Integer, "failed")
    ++ rationalIfCan(x) returns x as a rational number,
    ++ "failed" if x is not a rational number;
== add
  rational s   == retract s
  rational? s  == retractIfCan(s) case Fraction(Integer)
  rationalIfCan s == retractIfCan s

--- RATRET.dotabb ---

"RATRET" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RATRET"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
package ODERTRIC RationalRicDE

— RationalRicDE.input —

)set break resume
)sys rm -f RationalRicDE.output
)spool RationalRicDE.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RationalRicDE

--R RationalRicDE(F: Join(Field,CharacteristicZero,RetractableTo(Integer),RetractableTo(Fraction(Integer))),UP: UnivariatePolynomialCategory(F)) is a package constructor

--R Abbreviation for RationalRicDE is ODERTRIC

--R This constructor is not exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODERTRIC

--R

--R------------------------------- Operations --------------------------------

--R polyRicDE : (LinearOrdinaryDifferentialOperator2(UP,Fraction(UP)),(UP -> List(F))) -> List(Record(poly: UP,eq: LinearOrdinaryDifferentialOperator2(UP,Fraction(UP))))

--R ricDsolve : (LinearOrdinaryDifferentialOperator1(Fraction(UP)),(UP -> List(F))) -> List(Fraction(UP)) if F has ACF

--R ricDsolve : (LinearOrdinaryDifferentialOperator1(Fraction(UP)),(UP -> Factored(UP))) -> List(Fraction(UP)) if F has ACF

--R ricDsolve : LinearOrdinaryDifferentialOperator2(UP,Fraction(UP)) -> List(Fraction(UP)) if F has ACF

--R ricDsolve : (LinearOrdinaryDifferentialOperator2(UP,Fraction(UP)),(UP -> Factored(UP))) -> List(Fraction(UP)) if F has ACF

--R singRicDE : (LinearOrdinaryDifferentialOperator2(UP,Fraction(UP)),(UP -> Factored(UP))) -> List(Record(frac: Fraction(UP),eq: LinearOrdinaryDifferentialOperator2(UP,Fraction(UP))))

--R

)spool
)lisp (bye)

— RationalRicDE.help —
RationalRicDE examples

In-field solution of Riccati equations, rational case.

See Also:
o )show RationalRicDE

---

RationalRicDE (ODERTRIC)

Exports:
polyRicDE  ricDsolve  singRicDE

--- package ODERTRIC RationalRicDE ---

)abbrev package ODERTRIC RationalRicDE
++ Author: Manuel Bronstein
++ Date Created: 22 October 1991
++ Date Last Updated: 11 April 1994
++ Description:
++ In-field solution of Riccati equations, rational case.

RationalRicDE(F, UP): Exports == Implementation where
  F : Join(Join(Field, CharacteristicZero, RetractableTo Integer,
             RetractableTo Fraction Integer)
  UP : UnivariatePolynomialCategory F

N  ==> NonNegativeInteger
Z  ==> Integer
SY ==> Symbol
P  ==> Polynomial F
RF  ==> Fraction P
EQ  ==> Equation RF
QF  ==> Fraction UP
UP2  ==> SparseUnivariatePolynomial UP
SUP  ==> SparseUnivariatePolynomial P
REC  ==> Record(poly:SUP, vars:List SY)
SOL  ==> Record(var:List SY, val:List F)
POL  ==> Record(poly:UP, eq:L)
FRC  ==> Record(frac:QF, eq:L)
CNT  ==> Record(constant:F, eq:L)
UTS  ==> UnivariateTaylorSeries(F, dummy, 0)
UPS  ==> SparseUnivariatePolynomial UTS
L  ==> LinearOrdinaryDifferentialOperator2(UP, QF)
LQ  ==> LinearOrdinaryDifferentialOperator1 QF

Exports ==> with
ricDsolve: (LQ, UP -> List F) -> List QF
++ ricDsolve(op, zeros) returns the rational solutions of the associated
++ Riccati equation of \spad{op y = 0}.
++ \spad{zeros} is a zero finder in \spad{UP}.
ricDsolve: (LQ, UP -> List F, UP -> Factored UP) -> List QF
++ ricDsolve(op, zeros, ezfactor) returns the rational
++ solutions of the associated Riccati equation of \spad{op y = 0}.
++ \spad{zeros} is a zero finder in \spad{UP}.
++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
++ not necessarily into irreducibles.
ricDsolve: (L, UP -> List F) -> List QF
++ ricDsolve(op, zeros) returns the rational solutions of the associated
++ Riccati equation of \spad{op y = 0}.
++ \spad{zeros} is a zero finder in \spad{UP}.
ricDsolve: (L, UP -> List F, UP -> Factored UP) -> List QF
++ ricDsolve(op, zeros, ezfactor) returns the rational
++ solutions of the associated Riccati equation of \spad{op y = 0}.
++ \spad{zeros} is a zero finder in \spad{UP}.
++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
++ not necessarily into irreducibles.
singRicDE: (L, UP -> Factored UP) -> List FRC
++ singRicDE(op, ezfactor) returns \spad{[[f1,L1], [f2,L2],...,[fk,Lk]]}
++ such that the singular ++ part of any rational solution of the
++ associated Riccati equation of \spad{op y = 0} must be one of the fi's
++ (up to the constant coefficient), in which case the equation for
++ \spad{z = y e^{-\int ai}} is \spad{Li z = 0}.
++ Argument \spad{ezfactor} is a factorisation in \spad{UP},
++ not necessarily into irreducibles.
polyRicDE: (L, UP -> List F) -> List POL
++ polyRicDE(op, zeros) returns \spad{[[p1,L1], [p2,L2],...,[pk,Lk]]}
++ such that the polynomial part of any rational solution of the
++ associated Riccati equation of \spad{op y = 0} must be one of the pi's
++ (up to the constant coefficient), in which case the equation for
++ \spad{\text{ricDsolve}}: \text{LQ} \rightarrow \text{List QF}
++ \text{ricDsolve}(\text{op}) \text{ returns the rational solutions of the associated}
++ \text{Riccati equation of \spad{\text{op y = 0}}.}
ricDsolve: (\text{LQ}, \text{UP} \rightarrow \text{Factored UP}) \rightarrow \text{List QF}
++ \text{ricDsolve}(\text{op}, \text{ezfactor}) \text{ returns the rational solutions of the}
++ \text{associated Riccati equation of \spad{\text{op y = 0}}.}
++ \text{Argument \spad{\text{ezfactor}}} \text{ is a factorisation in \spad{\text{UP}},}
++ \text{not necessarily into irreducibles.}
ricDsolve: \text{L} \rightarrow \text{List QF}
++ \text{ricDsolve}(\text{op}) \text{ returns the rational solutions of the associated}
++ \text{Riccati equation of \spad{\text{op y = 0}}}.
ricDsolve: (\text{L}, \text{UP} \rightarrow \text{Factored UP}) \rightarrow \text{List QF}
++ \text{ricDsolve}(\text{op}, \text{ezfactor}) \text{ returns the rational solutions of the}
++ \text{associated Riccati equation of \spad{\text{op y = 0}}}.
++ \text{Argument \spad{\text{ezfactor}}} \text{ is a factorisation in \spad{\text{UP}},}
++ \text{not necessarily into irreducibles.}

Implementation ==> add
import \text{RatODETools}(\text{P, SUP})
import \text{RationalLODE}(\text{F, UP})
import \text{NonLinearSolvePackage} \text{F}
import \text{PrimitiveRatDE}(\text{F, UP, L, LQ})
import \text{PrimitiveRatRicDE}(\text{F, UP, L, LQ})

FifCan : RF \rightarrow \text{Union(F, "failed")}
UP2SUP : \text{UP} \rightarrow \text{SUP}
innersol : (\text{List UP, Boolean}) \rightarrow \text{List QF}
mapeval : (\text{SUP, List SY, List F}) \rightarrow \text{UP}
ratsol : \text{List List EQ} \rightarrow \text{List SOL}
ratsln : \text{List EQ} \rightarrow \text{Union(SOL, "failed")}
solveModulo : (\text{UP, UP2}) \rightarrow \text{List UP}
logDerOnly : \text{L} \rightarrow \text{List QF}
nonSingSolve : (\text{N, L, UP} \rightarrow \text{List F}) \rightarrow \text{List QF}
constantRic : (\text{UP, UP} \rightarrow \text{List F}) \rightarrow \text{List F}
nopoly : (\text{N, UP, L, UP} \rightarrow \text{List F}) \rightarrow \text{List QF}
reverseUP : \text{UP} \rightarrow \text{UTS}
reverseUTS : (\text{UTS, N}) \rightarrow \text{UP}
newtonSolution : (\text{L, F, N, UP} \rightarrow \text{List F}) \rightarrow \text{UP}
newtonSolve : (\text{UPS, F, N}) \rightarrow \text{Union(UTS, "failed")}
geneticPolynomial: (\text{SY, Z}) \rightarrow \text{Record(poly:SUP, vars:List SY)}
++ \text{genericPolynomial}(s, n) \text{ returns}
++ \spad{[[[s0 + s1 X +...+ sm X^n],[s0,...,sn]]].}

dummy := \text{new}()$\text{SY}

UP2SUP p == \text{map}(z \rightarrow z::P,p)

$\text{UnivariatePolynomialCategoryFunctions2(F,UP,P,SUP}$
CHAPTER 19. CHAPTER R

logDerOnly l == [differentiate(s) / s for s in ratDsolve(l, 0).basis]
ricDsolve(l:LQ, zeros:UP -> List F) == ricDsolve(l, zeros, squareFree)
ricDsolve(l:L, zeros:UP -> List F) == ricDsolve(l, zeros, squareFree)

singRicDE(l, ezfactor) == singRicDE(l, solveModulo, ezfactor)

ricDsolve(l:LQ, zeros:UP -> List F, ezfactor:UP -> Factored UP) ==
ricDsolve(splitDenominator(l, empty()).eq, zeros, ezfactor)

mapeval(p, ls, lv) ==
map(z +-> ground eval(z, ls, lv), p)

$UnivariatePolynomialCategoryFunctions2(P, SUP, F, UP)

FifCan f ==
((n := retractIfCan(numer f))@Union(F, "failed") case F) and
((d := retractIfCan(denom f))@Union(F, "failed") case F) =>
(n::F) / (d::F)
"failed"

-- returns [0, []] if n < 0
genericPolynomial(s, n) ==
ans:SUP := 0
l:List(SY) := empty()
for i in 0..n repeat
   ans := ans + monomial((sy := new s)::P, i::N)
l := concat(sy, l)
[ans, reverse_! l]

ratsln l ==
ls:List(SY) := empty()
lv:List(F) := empty()
for eq in l repeat
   ((u := FifCan rhs eq) case "failed") or
   ((v := retractIfCan(lhs eq)@Union(SY, "failed")) case "failed") =>
   return "failed"
   lv := concat(u::F, lv)
   ls := concat(v::SY, ls)
[ls, lv]

ratsol l ==
ans:List(SOL) := empty()
for sol in l repeat
   if ((u := ratsln sol) case SOL) then ans := concat(u::SOL, ans)
   ans

-- returns [] if the solutions of l have no polynomial component
polyRicDE(l, zeros) ==
ans:List(POL) := [[0, 1]]
empty?(lc := leadingCoefficientRicDE l) => ans
rec := first lc
for a in zeros(rec.eq) | a ^= 0 repeat
if (p := newtonSolution(l, a, rec.deg, zeros)) ^= 0 then
  ans := concat([p, changeVar(l, p)], ans)
ans

-- reverseUP(a_0 + a_1 x + ... + a_n x^n) = a_n + ... + a_0 x^n
reverseUP p ==
  ans:UTS := 0
  n := degree(p)::Z
  while p ^= 0 repeat
    ans := ans + monomial(leadingCoefficient p, (n - degree p)::N)
    p := reductum p
  ans

-- reverseUTS(a_0 + a_1 x + ..., n) = a_n + ... + a_0 x^n
reverseUTS(s, n) ==
  +/\[monomial(coefficient(s, i), (n - i)::N)$UP for i in 0..n]

-- returns a potential polynomial solution p with leading coefficient a^n
newtonSolution(l, a, n, zeros) ==
  i:N
  m:Z := 0
  aeq:UPS := 0
  op := l
  while op ^= 0 repeat
    mu := degree(op) * n + degree leadingCoefficient op
    op := reductum op
    if mu > m then m := mu
    while l ^= 0 repeat
      c := leadingCoefficient l
      d := degree l
      s:UTS := monomial(1, (m - d * n - degree c)::N)$UTS * reverseUP c
      aeq := aeq + monomial(s, d)
      l := reductum l
    (u := newtonSolve(aeq, a, n)) case UTS => reverseUTS(u::UTS, n)
  -- newton lifting failed, so revert to traditional method
  atn := monomial(a, n)$UP
  neq := changeVar(l, atn)
  sols := [sol.poly for sol in polyRicDE(neq, zeros) | degree(sol.poly) < n]
  empty? sols => atn
  atn + first sols

-- solves the algebraic equation eq for y, returns a solution of degree n with
-- initial term a
-- uses naive newton approximation for now
-- an example where this fails is y^2 + 2 x y + 1 + x^2 = 0
-- which arises from the differential operator D^2 + 2 x D + 1 + x^2
newtonSolve(eq, a, n) ==
  deq := differentiate eq
  sol := a::UTS
  for i in 1..n repeat
(xquo := eq(sol) exquo deq(sol)) case "failed" => return "failed"
sol := truncate(sol - xquo::UTS, i)
sol

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
ricDsolve(l:L, zeros:UP -> List F, ezfactor:UP -> Factored UP) ==
n := degree l
ans:List(QF) := empty()
for rec in singRicDE(l, ezfactor) repeat
  ans := removeDuplicates_! concat_!(ans,
    [rec.frac + f for f in nonSingSolve(n, rec.eq, zeros)])
  #ans = n => return ans
ans

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
nonSingSolve(n, l, zeros) ==
ans:List(QF) := empty()
for rec in polyRicDE(l, zeros) repeat
  ans := removeDuplicates_! concat_!(ans, nopoly(n, rec.poly, rec.eq, zeros))
  #ans = n => return ans
ans

constantRic(p, zeros) ==
zero? degree p => empty()
zeros squareFreePart p

-- there could be the same solutions coming in different ways, so we
-- stop when the number of solutions reaches the order of the equation
nopoly(n, p, l, zeros) ==
ans:List(QF) := empty()
for rec in constantCoefficientRicDE(l, zeros) repeat
  ans := removeDuplicates_! concat_!(ans,
    [(rec.constant::UP + p)::QF + f for f in logDerOnly(rec.eq)])
  #ans = n => return ans
ans

-- returns [p1,...,pn] s.t. h(x,pi(x)) = 0 mod c(x)
solveModulo(c, h) ==
rec := genericPolynomial(dummy, degree(c)::Z - 1)
unk:SUP := 0
while not zero? h repeat
  unk := unk + UP2SUP(leadingCoefficient h) * (rec.poly ** degree h)
  h := reductum h
sol := ratsol solve(coefficients(monicDivide(unk,UP2SUP c).remainder),
  rec.vars)
  [mapeval(rec.poly, s.var, s.val) for s in sol]
if F has AlgebraicallyClosedField then
zro1: UP -> List F
zro : (UP, UP -> Factored UP) -> List F

ricDsolve(l:L) == ricDsolve(l, squareFree)
ricDsolve(l:LQ) == ricDsolve(l, squareFree)

ricDsolve(l:L, ezfactor:UP -> Factored UP) ==
   ricDsolve(l, z -> zro(z, ezfactor), ezfactor)

ricDsolve(l:LQ, ezfactor:UP -> Factored UP) ==
   ricDsolve(l, z -> zro(z, ezfactor), ezfactor)

zro(p, ezfactor) ==
   concat [zro1(r.factor) for r in factors ezfactor p]

zro1 p ==
   [zeroOf(map((z:F):F +-> z, p)
     $UnivariatePolynomialCategoryFunctions2(F, UP, F,
     SparseUnivariatePolynomial F))]

——

— ODERTRIC.dotabb —

"ODERTRIC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODERTRIC"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"ODERTRIC" -> "UTSCAT"

——

dear

package RURPK RationalUnivariateRepresentationPackage

— RationalUnivariateRepresentationPackage.input —

)set break resume
)sys rm -f RationalUnivariateRepresentationPackage.output
)spool RationalUnivariateRepresentationPackage.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
RationalUnivariateRepresentationPackage

---

A package for computing the rational univariate representation of a zero-dimensional algebraic variety given by a regular triangular set. This package is essentially an interface for the InternalRationalUnivariateRepresentationPackage constructor. It is used in the ZeroDimensionalSolvePackage for solving polynomial systems with finitely many solutions.

See Also:
- )show RationalUnivariateRepresentationPackage
RationalUnivariateRepresentationPackage (RURPK)

Exports:
  rur

— package RURPK RationalUnivariateRepresentationPackage —

)abbrev package RURPK RationalUnivariateRepresentationPackage
++ Author: Marc Moreno Maza
++ Date Created: 01/1999
++ Date Last Updated: 23/01/1999
++ Description:
++ A package for computing the rational univariate representation
++ of a zero-dimensional algebraic variety given by a regular
++ triangular set. This package is essentially an interface for the
++ \spadtype{InternalRationalUnivariateRepresentationPackage} constructor.
++ It is used in the \spadtype{ZeroDimensionalSolvePackage}
++ for solving polynomial systems with finitely many solutions.

RationalUnivariateRepresentationPackage(R,ls): Exports == Implementation where
  R : Join(EuclideanDomain,CharacteristicZero)
  ls: List Symbol
  N ==> NonNegativeInteger
  Z ==> Integer
  P ==> Polynomial R
  LP ==> List P
  U ==> SparseUnivariatePolynomial(R)
  RUR ==> Record(complexRoots: U, coordinates: LP)

Exports == with

  rur: (LP,Boolean) -> List RUR
  ++ \spad{rur(lp,univ?)} returns a rational univariate representation
  ++ of \spad{lp}. This assumes that \spad{lp} defines a regular
  ++ triangular \spad{ts} whose associated variety is zero-dimensional
  ++ over \spad{R}. \spad{rur(lp,univ?)} returns a list of items
  ++ \spad{[u,lc]} where \spad{u} is an irreducible univariate polynomial
and each \texttt{\texttt{c}} in \texttt{\texttt{lc}} involves two variables: one from \texttt{\texttt{ls}},

the coordinate of \texttt{\texttt{c}}, and an extra variable which

represents any root of \texttt{\texttt{u}}. Every root of \texttt{\texttt{u}} leads to

two tuple of values for the coordinates of \texttt{\texttt{lc}}. Moreover,

a point \texttt{\texttt{x}} belongs to the variety associated with \texttt{\texttt{lp}} iff

there exists an item \texttt{\texttt{spad}}\{(\texttt{u,lc})\} in \texttt{\texttt{spad}}\{(\texttt{lp,univ?})\} and

a root \texttt{\texttt{r}} of \texttt{\texttt{u}} such that \texttt{\texttt{spad}}\{(\texttt{x})\} is given by the

two tuple of values for the coordinates of \texttt{\texttt{lc}} evaluated at \texttt{\texttt{r}}.

If \texttt{\texttt{spad}}\{(\texttt{univ?})\} is \texttt{\texttt{spad}}\{(\texttt{true})\} then each polynomial \texttt{\texttt{spad}}\{(\texttt{c})\}

will have a constant leading coefficient w.r.t. its coordinate.

See the example which illustrates the \texttt{\texttt{spadtype}}\{ZeroDimensionalSolvePackage\}

package constructor.

\begin{verbatim}
Implementation == add
news: Symbol := new()$Symbol
lv: List Symbol := concat(ls,news)
V ==> OrderedVariableList(lv)
Q ==> NewSparseMultivariatePolynomial(R,V)
E ==> IndexedExponents V
TS ==> SquareFreeRegularTriangularSet(R,E,V,Q)
QWT ==> Record(val: Q, tower: TS)
LQWT ==> Record(val: List Q, tower: TS)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,Q)
normpack ==> NormalizationPackage(R,E,V,Q,TS)
rurpack ==> InternalRationalUnivariateRepresentationPackage(R,E,V,Q,TS)
newv: V := variable(news)::V
newq : Q := newv :: Q
rur(lp: List P, univ?: Boolean, check?: Boolean): List RUR ==
  lp := remove(zero?,lp)
  empty? lp =>
    error "rur$RURPACK: #1 is empty"
  any?(ground?,lp) =>
    error "rur$RURPACK: #1 is not a triangular set"
  ts: TS := [[newq]$Q,List Q]
lq: List Q := []
for p in lp repeat
  rif: Union(Q,"failed") := retractIfCan(p)$Q
  rif case "failed" =>
    error "rur$RURPACK: #1 is not a subset of R[ls]"
  q: Q := rif::Q
  lq := cons(q,lq)
lq := sort(infRittWu?,lq)
toSee: List LQWT := [[lq,ts]$Q]$Q
  toSave: List TS := []
\end{verbatim}
while not empty? toSee repeat
    lqwt := first toSee; toSee := rest toSee
    lq := lqwt.val; ts := lqwt.tower
    empty? lq =>
        -- output(ts::OutputForm)$OutputPackage
        toSave := cons(ts,toSave)
    q := first lq; lq := rest lq
    not (mvar(q) > mvar(ts)) =>
        error "rur$RURPACK: #1 is not a triangular set"
    empty? (rest(ts)::TS) =>
        lfq := irreducibleFactors([q])$polsetpack
        for fq in lfq repeat
            newts := internalAugment(fq,ts)
            newlq := [remainder(q,newts).polnum for q in lq]
            toSee := cons([newlq,newts]$LQWT,toSee)
    lsfqwt: List QWT := squareFreePart(q,ts)
    for qwt in lsfqwt repeat
        q := qwt.val; ts := qwt.tower
        if not ground? init(q) then
            q := normalizedAssociate(q,ts)$normpack
            newts := internalAugment(q,ts)
            newlq := [remainder(q,newts).polnum for q in lq]
            toSee := cons([newlq,newts]$LQWT,toSee)
    toReturn: List RUR := []
    for ts in toSave repeat
        lus := rur(ts,univ?)$rurpack
        check? and (not checkRur(ts,lus)$rurpack) =>
            output("RUR for: ")$OutputPackage
            output(ts::OutputForm)$OutputPackage
            output("Is: ")$OutputPackage
            for us in lus repeat output(us::OutputForm)$OutputPackage
            error "rur$RURPACK: bad result with function rur$IRURPK"
        for us in lus repeat
            g: U := univariate(select(us,newv)::Q)$Q
            lc: LP := [convert(q)@P for q in parts(collectUpper(us,newv))]
            toReturn := cons([g,lc]$RUR, toReturn)
    toReturn

rur(lp: List P, univ?: Boolean): List RUR ==
rur(lp,univ?,false)

rur(lp: List P): List RUR == rur(lp,true)
package POLUTIL RealPolynomialUtilitiesPackage

This file describes the Real Closure 1.0 package which consists of different packages, categories and domains.

The package RealPolynomialUtilitiesPackage which receives a field and a univariate polynomial domain with coefficients in the field. It computes some simple functions such as Strum and Sylvester sequences.

The category RealRootCharacterizationCategory provides abstract functionalities to work with "real roots" of univariate polynomials. These resemble variables with some functionalities needed to compute important operations.

RealClosedField is a category with provides common operations available over real closed fields. These include finding all the roots of univariate polynomial, taking square roots, ...

CAVEATS
Since real algebraic expressions are stored as depending on "real roots" which are managed like variables, there is an ordering on these. This ordering is dynamical in the sense that any new algebraic takes precedence over older ones. In particular every creation function raises a new "real root". This has the effect that when you type something like sqrt(2) + sqrt(2) you have two new variables which happen to be equal. To avoid this name the expression such as in s2 := sqrt(2) ; s2 + s2

Also note that computing times depend strongly on the ordering you implicitly provide. Please provide algebraics in the order which most natural to you.

LIMITATIONS
The file reclos.input show some basic use of the package. This packages uses algorithms which are published in [1] and [2] which are based on field arithmetics, in particular for polynomial gcd related algorithms. This can be quite slow for high degree polynomials and subresultants methods usually work best. Beta versions of the package try to use these techniques in a better way and work significantly faster. These are mostly based on unpublished algorithms and cannot be distributed. Please contact the author if you have a particular problem to solve or want to use these versions.

Be aware that approximations behave as post-processing and that all computations are done exactly. They can thus be quite time consuming when depending on several "real roots".

— RealPolynomialUtilitiesPackage.input —

)set break resume
RealPolynomialUtilitiesPackage examples
====================================================================
RealPolynomialUtilitiesPackage provides common functions used by interval coding.
See Also:
  o )show RealPolynomialUtilitiesPackage
RealPolynomialUtilitiesPackage (POLUTIL)

Exports:
boundOfCauchy  lazyVariations  sturmSequence  sturmVariationsOf
sylvesterSequence

--- package POLUTIL RealPolynomialUtilitiesPackage ---

)abbrev package POLUTIL RealPolynomialUtilitiesPackage
++ Author: Renaud Rioboo
++ Date Created: summer 1992
++ Description:
++ \axiomType{RealPolynomialUtilitiesPackage} provides common functions used
++ by interval coding.

RealPolynomialUtilitiesPackage(TheField,ThePols) : PUB == PRIV where

  TheField : Field
  ThePols : UnivariatePolynomialCategory(TheField)

  Z ==> Integer
  N ==> NonNegativeInteger
  P ==> ThePols

  PUB == with

    sylvesterSequence : (ThePols,ThePols) -> List ThePols
    ++ \axiom{sylvesterSequence(p,q)} is the negated remainder sequence
    ++ of p and q divided by the last computed term
    sturmSequence : ThePols -> List ThePols
    ++ \axiom{sturmSequence(p) = sylvesterSequence(p,p')} if
    if TheField has OrderedRing then
      boundOfCauchy : ThePols -> TheField
      ++ \axiom{boundOfCauchy(p)} bounds the roots of p
      sturmVariationsOf : List TheField -> N
      ++ \axiom{sturmVariationsOf(l)} is the number of sign variations
      ++ in the list of numbers l,
++ note that the first term counts as a sign
lazyVariations : (List(TheField), Z, Z) -> N
++ \texttt{lazyVariations(l,s1,sn)} is the number of sign variations
++ in the list of non null numbers \texttt{[s1::l]@sn},

PRIV == add

sturmSequence(p) ==
    sylvesterSequence(p,differentiate(p))

sylvesterSequence(p1,p2) ==
    res : List(ThePols) := [p1]
    while (p2 ^= 0) repeat
        res := cons(p2 , res)
        (p1 , p2) := (p2 , -(p1 rem p2))
    if degree(p1) > 0
        then
            p1 := unitCanonical(p1)
            res := [ term quo p1 for term in res ]
        reverse! res
    if TheField has OrderedRing
    then
        boundOfCauchy(p) ==
            c :TheField := inv(leadingCoefficient(p))
            l := [ c*term for term in rest(coefficients(p))] 
            null(l) => 1
            1 + ("max" / [ abs(t) for t in l ])

    sturmVariationsOf(l) ==
        -- first 0 counts as a sign
        l1 := first(l)
        -- zeros don't count
        if not(zero?(term)) then l1 := cons(term,l1)
        null(l1) => error "POLUTIL: sturmVariationsOf: Bad sequence"

    sturmVariationsOf(l) ==
        null(l1) => error "POLUTIL: sturmVariationsOf: empty list !"
        l1 := first(l)

ln := first(ll)
ll := reverse(rest(ll))
-- if l1 is not zero then first(l) = first(ll)
-- if l1 is zero then first zero should count as a sign
zero?(ll) => 1 + lazyVariations(rest(ll),sign(first(ll)),sign(ln))
lazyVariations(ll, sign(l1), sign(ln))

lazyVariations(l,sl,sh) ==
zero?(sl) or zero?(sh) => error "POLUTIL: lazyVariations: zero sign!"
null(l) =>
  if sl = sh then 0 else 1
null(rest(l)) =>
  if zero?(first(l))
  then error "POLUTIL: lazyVariations: zero sign!"
  else
    if sl = sh
    then
      if (sl = sign(first(l)))
      then 0
      else 2
      -- in this case we save one test
      else 1
    s := sign(l.2)
lazyVariations([first(l)],sl,s) +
lazyVariations(rest(rest(l)),s,sh)

——

— POLUTIL.dotabb —

"POLUTIL" [color="FF4488",href="bookvol10.4.pdf#nameddest=POLUTIL"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"POLUTIL" -> "PFECAT"

——

package REALSOLV RealSolvePackage

— RealSolvePackage.input —

)set break resume
)sys rm -f RealSolvePackage.output
)spool RealSolvePackage.output
)set message test on
)
)
--S 1 of 14
p := 4*x^3 - 3*x^2 + 2*x - 4
--R
--R
--R 3 2
--R (1) 4x - 3x + 2x - 4
--R
--E 1

--S 2 of 14
ans1 := solve(p,0.01)$REALSOLV
--R
--R
--R (2) [1.11328125]
--R
--E 2

--S 3 of 14
ans2 := solve(p::POLY(FRAC(INT)),0.01)$REALSOLV
--R
--R
--R (3) [1.11328125]
--E 3

--S 4 of 14
R := Integer
--R
--R
--R (4) Integer
--E 4

--S 5 of 14
ls : List Symbol := [x,y,z,t]
--R
--R
--R (5) [x,y,z,t]
--E 5

--S 6 of 14
ls2 : List Symbol := [x,y,z,t,new()$Symbol]
--R
--R
--R (6) [x,y,z,t,%A]
--E
pack := ZDSOLVE(R, ls, ls2)

p1 := x**2*y*z + y*z

p2 := x**2*y**2*z + x + z

p3 := x**2*y**2*z**2 + z + 1

lp := [p1, p2, p3]
This package provides numerical solutions of systems of polynomial equations for use in ACPLT.

\[
p := 4x^3 - 3x^2 + 2x - 4
\]

\[
ans1 := \text{solve}(p,0.01)\text{REALSOLV}
\]

\[
ans2 := \text{solve}(p::\text{POLY(FRAC(INT))},0.01)\text{REALSOLV}
\]

R := Integer
ls : List Symbol := [x,y,z,t]
ls2 : List Symbol := [x,y,z,t,new()$Symbol]
pack := ZDSOLVE(R,ls,ls2)
p1 := x**2*y*z + y*z
p2 := x**2*y**2*z + x + z
p3 := x**2*y**2*z**2 + z + 1
lp := [p1, p2, p3]
ans3 := realSolve(lp,[x,y,z],0.01)

See Also:
  )show RealSolvePackage

---

RealSolvePackage (REALSOLV)

Exports:
  realSolve solve

--- package REALSOLV RealSolvePackage ---

)abbrev package REALSOLV RealSolvePackage
++ Description:
++ This package provides numerical solutions of systems of
++ polynomial equations for use in ACPLUT
RealSolvePackage(): Exports == Implementation where
I ==> Integer
IE ==> IndexedExponents Symbol
L ==> List
NF ==> Float
P ==> Polynomial
RN ==> Fraction Integer
SE ==> Symbol
RFI ==> Fraction Polynomial Integer
LIFT ==> PolynomialCategoryLifting(IE,SE,RN,P,RN,RFI)
SOLV ==> FloatingRealPackage Float

Exports ==> with

solve: (P RN,NF) -> L NF
++ solve(p,eps) finds the real zeroes of a
++ univariate rational polynomial p with precision eps.
++
++ X p := 4*x^3 - 3*x^2 + 2*x - 4
++ X solve(p::POLY(FRAC(INT)),0.01)$REALSOLV

solve: (P I,NF) -> L NF
++ solve(p,eps) finds the real zeroes of a univariate
++ integer polynomial p with precision eps.
++
++ X p := 4*x^3 - 3*x^2 + 2*x - 4
++ X solve(p,0.01)$REALSOLV

realSolve: (L P I,L SE,NF) -> L L NF
++ realSolve(lp,lv,eps) = compute the list of the real
++ solutions of the list lp of polynomials with integer
++ coefficients with respect to the variables in lv,
++ with precision eps.
++
++ X lp := [x**2*y*z + y*z
++ X p2 := x**2*y**2*z + x + z
++ X p3 := x**2*y**2*z**2 + z + 1
++ X lp := [p1, p2, p3]
++ X realSolve(lp,[x,y,z],0.01)

Implementation ==> add

prn2rfi: P RN -> RFI
prn2rfi p ==
map(x+->x::RFI, x+->(numer(x)::RFI)/(denom(x)::RFI), p)$LIFT

pi2rfi: P I -> RFI
pi2rfi p == p :: RFI

solve(p:P RN,eps:NF) == realRoots(prn2rfi p, eps)$SOLV
solve(p:P I, eps:NF) == realRoots(p::RFI, eps)$SOLV

realSolve(lp, lv, eps) ==
realRoots(map(pi2rfi, lp)$ListFunctions2(P I, RFI), lv, eps)$SOLV

package REAL0 RealZeroPackage

-- RealZeroPackage.input --

)set break resume
)sys rm -f RealZeroPackage.output
)spool RealZeroPackage.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show RealZeroPackage

--R
--R RealZeroPackage(Pol: UnivariatePolynomialCategory(Integer)) is a package constructor
--R Abbreviation for RealZeroPackage is REAL0
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for REAL0
--R
--R----------------------------------- Operations -----------------------------------
--R midpoint : Record(left: Fraction(Integer), right: Fraction(Integer)) -> Fraction(Integer)
--R midpoints : List(Record(left: Fraction(Integer), right: Fraction(Integer))) -> List(Fraction(Integer))
--R realZeros : Pol -> List(Record(left: Fraction(Integer), right: Fraction(Integer)))
--R realZeros : (Pol,Record(left: Fraction(Integer), right: Fraction(Integer))) -> List(Record(left: Fraction(Integer), right: Fraction(Integer)))
--R realZeros : (Pol,Fraction(Integer)) -> List(Record(left: Fraction(Integer), right: Fraction(Integer)))
--R refine : (Pol,Record(left: Fraction(Integer), right: Fraction(Integer)), Fraction(Integer)) -> Record(left: Fraction(Integer), right: Fraction(Integer))
RealZeroPackage (REAL0)

Exports:
midpoint midpoints realZeros refine

--- package REAL0 RealZeroPackage ---

)abbrev package REAL0 RealZeroPackage
++ Author: Andy Neff
++ Description:
++ This package provides functions for finding the real zeros
++ of univariate polynomials over the integers to arbitrary user-specified
++ precision. The results are returned as a list of
++ isolating intervals which are expressed as records with
++ "left" and "right" rational number components.

RealZeroPackage(Pol): T == C where
Pol: UnivariatePolynomialCategory Integer
RN ==> Fraction Integer
Interval ==> Record(left : RN, right : RN)
isoList ==> List(Interval)
T == with
  -- next two functions find isolating intervals
  realZeros: (Pol) -> isoList
    ++ realZeros(pol) returns a list of isolating intervals for
    ++ all the real zeros of the univariate polynomial pol.
  realZeros: (Pol, Interval) -> isoList
    ++ realZeros(pol, range) returns a list of isolating intervals
    ++ for all the real zeros of the univariate polynomial pol which
    ++ lie in the interval expressed by the record range.
  -- next two functions return intervals smaller then tolerance
  realZeros: (Pol, RN) -> isoList
    ++ realZeros(pol, eps) returns a list of intervals of length less
    ++ than the rational number eps for all the real roots of the
    ++ polynomial pol.
  realZeros: (Pol, Interval, RN) -> isoList
    ++ realZeros(pol, int, eps) returns a list of intervals of length
    ++ less than the rational number eps for all the real roots of the
    ++ polynomial pol which lie in the interval expressed by the
    ++ record int.
  refine: (Pol, Interval, RN) -> Interval
    ++ refine(pol, int, eps) refines the interval int containing
    ++ exactly one root of the univariate polynomial pol to size less
    ++ than the rational number eps.
  refine: (Pol, Interval, Interval) -> Union(Interval,"failed")
    ++ refine(pol, int, range) takes a univariate polynomial pol and
    ++ and isolating interval int containing exactly one real
    ++ root of pol; the operation returns an isolating interval which
    ++ is contained within range, or "failed" if no such isolating interval exists.
  midpoint: Interval -> RN
    ++ midpoint(int) returns the midpoint of the interval int.
  midpoints: isoList -> List RN
    ++ midpoints(isolist) returns the list of midpoints for the list
    ++ of intervals isolist.
C == add
  --Local Functions
  makeSqfr: Pol -> Pol
  ReZeroSqfr: (Pol) -> isoList
PosZero: (Pol) -> isoList
Zero1: (Pol) -> isoList
transMult: (Integer, Pol) -> Pol
transMultInv: (Integer, Pol) -> Pol
transAdd1: (Pol) -> Pol
invert: (Pol) -> Pol
minus: (Pol) -> Pol
negate: Interval -> Interval
rootBound: (Pol) -> Integer
var: (Pol) -> Integer
	negate(int : Interval):Interval == [−int.right,−int.left]

midpoint(i : Interval):RN == (1/2)*(i.left + i.right)

midpoints(li : isoList) : List RN ==

[ midpoint x for x in li ]

makeSqfr(F : Pol):Pol ==

sqfr := squareFree F
F := */[s.factor for s in factors(sqfr)]

realZeros(F : Pol) ==
ReZeroSqfr makeSqfr F

realZeros(F : Pol, rn : RN) ==
F := makeSqfr F
[refine(F,int,rn) for int in ReZeroSqfr(F)]

realZeros(F : Pol, bounds : Interval) ==
F := makeSqfr F
[rint::Interval for int in ReZeroSqfr(F) |
(rint:=refine(F,int,bounds)) case Interval]

realZeros(F : Pol, bounds : Interval, rn : RN) ==
F := makeSqfr F
[refine(F,int,rn) for int in realZeros(F,bounds)]

ReZeroSqfr(F : Pol) ==
F = 0 => error "ReZeroSqfr: zero polynomial"
L : isoList := []
degree(F) = 0 => L
if (r := minimumDegree(F)) > 0 then
L := [[0,0]$Interval]
tempF := F exquo monomial(1, r)
if not (tempF case "failed") then
F := tempF
J:isoList := [negate int for int in reverse(PosZero(minus(F)))]
K : isoList := PosZero(F)
append(append(J, L), K)
PosZero(F : Pol) == --F is square free, primitive
--and F(0) ^= 0; returns isoList for positive
--roots of F

b : Integer := rootBound(F)
F := transMult(b,F)
L : isoList := Zero1(F)
int : Interval
L := [[b*int.left, b*int.right]$Interval for int in L]

Zero1(F : Pol) == --returns isoList for roots of F in (0,1)
J : isoList
K : isoList
L : isoList
L := []
(v := var(transAdd1(invert(F)))) = 0 => []
v = 1 => L := [[0,1]$Interval]
G : Pol := transMultInv(2, F)
H : Pol := transAdd1(G)
if minimumDegree H > 0 then
-- H has a root at 0 => F has one at 1/2, and G at 1
L := [[1/2,1/2]$Interval]
Q : Pol := monomial(1, 1)
tempH : Union(Pol, "failed") := H exquo Q
if not (tempH case "failed") then H := tempH
Q := Q + monomial(-1, 0)
tempG : Union(Pol, "failed") := G exquo Q
if not (tempG case "failed") then G := tempG
int : Interval
J := [[(int.left+1)* (1/2),(int.right+1) * (1/2)]$Interval
for int in Zero1(H)]
K := [[int.left * (1/2), int.right * (1/2)]$Interval
for int in Zero1(G)]
append(append(J, L), K)

rootBound(F : Pol) == --returns power of 2 that is a bound
--for the positive roots of F
if leadingCoefficient(F) < 0 then F := -F
lcoef := leadingCoefficient(F)
F := reductum(F)
i : Integer := 0
while not (F = 0) repeat
if (an := leadingCoefficient(F)) < 0 then i := i - an
F := reductum(F)
b : Integer := 1
while (b * lcoef) <= i repeat
b := 2 * b
b
transMult(c : Integer, F : Pol) ==
--computes Pol G such that G(x) = F(c*x)
  G : Pol := 0
  while not (F = 0) repeat
    n := degree(F)
    G := G + monomial((c**n) * leadingCoefficient(F), n)
    F := reductum(F)
  G

transMultInv(c : Integer, F : Pol) ==
--computes Pol G such that G(x) = (c**n) * F(x/c)
  d := degree(F)
  cc : Integer := 1
  G : Pol := monomial(leadingCoefficient F,d)
  while (F:=reductum(F)) ^= 0 repeat
    n := degree(F)
    cc := cc*(c**(d-n):NonNegativeInteger)
    G := G + monomial(cc * leadingCoefficient(F), n)
  d := n
  G

  otransAdd1(F : Pol) ==
  --computes Pol G such that G(x) = F(x+1)
  -- G : Pol := F
  -- n : Integer := 1
  -- while (F := differentiate(F)) ^= 0 repeat
  --   if not ((tempF := F exquo n) case "failed") then F := tempF
  --   G := G + F
  --   n := n + 1
  -- G

transAdd1(F : Pol) ==
--computes Pol G such that G(x) = F(x+1)
  n := degree F
  v := vectorise(F, n+1)
  for i in 0..(n-1) repeat
    for j in (n-i)..n repeat
      qsetelt_!(v,j, qelt(v,j) + qelt(v,(j+1)))
  ans : Pol := 0
  for i in 0..n repeat
    ans := ans + monomial(qelt(v,(i+1)),i)
  ans

minus(F : Pol) ==
--computes Pol G such that G(x) = F(-x)
  G : Pol := 0
  while not (F = 0) repeat
    n := degree(F)
    coef := leadingCoefficient(F)

odd? n =>
   G := G + monomial(-coef, n)
   F := reductum(F)
G := G + monomial(coef, n)
F := reductum(F)
G

invert(F : Pol) ==
-- computes Pol G such that G(x) = (x**n) * F(1/x)
G := 0
n := degree(F)
while not (F = 0) repeat
   G := G + monomial(leadingCoefficient(F),
   (n-degree(F))::NonNegativeInteger)
   F := reductum(F)
G

var(F : Pol) == -- number of sign variations in coefs of F
i := 0
LastCoef := leadingCoefficient(F) < 0
while not ((F := reductum(F)) = 0) repeat
   next := leadingCoefficient(F) < 0
   if ((not LastCoef) and next) or
      ((not next) and LastCoef) then i := i + 1
   LastCoef := next
i

refine(F : Pol, int : Interval, bounds : Interval) ==
lseg := min(int.right, bounds.right) - max(int.left, bounds.left)
lseg < 0 => "failed"
lseg = 0 =>
   pt :=
      int.left = bounds.right => int.left
      int.right
   elt(transMultInv(denom(pt),F),numer pt) = 0 => [pt,pt]
   "failed"
lseg = int.right - int.left => int
   refine(F, refine(F, int, lseg), bounds)

refine(F : Pol, int : Interval, eps : RN) ==
a := int.left
b := int.right
a=b => [a,b]$Interval
an : Integer := numer(a)
ad : Integer := denom(a)
b_n : Integer := numer(b)
b_d : Integer := denom(b)
xfl : Boolean := false
if (u:=elt(transMultInv(ad, F), an)) = 0 then
   F := (F exquo (monomial(ad,1)-monomial(an,0)))::Pol
   u:=elt(transMultInv(ad, F), an)
if (v:=elt(transMultInv(bd, F), bn)) = 0 then
   F := (F exquo (monomial(bd,1)-monomial(bn,0)))::Pol
   v:=elt(transMultInv(bd, F), bn)
   u:=elt(transMultInv(ad, F), an)
if u > 0 then (F:=-F;v:=-v)
if v < 0 then
   error [int, "is not a valid isolation interval for", F]
if eps <= 0 then error "precision must be positive"
while (b - a) >= eps repeat
   mid : RN := (b + a) * (1/2)
   midn : Integer := numer(mid)
   midd : Integer := denom(mid)
   (v := elt(transMultInv(midd, F), midn)) < 0 =>
      a := mid
      an := midn
      ad := midd
   v > 0 =>
      b := mid
      bn := midn
      bd := midd
   v = 0 =>
      a := mid
      b := mid
      an := midn
      ad := midd
      xfl := true
[a, b]$Interval

——

— REAL0.dotabb —

"REAL0" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REAL0"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"REAL0" -> "PFECAT"

——

package REAL0Q RealZeroPackageQ

— RealZeroPackageQ.input —
RealZeroPackageQ examples

This package provides functions for finding the real zeros of univariate polynomials over the rational numbers to arbitrary user-specified precision. The results are returned as a list of isolating intervals, expressed as records with "left" and "right" rational number components.

See Also:
- )show RealZeroPackageQ
RealZeroPackageQ (REAL0Q)

Exports:
realZeros refine

package REAL0Q RealZeroPackageQ

\)

\)

++ Author: Andy Neff, Barry Trager
++ Date Last Updated: 7 April 1991
++ Description:
++ This package provides functions for finding the real zeros of univariate
++ polynomials over the rational numbers to arbitrary user-specified
++ precision. The results are returned as a list of isolating intervals,
++ expressed as records with "left" and "right" rational number components.

RealZeroPackageQ(Pol): T == C where
  RN ==> Fraction Integer
  I ==> Integer
  SUP ==> SparseUnivariatePolynomial
  Pol: UnivariatePolynomialCategory RN
  Interval ==> Record(left : RN, right : RN)
  isoList ==> List(Interval)
  ApproxInfo ==> Record(approx : RN, exFlag : Boolean)

T == with
  -- next two functions find isolating intervals
  realZeros: (Pol) -> isoList
    ++ realZeros(pol) returns a list of isolating intervals for
    ++ all the real zeros of the univariate polynomial pol.
  realZeros: (Pol, Interval) -> isoList
    ++ realZeros(pol, range) returns a list of isolating intervals
    ++ for all the real zeros of the univariate polynomial pol which
    ++ lie in the interval expressed by the record range.
  -- next two functions return intervals smaller then tolerance
  realZeros: (Pol, RN) -> isoList
    ++ realZeros(pol, eps) returns a list of intervals of length less
    ++ than the rational number eps for all the real roots of the
CHAPTER 19. CHAPTER R

++ polynomial pol.
realZeros: (Pol, Interval, RN) -> isoList
  ++ realZeros(pol, int, eps) returns a list of intervals of length
  ++ less than the rational number eps for all the real roots of the
  ++ polynomial pol which lie in the interval expressed by the
  ++ record int.
refine: (Pol, Interval, RN) -> Interval
  ++ refine(pol, int, eps) refines the interval int containing
  ++ exactly one root of the univariate polynomial pol to size less
  ++ than the rational number eps.
refine: (Pol, Interval, Interval) -> Union(Interval,"failed")
  ++ refine(pol, int, range) takes a univariate polynomial pol and
  ++ and isolating interval int which must contain exactly one real
  ++ root of pol, and returns an isolating interval which
  ++ is contained within range, or "failed" if no such isolating interval exists.
C == add
  import RealZeroPackage SparseUnivariatePolynomial Integer
  convert2PolInt: Pol -> SparseUnivariatePolynomial Integer
  convert2PolInt(f : Pol) ==
    pden:I :=lcm([denom c for c in coefficients f])
    map(numer,pden * f)$UnivariatePolynomialCategoryFunctions2(RN,Pol,I,SUP I)
realZeros(f : Pol) == realZeros(convert2PolInt f)
realZeros(f : Pol, rn : RN) == realZeros(convert2PolInt f, rn)
realZeros(f : Pol, bounds : Interval) ==
  realZeros(convert2PolInt f, bounds)
realZeros(f : Pol, bounds : Interval, rn : RN) ==
  realZeros(convert2PolInt f, bounds, rn)
refine(f : Pol, int : Interval, eps : RN) ==
  refine(convert2PolInt f, int, eps)
refine(f : Pol, int : Interval, bounds : Interval) ==
  refine(convert2PolInt f, int, bounds)

—— REAL0Q.dotabb ——

"REAL0Q" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REAL0Q"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"REAL0Q" -> "PFECAT"

——
package RMCAT2 RectangularMatrixCategoryFunctions2

--- RectangularMatrixCategoryFunctions2.input ---

)set break resume
/sys rm -f RectangularMatrixCategoryFunctions2.output
/spool RectangularMatrixCategoryFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RectangularMatrixCategoryFunctions2
--R
--R RectangularMatrixCategoryFunctions2(m: NonNegativeInteger,n: NonNegativeInteger,R1: Ring,Row1: DirectProductCategory(m,R2),M2: RectangularMatrixCategory(m,n,R2,Row2,Col2)) is a package constructor
--R Abbreviation for RectangularMatrixCategoryFunctions2 is RMCAT2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RMCAT2
--R
--R------------------------------- Operations -------------------------------
--R map : ((R1 -> R2),M1) -> M2
--R reduce : (((R1,R2) -> R2),M1,R2) -> R2
--R
--E 1

)spool
)lisp (bye)

---

--- RectangularMatrixCategoryFunctions2.help ---

====================================================================
RectangularMatrixCategoryFunctions2 examples
====================================================================

RectangularMatrixCategoryFunctions2 provides functions between two matrix domains. The functions provided are map and reduce.

See Also:
  o )show RectangularMatrixCategoryFunctions2

---
RectangularMatrixCategoryFunctions2 (RMCAT2)

Exports:
  map  reduce

— package RMCAT2 RectangularMatrixCategoryFunctions2 —

)abbrev package RMCAT2 RectangularMatrixCategoryFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 21 November 1989
++ Date Last Updated: 12 June 1991
++ Description:
++ \spadtype{RectangularMatrixCategoryFunctions2} provides functions between
++ two matrix domains. The functions provided are \spadfun{map} and
++ \spadfun{reduce}.

RectangularMatrixCategoryFunctions2(m,n,R1,Row1,Col1,M1,R2,Row2,Col2,M2):_
  Exports == Implementation where
  m,n : NonNegativeInteger
  R1 : Ring
  Row1 : DirectProductCategory(n, R1)
  Col1 : DirectProductCategory(m, R1)
  M1 : RectangularMatrixCategory(m,n,R1,Row1,Col1)
  R2 : Ring
  Row2 : DirectProductCategory(n, R2)
  Col2 : DirectProductCategory(m, R2)
  M2 : RectangularMatrixCategory(m,n,R2,Row2,Col2)

Exports == with
  map: (R1 -> R2,M1) -> M2
  ++ \spad{map(f,m)} applies the function \spad{f} to the elements of the
  ++ matrix \spad{m}.
  reduce: ((R1,R2) -> R2,M1,R2) -> R2
  ++ \spad{reduce(f,m,r)} returns a matrix \spad{n} where
  ++ \spad{n[i,j] = f(m[i,j],r)} for all indices \spad{i} and \spad{j}.

Implementation == add
minr ==> minRowIndex
maxr ==> maxRowIndex
minc ==> minColIndex
maxc ==> maxColIndex

map(f,mat) ==
  ans : M2 := new(m,n,0)$Matrix(R2) pretend M2
  for i in minr(mat)..maxr(mat) for k in minr(ans)..maxr(ans) repeat
    for j in minc(mat)..maxc(mat) for l in minc(ans)..maxc(ans) repeat
      qsetelt_!(ans pretend Matrix R2,k,l,f qelt(mat,i,j))
  ans

reduce(f,mat,ident) ==
  s := ident
  for i in minr(mat)..maxr(mat) repeat
    for j in minc(mat)..maxc(mat) repeat
      s := f(qelt(mat,i,j),s)
  s

;;

-- RMCAT2.dotabb --

"RMCAT2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RMCAT2"]
"RMATCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RMATCAT"]
"RMCAT2" -> "RMATCAT"

;;

package RECOP RecurrenceOperator

-- RecurrenceOperator.input --

)set break resume
)sys rm -f RecurrenceOperator.output
)spool RecurrenceOperator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RecurrenceOperator
--R
--R RecurrenceOperator(R: Join(OrderedSet,IntegralDomain,ConvertibleTo(InputForm)),F: Join(FunctionSpace(R),AbelianMonoid,RetractableTo(Integer),RetractableTo(Symbol),PartialDifferentialRing(Symbol),CombinatorialOpsCategory)) is a package constructor
--R Abbreviation for RecurrenceOperator is RECOP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RECOP
--R
--R------------------------------- Operations -------------------------------
--R getEq : F -> F
     getOp : F -> BasicOperator
--R evalADE : (BasicOperator,Symbol,F,F,F,List(F)) -> F
--R evalRec : (BasicOperator,Symbol,F,F,F,List(F)) -> F
--R getShiftRec : (BasicOperator,Kernel(F),Symbol) -> Union(Integer,"failed") if R has RING
--R numberOfValuesNeeded : (Integer,BasicOperator,Symbol,F) -> Integer
--R shiftInfoRec : (BasicOperator,Symbol,F) -> Record(max: Union(Integer,"failed"),ord: Union(Integer,"failed"),ker: Kernel(F)) if R has RING
--R

)spool
)lisp (bye)

---------------------------------
--- RecurrenceOperator.help ---

This package provides an operator for the n-th term of a recurrence and an operator for the coefficient of x^n in a function specified by a functional equation.

See Also:
  o )show RecurrenceOperator

---------
RecurrenceOperator (RECOP)

Exports:
getEq getOp evalADE evalRec
getShiftRec numberOfValuesNeeded shiftInfoRec

— package RECOP RecurrenceOperator —

)abbrev package RECOP RecurrenceOperator
++ Author: Martin Rubey
++ Description:
++ This package provides an operator for the n-th term of a recurrence and an
++ operator for the coefficient of x^n in a function specified by a functional
++ equation.

RecurrenceOperator(R, F): Exports == Implementation where
R: Join(OrderedSet, IntegralDomain, ConvertibleTo InputForm)
F: Join(FunctionSpace R, AbelianMonoid, RetractableTo Integer, _
RetractableTo Symbol, PartialDifferentialRing Symbol, _
CombinatorialOpsCategory)
--RecurrenceOperator(F): Exports == Implementation where
-- F: Join(ExpressionSpace, AbelianMonoid, RetractableTo Integer, _
-- RetractableTo Symbol, PartialDifferentialRing Symbol)

Exports == with

evalRec: (BasicOperator, Symbol, F, F, F, List F) -> F
++ \spad{evalRec(u, dummy, n, n0, eq, values)} creates an expression that
++ stands for u(n0), where u(n) is given by the equation eq. However, for
++ technical reasons the variable n has to be replaced by a dummy
++ variable dummy in eq. The argument values specifies the initial values
++ of the recurrence u(0), u(1),...
++ For the moment we don't allow recursions that contain u inside of
++ another operator.

evalADE: (BasicOperator, Symbol, F, F, F, List F) -> F
++ \spad{evalADE(f, dummy, x, n, eq, values)} creates an expression that
 CHAPTER 19. CHAPTER R

 gets for the coefficient of x^n in the Taylor expansion of f(x),
 where f(x) is given by the functional equation eq. However, for
 technical reasons the variable x has to be replaced by a dummy
 variable dummy in eq. The argument values specifies the first few
 Taylor coefficients.

 \spad{getEq f} returns the defining equation, if f represents the
 coefficient of an ADE or a recurrence.

 \spad{getOp f}, if f represents the coefficient of a recurrence or
 ADE, returns the operator representing the solution

 -- should be local
 numberOfValuesNeeded: (Integer, BasicOperator, Symbol, F) -> Integer

 -- should be local
 if R has Ring then
 getShiftRec: (BasicOperator, Kernel F, Symbol) -> Union(Integer, "failed")

 shiftInfoRec: (BasicOperator, Symbol, F) ->
 Record(max: Union(Integer, "failed"),
 ord: Union(Integer, "failed"),
 ker: Kernel F)

 Implementation == add
 \getchunk{implementation: RecurrenceOperator}

 Defining new operators

 We define two new operators, one for recurrences, the other for functional equations. The
 operators for recurrences represents the n^th term of the corresponding sequence, the other
 the coefficient of x^n in the Taylor series expansion.

 — implementation: RecurrenceOperator —

 oprecur := operator("rootOfRec":::Symbol)$BasicOperator
 opADE := operator("rootOfADE":::Symbol)$BasicOperator

 setProperty(oprecur, "%dummyVar", 2 pretend None)
 setProperty(opADE, "%dummyVar", 2 pretend None)
Setting these properties implies that the second and third arguments of oprecur are dummy variables and affects `tower$ES`: the second argument will not appear in `tower$ES`, if it does not appear in any argument but the first and second. The third argument will not appear in `tower$ES`, if it does not appear in any other argument. (%defsum is a good example)

The arguments of the two operators are as follows:

1. eq, i.e. the vanishing expression

    — implementation: RecurrenceOperator —

        `eqAsF`: `List F -> F`
        `eqAsF 1 == 1.1`

          —

2. dummy, a dummy variable to make substitutions possible

    — implementation: RecurrenceOperator —

        `dummy`: `List F -> Symbol`
        `dummy 1 == retract(1.2)@Symbol`

        `dummyAsF`: `List F -> F`
        `dummyAsF 1 == 1.2`

          —

3. the variable for display

    — implementation: RecurrenceOperator —

        `displayVariable`: `List F -> F`
        `displayVariable 1 == 1.3`

          —

4. `operatorName(argument)`

    — implementation: RecurrenceOperator —

        `operatorName`: `List F -> BasicOperator`
        `operatorName 1 == operator(kernels(1.4).1)`

        `operatorNameAsF`: `List F -> F`
        `operatorNameAsF 1 == 1.4`

        `operatorArgument`: `List F -> F`
        `operatorArgument 1 == argument(kernels(1.4).1).1`
Concerning rootOfADE, note that although we have arg as argument of the operator, it is intended to indicate the coefficient, not the argument of the power series.

5. values in reversed order.

- rootOfRec: maybe values should be preceded by the index of the first given value. Currently, the last value is interpreted as \( f(0) \).
- rootOfADE: values are the first few coefficients of the power series expansion in order.

| implementation: RecurrenceOperator |

| initialValues: List F -> List F |
| initialValues 1 == rest(1, 4) |

Recurrences

Extracting some information from the recurrence

We need to find out whether we can determine the next term of the sequence, and how many initial values are necessary.

| implementation: RecurrenceOperator |

```lisp
if R has Ring then
  getShiftRec(op: BasicOperator, f: Kernel F, n: Symbol) : Union(Integer, "failed") ==
    a := argument f
    if every?(z +-> freeOf?(z, n::F), a) then return 0
    if #a ~= 1 then error "RECOP: operator should have only one argument"
    p := univariate(a.1, retract(n::F)@Kernel(F))
    if denominator p ~= 1 then return "failed"
    num := numerator p
    if degree num = 1 and coefficient(num, 1) = 1
      and every?(z +-> freeOf?(z, n::F), coefficients num)
      then return retractIfCan(coefficient(num, 0))
      else return "failed"
```
-- if the recurrence is of the form
-- $p(n, f(n+m-o), f(n+m-o+1), \ldots, f(n+m)) = 0$
-- in which case shiftInfoRec returns $[m, o, f(n+m)]$.

shiftInfoRec(op: BasicOperator, argsym: Symbol, eq: F):
  Record(max: Union(Integer, "failed"),
         ord: Union(Integer, "failed"),
         ker: Kernel F) ==

  -- ord and ker are valid only if all shifts are Integers
  -- ker is the kernel of the maximal shift.
  maxShift: Integer
  minShift: Integer
  nextKernel: Kernel F

  -- We consider only those kernels that have op as operator. If there is none,
  -- we raise an error. For the moment we don't allow recursions that contain op
  -- inside of another operator.

  error? := true

  for f in kernels eq repeat
    if is?(f, op) then
      shift := getShiftRec(op, f, argsym)
      if error? then
        error? := false
        nextKernel := f
      if shift case Integer then
        maxShift := shift
        minShift := shift
      else return ["failed", "failed", nextKernel]
    else
      if shift case Integer then
        if maxShift < shift then
          maxShift := shift
          nextKernel := f
        if minShift > shift then
          minShift := shift
      else return ["failed", "failed", nextKernel]
    if error? then error "evalRec: equation does not contain operator"

  [maxShift, maxShift - minShift, nextKernel]
Evaluating a recurrence

--- implementation: RecurrenceOperator ---

evalRec(op, argsym, argdisp, arg, eq, values) ==
if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed")
or (n < 0)
then
  shiftInfo := shiftInfoRec(op, argsym, eq)
  if (shiftInfo.ord case "failed") or ((shiftInfo.ord)::Integer > 0)
  then
    kernel(oprecur,
    append([eq, argsym::F, argdisp, op(arg)], values))
  else
    p := univariate(eq, shiftInfo.ker)
    num := numer p

-- If the degree is 1, we can return the function explicitly.

    if degree num = 1 then
      eval(-coefficient(num, 0)/coefficient(num, 1), argsym::F,
      arg::F-(shiftInfo.max)::Integer::F)
    else
      kernel(oprecur,
      append([eq, argsym::F, argdisp, op(arg)], values))
    else
      len: Integer := #values
      if n < len
      then values.(len-n)
      else
        shiftInfo := shiftInfoRec(op, argsym, eq)

      if shiftInfo.max case Integer then
        p := univariate(eq, shiftInfo.ker)
        num := numer p

      if degree num = 1 then

        next := -coefficient(num, 0)/coefficient(num, 1)
        nextval := eval(next, argsym::F,
        (len-(shiftInfo.max)::Integer)::F)
        newval := eval(nextval, op,
        z->evalRec(op, argsym, argdisp, z, eq, values))
        evalRec(op, argsym, argdisp, arg, eq, cons(newval, values))
      else
        kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))
else
    kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))

numberOfValuesNeeded(numberOfValues: Integer,
    op: BasicOperator,
    argsym: Symbol, eq: F): Integer ==
order := shiftInfoRec(op, argsym, eq).ord
if order case Integer
    then min(numberOfValues, retract(order)@Integer)
else numberOfValues

else
    evalRec(op, argsym, argdisp, arg, eq, values) ==
    kernel(oprecur,
        append([eq, argsym::F, argdisp, op(arg)], values))

numberOfValuesNeeded(numberOfValues: Integer,
    op: BasicOperator, argsym: Symbol, eq: F): Integer ==
    numberOfValues

Setting the evaluation property of oprecur

irecur is just a wrapper that allows us to write a recurrence relation as an operator.

— implementation: RecurrenceOperator —

irecur: List F -> F
irecur l ==
evalRec(operatorName l,
    dummy l, displayVariable l,
    operatorArgument l, eqAsF l, initialValues l)
evaluate(oprecur, irecur)$BasicOperatorFunctions1(F)

Displaying a recurrence relation

— implementation: RecurrenceOperator —
Functional Equations

Determining the number of initial values for ADE's

We use Joris van der Hoeven’s instructions for ADE's. Given \( Q = p(f, f', \ldots, f^{(r)}) \) we first need to differentiate \( Q \) with respect to \( f^{(i)} \) for \( i \in \{0, 1, \ldots, r\} \), plug in the given truncated power series solution and determine the valuation.

— NOTYET implementation: RecurrenceOperator —

```
getValuation(op, argsym, eq, maxorder, values): Integer ==
max: Integer := -1;
ker: Kernel F
for i in 0..maxorder repeat
  ker := D(op(argsym), argsym, i)::Kernel F
  pol := univariate(eq, ker)
  dif := D pol
  ground numer(dif.D(op(argsym), argsym, i))
```
Extracting some information from the functional equation

getOrder returns the maximum derivative of op occurring in f.

--- implementation: RecurrenceOperator ---

```pascal
getOrder(op: BasicOperator, f: Kernel F): NonNegativeInteger ==
res: NonNegativeInteger := 0
  g := f
while is?(g, %diff) repeat
  g := kernels(argument(g).1).1
  res := res+1
if is?(g, op) then res else 0
```

---

Extracting a coefficient given a functional equation

--- implementation: RecurrenceOperator ---

evalADE(op, argsym, argdisp, arg, eq, values) ==
if not freeOf?(eq, retract(argdisp)@Symbol) then error "RECOP: The argument should not be used in the equation of the ADE"

if ((n := retractIfCan(arg)@Union(Integer, "failed")) case "failed") then
  -- try to determine the necessary number of initial values
  keq := kernels eq
  order := getOrder(op, keq.1)
  for k in rest keq repeat order := max(order, getOrder(op, k))

  p: Fraction SparseUnivariatePolynomial F := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F
  if one? degree numer p
    -- the equation is holonomic
    then kernel(opADE, [eq, argsym::F, argdisp, op(arg)], reverse first(reverse values, order))
    else kernel(opADE, [eq, argsym::F, argdisp, op(arg)], values))
  else
    if n < 0 then 0
else
  keq := kernels eq
  order := getOrder(op, keq.1)
  -- output(hconcat("The order is ", order::OutputForm))$OutputPackage
  for k in rest keq repeat order := max(order, getOrder(op, k))

  p: Fraction SparseUnivariatePolynomial F
  := univariate(eq, kernels(D(op(argsym::F), argsym, order)).1)$F
  -- output(hconcat("p: ", p::OutputForm))$OutputPackage

  if degree numer p > 1
    then
      -- kernel(opADE,
      --   append([eq, argsym::F, argdisp, op(arg)], values))

      s := seriesSolve(eq, op, argsym, reverse values)
      $ExpressionSolve(R, F,
      UnivariateFormalPowerSeries F,
      UnivariateFormalPowerSeries
      SparseUnivariatePolynomialExpressions F)
      elt(s, n::Integer::NonNegativeInteger)
    else
      s := seriesSolve(eq, op, argsym, first(reverse values, order))
      $ExpressionSolve(R, F,
      UnivariateFormalPowerSeries F,
      UnivariateFormalPowerSeries
      SparseUnivariatePolynomialExpressions F)
      elt(s, n::Integer::NonNegativeInteger)

iADE: List F -> F
-- This is just a wrapper that allows us to write a recurrence relation as an
-- operator.
  iADE l ==
    evalADE(operatorName l,
    dummy l, displayVariable l,
    operatorArgument l, eqAsF l, initialValues l)

evaluate(opADE, iADE)$BasicOperatorFunctions1(F)

getEq(f: F): F ==
  ker := kernels f
  if one?(#ker) and _
    (is?(operator(ker.1), "rootOfADE"::Symbol) or _
     is?(operator(ker.1), "rootOfRec"::Symbol)) then
    l := argument(ker.1)
eval(eqAsF l, dummyAsF l, displayVariable l)
else
  error "getEq: argument should be a single rootOfADE or rootOfRec object"

getOp(f: F): BasicOperator ==
ker := kernels f
if one?(#ker) and _
  (is?(operator(ker.1), "rootOfADE"::Symbol) or _
    is?(operator(ker.1), "rootOfRec"::Symbol)) then
  operatorName argument(ker.1)
else
  error "getOp: argument should be a single rootOfADE or rootOfRec object"

——

Displaying a functional equation

— implementation: RecurrenceOperator —

ddADE: List F -> OutputForm
ddADE l ==
  op := operatorName l
  values := reverse l
vals: List OutputForm
  := cons(eval(eqAsF l, dummyAsF l, displayVariable l)::OutputForm = _
    0::OutputForm,
    [eval(D(op(dummyAsF l), dummy l, i), _
      dummyAsF l=0)::OutputForm = _
      (values.(i+1))::OutputForm * _
      factorial(box(i::R::F)$F)::OutputForm _
      for i in 0..min(4,#values-5)])

bracket(hconcat([bracket((displayVariable l)::OutputForm ** _
  (operatorArgument l)::OutputForm),
    (op(displayVariable l))::OutputForm, ": ",
    commaSeparate vals)])

setProperty(opADE, "%specialDisp",
  ddADE$((List F -> OutputForm) pretend None)

——

— RECP.dotabb —

"RECP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RECOP"]
package RDIV ReducedDivisor

— ReducedDivisor.input —

)set break resume
)sys rm -f ReducedDivisor.output
)spool ReducedDivisor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ReducedDivisor
--R
--R ReducedDivisor(F1: Field,UP: UnivariatePolynomialCategory(F1),UPUP: UnivariatePolynomialCategory(Fraction(UP)),R: FunctionFieldCategory(F1,UP,UPUP),F2: Join(Finite,Field)) is a package constructor
--R Abbreviation for ReducedDivisor is RDIV
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RDIV
--R
--R------------------------------- Operations --------------------------------
--R order : (FiniteDivisor(F1,UP,UPUP,R),UPUP,(F1 -> F2)) -> NonNegativeInteger
--R
--E 1

)spool
)lisp (bye)

— ReducedDivisor.help —

====================================================================
ReducedDivisor examples
====================================================================

Finds the order of a divisor over a finite field

See Also:
  o )show ReducedDivisor
ReducedDivisor (RDIV)

Exports:
order

— package RDIV ReducedDivisor —

)abbrev package RDIV ReducedDivisor
++ Author: Manuel Bronstein
++ Date Created: 1988
++ Date Last Updated: 8 November 1994
++ Description:
++ Finds the order of a divisor over a finite field

ReducedDivisor(F1, UP, UPUP, R, F2): Exports == Implementation where
F1 : Field
UP : UnivariatePolynomialCategory F1
UPUP : UnivariatePolynomialCategory Fraction UP
R : FunctionFieldCategory(F1, UP, UPUP)
F2 : Join(Finite, Field)

N ==> NonNegativeInteger
FD ==> FiniteDivisor(F1, UP, UPUP, R)
UP2 ==> SparseUnivariatePolynomial F2
UPUP2 ==> SparseUnivariatePolynomial Fraction UP2

Exports ==> with
  order: (FD, UPUP, F1 -> F2) -> N
  ++ order(f,u,g) \undocumented

Implementation ==> add
  algOrder : (FD, UPUP, F1 -> F2) -> N
rootOrder: (FD, UP, N, F1 -> F2) -> N

-- pp is not necessarily monic
order(d, pp, f) ==
  (r := retractIfCan(reductum pp)@Union(Fraction UP, "failed"))
  case "failed" => algOrder(d, pp, f)
  rootOrder(d, - retract(r::Fraction(UP) / leadingCoefficient pp)@UP,
             degree pp, f)

algOrder(d, modulus, reduce) ==
  redmod := map(reduce, modulus)$MultipleMap(F1,UP,UPUP,F2,UP2,UPUP2)
  curve := AlgebraicFunctionField(F2, UP2, UPUP2, redmod)
  order(map(reduce,
          d)$FiniteDivisorFunctions2(F1,UP,UPUP,R,F2,UP2,UPUP2,curve)
          )$FindOrderFinite(F2, UP2, UPUP2, curve)

rootOrder(d, radicand, n, reduce) ==
  redrad := map(reduce,
                radicand)$UnivariatePolynomialCategoryFunctions2(F1,UP,F2,UP2)
  curve := RadicalFunctionField(F2, UP2, UPUP2, redrad::Fraction UP2, n)
  order(map(reduce,
         d)$FiniteDivisorFunctions2(F1,UP,UPUP,R,F2,UP2,UPUP2,curve)
         )$FindOrderFinite(F2, UP2, UPUP2, curve)

package ODERED ReduceLODE

— ReduceLODE.input —

)set break resume
)sys rm -f ReduceLODE.output
)spool ReduceLODE.output
)set message test on
)set message auto off
)clear all
---S 1 of 1
)show ReduceLODE
--R
--R ReduceLODE(F: Field,L: LinearOrdinaryDifferentialOperatorCategory(F),UP: UnivariatePolynomialCategory(F),A: MonogenicAlgebra(F,UP),LO: LinearOrdinaryDifferentialOperatorCategory(A)) is a package constructor
--R Abbreviation for ReduceLODE is ODERED
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODERED
--R
--R--------------------------------------------------- Operations --------------------------
--R reduceLODE : (LO,A) -> Record(mat: Matrix(L),vec: Vector(F))
--R
--E 1

)spool
)lisp (bye)

---

— ReduceLODE.help —

====================================================================
ReduceLODE examples
====================================================================

Elimination of an algebraic from the coefficients of a linear ordinary
differential equation.

See Also:
  o )show ReduceLODE

---
ReduceLODE (ODERED)

Exports:
reduceLODE

— package ODERED ReduceLODE —

)abbrev package ODERED ReduceLODE
++ Author: Manuel Bronstein
++ Date Created: 19 August 1991
++ Date Last Updated: 11 April 1994
++ Description:
++ Elimination of an algebraic from the coefficients
++ of a linear ordinary differential equation.

ReduceLODE(F, L, UP, A, LO): Exports == Implementation where
F : Field
L : LinearOrdinaryDifferentialOperatorCategory F
UP: UnivariatePolynomialCategory F
A : MonogenicAlgebra(F, UP)
LO: LinearOrdinaryDifferentialOperatorCategory A

V ==> Vector F
M ==> Matrix L

Exports ==> with
reduceLODE: (LO, A) -> Record(mat:M, vec:V)
  ++ reduceLODE(op, g) returns \spad{[m, v]} such that
  ++ any solution in \spad{A} of \spad{op z = g}
  ++ is of the form \spad{z = (z_1,...,z_m) . (b_1,...,b_m)} where
  ++ the \spad{b_i's} are the basis of \spad{A} over \spad{F} returned
  ++ by \spad{spadfun{basis}()} from \spad{A}, and the \spad{z_i's} satisfy the
  ++ differential system \spad{M.z = v}.

Implementation ==> add
matF2L: Matrix F -> M
diff := D()$L

-- coerces a matrix of elements of F into a matrix of (order 0) L.O.D.O's
matF2L m ==
  map((f1:F):L+->f1::L, m)$MatrixCategoryFunctions2(F, V, V, Matrix F, L, Vector L, Vector L, M)

-- This follows the algorithm and notation of
-- "The Risch Differential Equation on an Algebraic Curve", M. Bronstein,
reduceLODE(l, g) ==
  n := rank()$A
  -- md is the basic differential matrix (D x I + Dy)
  md := matF2L transpose derivationCoordinates(basis(), (f1:F):F+->diff f1)
  for i in minRowIndex md .. maxRowIndex md
    for j in minColIndex md .. maxColIndex md repeat
      md(i, j) := diff + md(i, j)
  -- mdi will go through the successive powers of md
  mdi := copy md
  sys := matF2L(transpose regularRepresentation coefficient(l, 0))
  for i in 1..degree l repeat
    sys := sys +
      matF2L(transpose regularRepresentation coefficient(l, i)) * mdi
  mdi := md * mdi
  [sys, coordinates g]
ReductionOfOrder(R) provides functions for reducing the order of linear ordinary differential equations once some solutions are known.

See Also:

- )show ReductionOfOrder

---

ReductionOfOrder examples

---

ReductionOfOrder provides functions for reducing the order of linear ordinary differential equations once some solutions are known.

See Also:

- )show ReductionOfOrder

---
ReductionOfOrder (REDORDER)

Exports:
ReduceOrder

— package REDORDER ReductionOfOrder —

)abbrev package REDORDER ReductionOfOrder
++ Author: Manuel Bronstein
++ Date Created: 4 November 1991
++ Date Last Updated: 3 February 1994
++ Description:
++ \spadtype{ReductionOfOrder} provides
++ functions for reducing the order of linear ordinary differential equations
++ once some solutions are known.

ReductionOfOrder(F, L): Exports == Impl where
  F: Field
  L: LinearOrdinaryDifferentialOperatorCategory F

Z ==> Integer
A ==> PrimitiveArray F

Exports ==> with
ReduceOrder: (L, F) -> L
  ++ ReduceOrder(op, s) returns \spad{op1} such that for any solution
  ++ \spad{op1 z = 0}, \spad{y = s \int z} is a solution of
  ++ \spad{op y = 0}. \spad{s} must satisfy \spad{op s = 0}.
ReduceOrder: (L, List F) -> Record(eq:L, op:List F)
  ++ ReduceOrder(op, \[f1,...,fk\]) returns \spad{[op1,\[g1,...,gk\]]} such that
  ++ for any solution \spad{op1 z = 0},
  ++ \spad{y = gk \int(g_{k-1} \int(... \int(g1 \int z)...)} is a solution
  ++ of \spad{op y = 0}. Each \spad{fi} must satisfy \spad{op fi = 0}.

Impl ==> add
ithcoef : (L, Z, A) -> F
locals       : (A, Z, Z) -> F
localbinom: (Z, Z) -> Z

\[ \text{diff} := D(), \text{L} \]

localbinom(j, i) == (j > i => binomial(j, i+1); 0)
locals(s, j, i) == (j > i => qelt(s, j - i - 1); 0)

ReduceOrder(l:L, sols:List F) ==
empty? sols => [l, empty()]
neweq := ReduceOrder(l, sol := first sols)
rec := ReduceOrder(neweq, [diff(s / sol) for s in rest sols])
[rec.eq, concat_!(rec.op, sol)]

ithcoef(eq, i, s) ==
ans:F := 0
while eq ^= 0 repeat
  j := degree eq
  ans := ans + localbinom(j, i) * locals(s, j, i) * leadingCoefficient eq
  eq := reductum eq
ans

ReduceOrder(eq:L, sol:F) ==
s:A := new(n := degree eq, 0) -- will contain derivatives of sol
si := sol -- will run through the derivatives
qsetelt_!(s, 0, si)
for i in 1..(n-1)::NonNegativeInteger repeat
  qsetelt_!(s, i, si := diff si)
ans:L := 0
for i in 0..(n-1)::NonNegativeInteger repeat
  ans := ans + monomial(ithcoef(eq, i, s), i)
ans

REDORDER.dotabb

"REDORDER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REDORDER"]
"OREPCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OREPCAT"]
"A1AGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=A1AGG"]
"REDORDER" -> "OREPCAT"
"REDORDER" -> "A1AGG"
package RSDCMPK RegularSetDecompositionPackage

--- RegularSetDecompositionPackage.input ---

)set break resume
)sys rm -f RegularSetDecompositionPackage.output
)spool RegularSetDecompositionPackage.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show RegularSetDecompositionPackage

--R RegularSetDecompositionPackage(R: GcdDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategory(R,E,V),TS: RegularTriangularSetCategory(R,E,V,P)) is a package constructor

--R Abbreviation for RegularSetDecompositionPackage is RSDCMPK

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RSDCMPK

--R

--R------------------------------------- Operations -------------------------------------

--R KrullNumber : (List(P),List(TS)) -> NonNegativeInteger

--R algebraicDecompose : (P,TS,Boolean) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))

--R convert : Record(val: List(P),tower: TS) -> String

--R decompose : (List(P),List(TS),Boolean,Boolean) -> List(TS)

--R decompose : (List(P),List(TS),Boolean,Boolean,Boolean,Boolean,Boolean) -> List(TS)

--R internalDecompose : (P,TS,NonNegativeInteger,Boolean) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))

--R internalDecompose : (P,TS,NonNegativeInteger) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))

--R internalDecompose : (P,TS) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))

--R numberOfVariables : (List(P),List(TS)) -> NonNegativeInteger

--R printInfo : (List(Record(val: List(P),tower: TS)),NonNegativeInteger) -> Void

--R transcendentalDecompose : (P,TS,NonNegativeInteger) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))

--R transcendentalDecompose : (P,TS) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))

--R upDateBranches : (List(P),List(TS),List(Record(val: List(P),tower: TS)),Record(done: List(TS),todo: List(Record(val: List(P),tower: TS))))

--R

--E 1

)spool
)lisp (bye)

---

--- RegularSetDecompositionPackage.help ---

====================================================================
RegularSetDecompositionPackage examples
====================================================================

A package providing a new algorithm for solving polynomial systems by
means of regular chains. Two ways of solving are proposed: in the sense of Zariski closure (like in Kalkbrener’s algorithm) or in the sense of the regular zeros (like in Wu, Wang or Lazard methods). This algorithm is valid for any type of regular set. It does not care about the way a polynomial is added in an regular set, or how two quasi-components are compared (by an inclusion-test), or how the invertibility test is made in the tower of simple extensions associated with a regular set.

These operations are realized respectively by the domain TS and the packages QCMPACK(R,E,V,P,TS) and RSETGCD(R,E,V,P,TS).

The same way it does not care about the way univariate polynomial gcd (with coefficients in the tower of simple extensions associated with a regular set) are computed. The only requirement is that these gcd need to have invertible initials (normalized or not).

WARNING. There is no need for a user to call directly any operation of this package since they can be accessed by the domain TS. Thus, the operations of this package are not documented.

See Also:
◦ )show RegularSetDecompositionPackage

RegularSetDecompositionPackage (RSDCMPK)

Exports:
KrullNumber algebraicDecompose convert decompose
internalDecompose numberOfVariables printInfo transcendentalDecompose
upDateBranches

— package RSDCMPK RegularSetDecompositionPackage —
ABBREV PACKAGE RSDCMPK RegularSetDecompositionPackage
  ++ Author: Marc Moreno Maza
  ++ Date Created: 09/16/1998
  ++ Date Last Updated: 12/16/1998
  ++ References :
  ++ Description:
  ++ A package providing a new algorithm for solving polynomial systems
  ++ by means of regular chains. Two ways of solving are proposed:
  ++ in the sense of Zariski closure (like in Kalkbrener's algorithm)
  ++ or in the sense of the regular zeros (like in Wu, Wang or Lazard
  ++ methods). This algorithm is valid for any type
  ++ of regular set. It does not care about the way a polynomial is
  ++ added in an regular set, or how two quasi-components are compared
  ++ (by an inclusion-test), or how the invertibility test is made in
  ++ the tower of simple extensions associated with a regular set.
  ++ These operations are realized respectively by the domain \spad{TS}
  ++ and the packages
  ++ \texttt{axiomType{QCMPACK}}(R,E,V,P,TS) and \texttt{axiomType{RSETGCD}}(R,E,V,P,TS).
  ++ The same way it does not care about the way univariate polynomial
  ++ gcd (with coefficients in the tower of simple extensions associated
  ++ with a regular set) are computed. The only requirement is that these
  ++ gcd need to have invertible initials (normalized or not).
  ++ WARNING. There is no need for a user to call directly any operation
  ++ of this package since they can be accessed by the domain \spad{TS}.
  ++ Thus, the operations of this package are not documented.

RegularSetDecompositionPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LP ==> List P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Wip ==> Record(done: Split, todo: List LpWT)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,\texttt{"failed"})
Split ==> List TS
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> QuasiComponentPackage(R,E,V,P,TS)
CHAPTER 19. CHAPTER R

regsetgcdpack ==> RegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports == with

KrullNumber: (LP, Split) -> N
numberOfVariables: (LP, Split) -> N
algebraicDecompose: (P,TS,B) -> Record(done: Split, todo: List LpWT)
transcendentalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS,N,B) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
decompose: (LP, Split, B, B) -> Split
decompose: (LP, Split, B, B, B, B, B) -> Split
upDateBranches: (LP,Split,List LpWT,Wip,N) -> List LpWT
convert: Record(val: List P,tower: TS) -> String
printInfo: (List Record(val: List P,tower: TS), N) -> Void

Implementation == add

KrullNumber(lp: LP, lts: Split): N ==
ln: List N := [#(ts) for ts in lts]
n := #lp + reduce(max,ln)

numberOfVariables(lp: LP, lts: Split): N ==
lv: List V := variables([lp]$PS)
for ts in lts repeat lv := concat(variables(ts), lv)
# removeDuplicates(lv)

ground? p =>
    error " in algebraicDecompose$REGSET: should never happen !"
v := mvar(p); n := #ts
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
ts_v := select(ts,v)::P
if mdeg(p) < mdeg(ts_v)
    then
        lgwt := internalLastSubResultant(ts_v,p,ts_v_-true,false)$regsetgcdpack
    else
        lgwt := internalLastSubResultant(p,ts_v,ts_v_-true,false)$regsetgcdpack
lts: Split := []
llpwt: List LpWT := []
for gwt in lgwt repeat
g := gwt.val; us := gwt.tower
zero? g =>
    error " in algebraicDecompose$REGSET: should never happen !"
ground? g => "leave"
if mvar(g) = v then lts := concat(members(ts_v_+),augment(g,us)),lts)
h := leadingCoefficient(g,v)
b: Boolean := purelyAlgebraic?(us)
lsfp := squareFreeFactors(h)$polsetpack
lus := augment(members(ts_v_+), augment(ts_v,us)@Split)
for f in lsfp repeat
  ground? f => "leave"
  b and purelyAlgebraic?(f,us) => "leave"
for vs in lus repeat
  llpwt := cons([[f,p],vs]$LpWT, llpwt)
[lts,llpwt]

lts: Split
if #ts < bound then
  lts := augment(p,ts)
else
  lts := []
llpwt: List LpWT := []
[lts,llpwt]

lts: Split:= augment(p,ts)
llpwt: List LpWT := []
[lts,llpwt]

clos? => internalDecompose(p,ts,bound)
internalDecompose(p,ts)

  -- ASSUME p not constant
  llpwt: List LpWT := []
lts: Split := []
  -- EITHER mvar(p) is null
  if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p))) then
    llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
p := (p exquo lmp)::P
ip := squareFreePart init(p); tp := tail p
p := mainPrimitivePart p
  -- OR init(p) is null or not
  lbwt := invertible?(ip,ts)@(List BWT)
  for bwt in lbwt repeat
    bwt.val =>
      if algebraic?(mvar(p),bwt.tower) then
        rsl := algebraicDecompose(p,bwt.tower,true)
      else
        rsl := transcendentalDecompose(p,bwt.tower,bound)
lts := concat(rsl.done,lts)
llpwt := concat(rsl.todo,llpwt)

-- purelyAlgebraicLeadingMonomial?(ip,bwt.tower) => "leave" -- UNPROVED CRITERIA
purelyAlgebraic?(ip,bwt.tower) and purelyAlgebraic?(bwt.tower) => "leave" -- SAFE
(not ground? ip) =>
  zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
  (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)

riv := removeZero(ip,bwt.tower)
(zero? riv) =>
  zero? tp => lts := cons(bwt.tower,lts)
  (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
[lts,llpwt]


-- ASSUME p not constant
llpwt: List LpWT := []
lts: Split := []

-- EITHER mvar(p) is null
if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
then
  llpwt := cons([[mvar(p)::P],ts]$LpWT, llpwt)
  p := (p exquo lmp)::P
  ip := squareFreePart init(p); tp := tail p
  p := mainPrimitivePart p
-- OR init(p) is null or not
lbwt := invertible?(ip,ts)@(List BWT)
  for bwt in lbwt repeat
    bwt.val =>
      if algebraic?(mvar(p),bwt.tower)
      then
        rsl := algebraicDecompose(p,bwt.tower,false)
      else
        rsl := transcendentalDecompose(p,bwt.tower)
lts := concat(rsl.done,lts)
llpwt := concat(rsl.todo,llpwt)
purelyAlgebraic?(ip,bwt.tower) and purelyAlgebraic?(bwt.tower) => "leave"
(not ground? ip) =>
  zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
  (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)

riv := removeZero(ip,bwt.tower)
(zero? riv) =>
  zero? tp => lts := cons(bwt.tower,lts)
  (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
[lts,llpwt]

decompose(lp,lts,false,false,clos?,true,info?)

convert(lpwt: LpWT): String ==
ls: List String := ["<", string(#(lpwt.val)::Z), ",", string(#(lpwt.tower)::Z), ">"]
concat ls

printInfo(toSee: List LpWT, n: N): Void ==
lpwt := first toSee
s: String := concat ["[", string(#toSee)::Z], " ", convert(lpwt)@String]
m: N := #(lpwt.val)
toSee := rest toSee
for lpwt in toSee repeat
  m := m + #(lpwt.val)
  s := concat [s, ",", convert(lpwt)@String]
  s := concat [s, "+", string(m::Z), "; 
               
               
               \{"; string(n::Z),"\}]]
iprint(s)$iprintpack
void()

  -- if cleanW? then REMOVE REDUNDANT COMPONENTS in lts
  -- if sqfr? then SPLIT the system with SQUARE-FREE FACTORIZATION
  -- if clos? then SOLVE in the closure sense
  -- if rem? then REDUCE the current p by using remainder
  -- if info? then PRINT info
empty? lp => lts
branches: List Branch := prepareDecompose(lp,lts,cleanW?,sqfr?)$quasicomppack
empty? branches => []
toSee: List LpWT := [[br.eq,br.tower]$LpWT for br in branches]
toSave: Split := []
if clos? then bound := KrullNumber(lp,lts) else bound := numberOfVariables(lp,lts)
while (not empty? toSee) repeat
  if info? then printInfo(toSee,#toSave)
  lpwt := first toSee; toSee := rest toSee
  lp := lpwt.val; ts := lpwt.tower
  empty? lp =>
    toSave := cons(ts, toSave)
  p := first lp;  lp := rest lp
  if rem? and (not ground? p) and (not empty? ts)
    then
      p := remainder(p,ts).polnum
      p := removeZero(p,ts)
      zero? p => toSee := cons([lp,ts]$LpWT, toSee)
      ground? p => "leave"
      rs1 := internalDecompose(p,ts,bound,clos?)
      toSee := upDateBranches(lp,toSave,toSee,rs1,bound)
      removeSuperfluousQuasiComponents(toSave)$quasicomppack
newBranches: List LpWT := wip.todo
newComponents: Split := wip.done
branches1, branches2: List LpWT
branches1 := []; branches2 := []
for branch in newBranches repeat
us := branch.tower
#us > n => "leave"
newleq := sort(infRittWu?, concat(lt, branch.val))
--foo := rewriteSetWithReduction(newleq, us, initiallyReduce, initiallyReduced?)
--any?(ground?, foo) => "leave"
branches1 := cons([newleq, us] $LpWT, branches1)
for us in newComponents repeat
#us > n => "leave"
subQuasiComponent?(us, lts) $quasicomppack => "leave"
--newleq := leq
--foo := rewriteSetWithReduction(newleq, us, initiallyReduce, initiallyReduced?)
--any?(ground?, foo) => "leave"
branches2 := cons([leq, us] $LpWT, branches2)
empty? branches1 =>
empty? branches2 => current
concat(branches2, current)
branches := concat(branches2, branches1, current)
-- branches := concat(branches, current)
removeSuperfluousCases(branches) $quasicomppack

— RSDCMPK.dotabb —

"RSDCMPK" [color="#FF4488", href="bookvol10.4.pdf#nameddest=RSDCMPK"]
"RSETCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=RSETCAT"]
"RSDCMPK" -> "RSETCAT"

— RSETGCD —

package RSETGCD RegularTriangularSetGcdPackage

— RegularTriangularSetGcdPackage.input —

)set break resume
)sys rm -f RegularTriangularSetGcdPackage.output
)spool RegularTriangularSetGcdPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RegularTriangularSetGcdPackage

--R
--R RegularTriangularSetGcdPackage(R: GcdDomain, E: OrderedAbelianMonoidSup, V: OrderedSet, P: RecursivePolynomialCategory(R, E, V), TS: RegularTriangularSetCategory(R, E, V, P)) is a package constructor
--R Abbreviation for RegularTriangularSetGcdPackage is RSETGCD
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RSETGCD
--R
--R-------------------------------- Operations --------------------------------
--R stopTableGcd! : () -> Void stopTableInvSet! : () -> Void
--R toseInvertible?: (P,TS) -> Boolean
--R integralLastSubResultant : (P,P,TS) -> List(Record(val: P,tower: TS))
--R internalLastSubResultant : (P,P,TS,Boolean,Boolean) -> List(Record(val: P,tower: TS))
--R internalLastSubResultant : (List(Record(val: List(P),tower: TS)),V,Boolean) -> List(Record(val: P,tower: TS))
--R prepareSubResAlgo : (P,P,TS) -> List(Record(val: List(P),tower: TS))
--R startTableGcd! : (String,String,String) -> Void
--R startTableInvSet! : (String,String,String) -> Void
--R toseInvertible? : (P,TS) -> List(Record(val: Boolean,tower: TS))
--R toseInvertibleSet : (P,TS) -> List(TS)
--R toseLastSubResultant : (P,P,TS) -> List(Record(val: P,tower: TS))
--R toseSquareFreePart : (P,TS) -> List(Record(val: P,tower: TS))
--R
--E 1

)spool
)lisp (bye)

— RegularTriangularSetGcdPackage.help —

====================================================================
RegularTriangularSetGcdPackage examples
====================================================================

An internal package for computing gcds and resultants of univariate
polynomials with coefficients in a tower of simple extensions of a field.

See Also:
o )show RegularTriangularSetGcdPackage

———
RegularTriangularSetGcdPackage (RSETGCD)

Exports:
stopTableGcd!  stopTableInvSet!  integralLastSubResultant
prepareSubResAlgo  startTableGcd!  startTableInvSet!
toseInvertible?  toseInvertibleSet  toseLastSubResultant
toseSquareFreePart

— package RSETGCD RegularTriangularSetGcdPackage —

)abbrev package RSETGCD RegularTriangularSetGcdPackage
++ Author: Marc Moreno Maza (marc@nag.co.uk)
++ Date Created: 08/30/1998
++ Date Last Updated: 12/15/1998
++ References :
++ [1] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++ algebraic towers of simple extensions" In proceedings of AAECC11
++ d’extensions simples et resolution des systemes d’équations
++ Description :
++ An internal package for computing gcds and resultants of univariate
++ polynomials with coefficients in a tower of simple extensions of a field.

RegularTriangularSetGcdPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N => NonNegativeInteger
Z => Integer
B => Boolean
\begin{verbatim}
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
EntryGcd ==> List PWT
HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
KeyInvSet ==> Record(arg1: P, arg3: TS)
EntryInvSet ==> List TS
HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> QuasiComponentPackage(R,E,V,P,TS)

Exports == with
  startTableGcd!: (S,S,S) -> Void
  \texttt{startTableGcd!(s1,s2,s3)}  
  \texttt{is an internal subroutine, exported only for developement.}
  stopTableGcd!: () -> Void
  \texttt{stopTableGcd!()} 
  \texttt{is an internal subroutine, exported only for developement.}
  startTableInvSet!: (S,S,S) -> Void
  \texttt{startTableInvSet!(s1,s2,s3)}  
  \texttt{is an internal subroutine, exported only for developement.}
  stopTableInvSet!: () -> Void
  \texttt{stopTableInvSet!()} \texttt{is an internal subroutine,}
  \texttt{exported only for developement.}
  prepareSubResAlgo: (P,P,TS) -> List LpWT
  \texttt{prepareSubResAlgo(p1,p2,ts)} 
  \texttt{is an internal subroutine, exported only for developement.}
  internalLastSubResultant: (P,P,TS,B,B) -> List PWT
  \texttt{internalLastSubResultant(p1,p2,ts,inv?,break?)} 
  \texttt{is an internal subroutine, exported only for developement.}
  internalLastSubResultant: (List LpWT,V,B) -> List PWT
  \texttt{internalLastSubResultant(lpwt,v,flag)} \texttt{is an internal}
  \texttt{subroutine, exported only for developement.}
  integralLastSubResultant: (P,P,TS) -> List PWT
  \texttt{integralLastSubResultant(p1,p2,ts)} 
  \texttt{is an internal subroutine, exported only for developement.}
  toseLastSubResultant: (P,P,TS) -> List PWT
  \texttt{toseLastSubResultant(p1,p2,ts)} \texttt{has the same specifications}
  \texttt{as lastSubResultant from RegularTriangularSetCategory.}
  toseInvertible?: (P,TS) -> B
  \texttt{toseInvertible?(p1,ts)} \texttt{has the same specifications as}
  \texttt{invertible? from RegularTriangularSetCategory.}
\end{verbatim}
toseInvertible?: (P,TS) -> List BWT
++ \texttt{toseInvertible?\{(p1,p2,ts)\}} has the same specifications as
++ invertible? from \texttt{RegularTriangularSetCategory}.

toseInvertibleSet: (P,TS) -> Split
++ \texttt{toseInvertibleSet\{(p1,p2,ts)\}} has the same specifications as
++ invertibleSet from \texttt{RegularTriangularSetCategory}.

toseSquareFreePart: (P,TS) -> List FWT
++ \texttt{toseSquareFreePart\{(p,ts)\}} has the same specifications as
++ squareFreePart from \texttt{RegularTriangularSetCategory}.

Implementation == add

startTableGcd!(ok: S, ko: S, domainName: S): Void ==
  initTable!(:)$HGcd
  printInfo!(ok,ko)$HGcd
  startStats!(domainName)$HGcd
  void()

stopTableGcd!(): Void ==
  if makingStats?!$HGcd then printStats!(:)$HGcd
  clearTable!(:)$HGcd

startTableInvSet!(ok: S, ko: S, domainName: S): Void ==
  initTable!(:)$HInvSet
  printInfo!(ok,ko)$HInvSet
  startStats!(domainName)$HInvSet
  void()

stopTableInvSet!(): Void ==
  if makingStats?!$HInvSet then printStats!(:)$HInvSet
  clearTable!(:)$HInvSet

  q := primitivePart initiallyReduce(p,ts)
  zero? q => false
  normalized?(q,ts) => true
  v := mvar(q)
  not algebraic?(v,ts) =>
    toCheck: List BWT := toseInvertible?(p,ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val = false => return false
    return true
  ts_v := select(ts,v)::P
  ts_v_- := collectUnder(ts,v)
  lgwt := internalLastSubResultant(ts_v,q,ts_v_-false,\true)
  for gwt in lgwt repeat
    g := gwt.val;
    (not ground? g) and (mvar(g) = v) =>
      return false
  true
q := primitivePart initiallyReduce(p, ts)
zero? q => [[false, ts]$BWT]
normalized?(q, ts) => [[true, ts]$BWT]
v := mvar(q)
not algebraic?(v, ts) =>
lbw: List BWT := []
toCheck: List BWT := toseInvertible?(init(q), ts)@$(List BWT)$
for bwt in toCheck repeat
  bwt.val => lbwt := cons(bwt, lbwt)
  newq := removeZero(q, bwt.tower)
  zero? newq => lbwt := cons(bwt, lbwt)
  lbwt := concat(toseInvertible?(newq, bwt.tower)@$(List BWT)$, lbwt)
return lbwt

ts_v := select(ts, v)::P
ts_v_- := collectUnder(ts, v)
ts_v_+ := collectUpper(ts, v)
lgwt := internalLastSubResultant(ts_v, q, ts_v_, false, false)
lbw: List BWT := []
for gwt in lgwt repeat
  g := gwt.val; ts := gwt.tower
  (ground? g) or (mvar(g) < v) =>
    ts := internalAugment(ts_v, ts)
    ts := internalAugment(members(ts_v_+), ts)
    lbwt := cons([[true, ts]$BWT], lbwt)
  g := mainPrimitivePart g
  ts_g := internalAugment(g, ts)
  ts_g := internalAugment(members(ts_v_+), ts_g)
  -- USE internalAugment with parameters ?
  lbwt := cons([[false, ts_g]$BWT], lbwt)
  h := lazyPquo(ts_v, g)
  (ground? h) or (mvar(h) < v) => "leave"
  h := mainPrimitivePart h
  ts_h := internalAugment(h, ts)
  ts_h := internalAugment(members(ts_v_+), ts_h)
  -- USE internalAugment with parameters ?
  -- CAN BE OPTIMIZED if the input tower is separable
  inv := toseInvertible?(q, ts_h)@$(List BWT)$
  lbwt := concat([bwt for bwt in inv | bwt.val]@$(List BWT)$)
  sort((x, y) => x.val < y.val, lbwt)

k: KeyInvSet := [p, ts]
e := extractIfCan(k)@$HInvSet$
e case EntryInvSet => e::EntryInvSet
q := primitivePart initiallyReduce(p, ts)
zero? q => []
normalized?(q, ts) => [ts]
v := mvar(q)
toSave: Split := []
not algebraic?(v,ts) =>
toCheck: List BWT := toseInvertible?(init(q),ts)@(List BWT)
for bwt in toCheck repeat
  bwt.val => toSave := cons(bwt.tower,toSave)
  newq := removeZero(q,bwt.tower)
  zero? newq => "leave"
  toSave := concat(toseInvertibleSet(newq,bwt.tower), toSave)
toSave := removeDuplicates toSave
return algebraicSort(toSave)$quasicomppack
ts_v := select(ts,v)::P
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
lgwt := internalLastSubResultant(ts_v,q,ts_v_-,false,false)
for gwt in lgwt repeat
g := gwt.val; ts := gwt.tower
(ground? g) or (mvar(g) < v) =>
ts := internalAugment(ts_v,ts)
ts := internalAugment(members(ts_v_+),ts)
toSave := cons(ts,toSave)
g := mainPrimitivePart g
h := lazyPquo(ts_v,g)
h := mainPrimitivePart h
(ground? h) or (mvar(h) < v) => "leave"
ts_h := internalAugment(h,ts)
ts_h := internalAugment(members(ts_v_+),ts_h)
inv := toseInvertibleSet(q,ts_h)
toSave := removeDuplicates concat(inv,toSave)
toSave := algebraicSort(toSave)$quasicomppack
insert!(k,toSave)$HInvSet
toSave

-- ASSUME p is not constant and mvar(p) > mvar(ts)
-- ASSUME init(p) is invertible w.r.t. ts
-- ASSUME p is mainly primitive
-- one? mdeg(p) => [[p,ts]$PWT]
mdeg(p) = 1 => [[p,ts]$PWT]
v := mvar(p)$P
q: P := mainPrimitivePart D(p,v)
lgwt: List PWT := internalLastSubResultant(p,q,ts,true,false)
lwt : List PWT := []
sfp : P
for gwt in lgwt repeat
g := gwt.val; us := gwt.tower
(ground? g) or (mvar(g) < v) =>
lwt := cons([p,us],lwt)
g := mainPrimitivePart g
sfp := lazyPquo(p,g)
sfp := mainPrimitivePart stronglyReduce(sfp,us)
lpwt := cons([sfp,us],lpwt)
lpwt


  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME init(p1) invertible modulo ts !!!
toSee: List LpWT := [[[p1,p2],ts]$LpWT]
toSave: List LpWT := []
v := mvar(p1)

while (not empty? toSee) repeat
  lpwt := first toSee; toSee := rest toSee
  p1 := lpwt.val.1; p2 := lpwt.val.2
  ts := lpwt.tower
  lbwt := toseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
  for bwt in lbwt repeat
    (bwt.val = true) and (degree(p2,v) > 0) =>
      p3 := prem(p1, -p2)
      s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
      toSave := cons([[p2,p3,s],bwt.tower]$LpWT,toSave)
      -- p2 := initiallyReduce(p2,bwt.tower)
      newp2 := primitivePart initiallyReduce(p2,bwt.tower)
      (bwt.val = true) =>
      -- toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
      toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
      -- zero? p2 =>
      zero? newp2 =>
      toSave := cons([[p1,0,1],bwt.tower]$LpWT,toSave)
      -- toSee := cons([[p1,p2],ts]$LpWT,toSee)
      toSee := cons([[p1,newp2],bwt.tower]$LpWT,toSee)
      toSave

  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME p1 and p2 have no algebraic coefficients
  lsr := lastSubResultant(p1, p2)
  ground?(lsr) => [[lsr,ts]$PWT]
  mvar(lsr) < mvar(p1) => [[lsr,ts]$PWT]
  gii2 := gcd(init(p1),init(p2))
  ex: Union(P,"failed") := (gii2 * lsr) exquo init(lsr)
  ex case "failed" => [[lsr,ts]$PWT][[ex::P,ts]$PWT]

  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- if b1 ASSUME init(p2) invertible w.r.t. ts
  -- if b2 BREAK with the first non-trivial gcd

k: KeyGcd := [p1, p2, ts, b2]
e := extractIfCan(k)HGcd
e case EntryGcd => e::EntryGcd
toSave: List PWT
empty? ts =>
    toSave := integralLastSubResultant(p1, p2, ts)
    insert!(k, toSave)HGcd
    return toSave
toSee: List LpWT
if b1
    then
        p3 := prem(p1, -p2)
        s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
        toSee := [[p2, p3, s], ts]$LpWT
    else
        toSee := prepareSubResAlgo(p1, p2, ts)
toSave := internalLastSubResultant(toSee, mvar(p1), b2)
insert!(k, toSave)HGcd
toSave

internalLastSubResultant(llpwt: List LpWT, v: V, b2: B): List PWT ==
toReturn: List PWT := []; toSee: List LpWT;
while (not empty? llpwt) repeat
    toSee := llpwt; llpwt := []
    -- CONSIDER FIRST the vanishing current last subresultant
    for lpwt in toSee repeat
        p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3; ts := lpwt.tower
        lbwt := toseInvertible?(leadingCoefficient(p2, v), ts)@(List BWT)
        for bwt in lbwt repeat
            bwt.val = false =>
                toReturn := cons([p1, bwt.tower]$PWT, toReturn)
            b2 and positive?(degree(p1, v)) => return toReturn
            llpwt := cons([[p1, p2, s], bwt.tower]$LpWT, llpwt)
        empty? llpwt => "leave"
        -- CONSIDER NOW the branches where the computations continue
        toSee := llpwt; llpwt := []
        lpwt := first toSee; toSee := rest toSee
        p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3
        delta: N := (mdeg(p1) - degree(p2, v))::N
        p3: P := LazardQuotient2(p2, leadingCoefficient(p2, v), s, delta)
        zero?(degree(p3, v)) =>
            toReturn := cons([p3, lpwt.tower]$PWT, toReturn)
        for lpwt in toSee repeat
            toReturn := cons([p3, lpwt.tower]$PWT, toReturn)
        (p1, p2) := (p3, next_subResultant2(p1, p2, p3, s))
        s := leadingCoefficient(p1, v)
        llpwt := cons([[p1, p2, s], lpwt.tower]$LpWT, llpwt)
    for lpwt in toSee repeat
        llpwt := cons([[p1, p2, s], lpwt.tower]$LpWT, llpwt)
toReturn
ground? p1 =>
  error"in toseLastSubResultantElseSplit$TOSEGCD: bad #1"
ground? p2 =>
  error"in toseLastSubResultantElseSplit$TOSEGCD: bad #2"
not (mvar(p2) = mvar(p1)) =>
  error"in toseLastSubResultantElseSplit$TOSEGCD: bad #2"
algebraic?(mvar(p1),ts) =>
  error"in toseLastSubResultantElseSplit$TOSEGCD: bad #1"
not initiallyReduced?(p1,ts) =>
  error"in toseLastSubResultantElseSplit$TOSEGCD: bad #1"
not initiallyReduced?(p2,ts) =>
  error"in toseLastSubResultantElseSplit$TOSEGCD: bad #2"
purelyTranscendental?(p1,ts) and purelyTranscendental?(p2,ts) =>
  integralLastSubResultant(p1,p2,ts)
if mdeg(p1) < mdeg(p2) then
  (p1, p2) := (p2, p1)
  if odd?(mdeg(p1)) and odd?(mdeg(p2)) then p2 := - p2
  internalLastSubResultant(p1,p2,ts,false,false)

— RSETGCD.dotabb —

"RSETGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RSETGCD"]
"RPOLCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RPOLCAT"]
"RSETGCD" -> "RPOLCAT"

package REPDB RepeatedDoubling

— RepeatedDoubling.input —

)set break resume
)sys rm -f RepeatedDoubling.output
)spool RepeatedDoubling.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show RepeatedDoubling
--R
--R RepeatedDoubling(S) where
--R S: SetCategory with
--R ?+? : (%,% -> % is a package constructor
--R Abbreviation for RepeatedDoubling is REPDB
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for REPDB
--R
--R-------------------------------- Operations --------------------------------
--R double : (PositiveInteger,S) -> S
--R
)--spool
)--lisp (bye)

-- RepeatedDoubling.help --

====================================================================
RepeatedDoubling examples
====================================================================

Implements multiplication by repeated addition

See Also:
  o )show RepeatedDoubling

RepeatedDoubling (REPDB)
Exports:
double

— package REPDB RepeatedDoubling —

)abbrev package REPDB RepeatedDoubling
++ Description:
++ Implements multiplication by repeated addition
-- the following package is only instantiated over %
-- thus shouldn't be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('RepeatedDoubling, $mutableDomains)

RepeatedDoubling(S):Exports ==Implementation where
S: SetCategory with
  "+":(%,%)->%
  ++ x+y returns the sum of x and y
Exports == with
double: (PositiveInteger,S) -> S
  ++ double(i, r) multiplies r by i using repeated doubling.
Implementation == add
x: S
n: PositiveInteger
double(n,x) ==
  -- one? n => x
  (n = 1) => x
  odd?(n)$Integer =>
    x + double(shift(n,-1) pretend PositiveInteger,(x+x))
  double(shift(n,-1) pretend PositiveInteger,(x+x))

— REPDB.dotabb —
package REPSQ RepeatedSquaring

--- RepeatedSquaring.input ---

)set break resume
/sys rm -f RepeatedSquaring.output
/spool RepeatedSquaring.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)>show RepeatedSquaring

--R
--R RepeatedSquaring(S)where
--R S: SetCategorywith
--R ?*? : (%,%) -> % is a package constructor
--R Abbreviation for RepeatedSquaring is REPSQ
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for REPSQ

--) swagger
--R------------------------------- Operations --------------------------------
--R expt : (S,PositiveInteger) -> S

--) E 1

)>spool
)>lisp (bye)

--- RepeatedSquaring.help ---

====================================================================
RepeatedSquaring examples
====================================================================

This code computes $x^n$ by repeated squaring. It takes advantage of the
law of exponents so that if $n=a\cdot b$ then

$$x^n = x^a \cdot x^b$$

It also takes advantage of the fact that $n$ has a binary representation
so that a one-bit right-shift of a number divides it by 2. Thus we get
three cases:

- $n = 1 \rightarrow x^1$
- $n$ odd \rightarrow repeatedsquare$(x\cdot x, n/2) \cdot x$
n even -> repeatedsquared(x*x, n/2)

Since each recursive call divides n by 2 the algorithm is $O(\log(n))$.

See Also:
- )show RepeatedSquaring

---

RepeatedSquaring (REPSQ)

Exports:
expt

— package REPSQ RepeatedSquaring —

)abbrev package REPSQ RepeatedSquaring
++ Description:
++ Implements exponentiation by repeated squaring

-- the following package is only instantiated over %
-- thus shouldn't be cached. We prevent it
-- from being cached by declaring it to be mutableDomains

)bo PUSH('RepeatedSquaring, $mutableDomains)

RepeatedSquaring(S): Exports == Implementation where
S: SetCategory with
"*":(%,%)->%
  ++ x*y returns the product of x and y
Exports == with
  expt: (S,PositiveInteger) -> S
++ expt(r, i) computes r**i by repeated squaring
Implementation == add
x: S
n: PositiveInteger
expt(x, n) ==
-- one? n => x
(n = 1) => x
odd?(n)$Integer => x * expt(x*x, shift(n, -1) pretend PositiveInteger)
expt(x*x, shift(n, -1) pretend PositiveInteger)

---

package REP1 RepresentationPackage1

--- RepresentationPackage1.input ---

)set break resume
)sys rm -f RepresentationPackage1.output
)spool RepresentationPackage1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RepresentationPackage1
--R
--R RepresentationPackage1(R: Ring) is a package constructor
--R Abbreviation for RepresentationPackage1 is REP1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for REP1
--R
--R---------------------------------- Operations ----------------------------------
--R antisymmetricTensors : (Matrix(R), PositiveInteger) -> Matrix(R) if R has commutative(*)
--R antisymmetricTensors : (List(Matrix(R)), PositiveInteger) -> List(Matrix(R)) if R has commutative(*)
--R createGenericMatrix : NonNegativeInteger -> Matrix(Polynomial(R))
RepresentationPackage1 provides functions for representation theory for finite groups and algebras.

The package creates permutation representations and uses tensor products and its symmetric and antisymmetric components to create new representations of larger degree from given ones.

Note that instead of having parameters from Permutation this package allows list notation of permutations as well: e.g. \([1, 4, 3, 2]\) denotes permutes 2 and 4 and fixes 1 and 3.

See Also:
\(\)show RepresentationPackage1
RepresentationPackage1 (REP1)

Exports:
antisymmetricTensors createGenericMatrix permutationRepresentation
symmetricTensors tensorProduct

— package REP1 RepresentationPackage1 —

)abbrev package REP1 RepresentationPackage1
++ Authors: Holger Gollan, Johannes Grabmeier, Thorsten Werther
++ Date Created: 12 September 1987
++ Date Last Updated: 24 May 1991
++ References:
++ G. James, A. Kerber: The Representation Theory of the Symmetric
++ J. Grabmeier, A. Kerber: The Evaluation of Irreducible
++ Polynomial Representations of the General Linear Groups
++ and of the Unitary Groups over Fields of Characteristic 0,
++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++ their Realization in the Computer Algebra System Scratchpad,
++ Bayreuther Mathematische Schriften, Heft 33, 1990, 1-23
++ Description:
++ RepresentationPackage1 provides functions for representation theory
++ for finite groups and algebras.
++ The package creates permutation representations and uses tensor products
++ and its symmetric and antisymmetric components to create new
++ representations of larger degree from given ones.
++ Note that instead of having parameters from \spadtype{Permutation}
++ this package allows list notation of permutations as well:
++ e.g. \spad{[1,4,3,2]} denotes permutes 2 and 4 and fixes 1 and 3.

RepresentationPackage1(R): public == private where

R : Ring
OF => OutputForm
NNI => NonNegativeInteger
if $R$ has commutative("*") then
antisymmetricTensors : (M R, PI) -> M R
++ antisymmetricTensors(a,n) applies to the square matrix
++ a the irreducible, polynomial representation of the
++ general linear group $GL_m$, where $m$ is the number of
++ rows of $a$, which corresponds to the partition
++ $(1,1,...,1,0,0,...,0)$ of $n$.
++ Error: if $n$ is greater than $m$.
++ Note that this corresponds to the symmetrization of the
++ representation with the sign representation of the symmetric group
++ $S_n$. The carrier spaces of the representation are the antisymmetric
++ tensors of the n-fold tensor product.
if $R$ has commutative("*") then
antisymmetricTensors : (L M R, PI) -> L M R
++ antisymmetricTensors(la,n) applies to each
++ m-by-m square matrix in
++ the list $la$ the irreducible, polynomial representation
++ of the general linear group $GL_m$
++ which corresponds
++ to the partition $(1,1,...,1,0,0,...,0)$ of $n$.
++ Error: if $n$ is greater than $m$.
++ Note that this corresponds to the symmetrization of the
++ representation with the sign representation of the symmetric group
++ $S_n$. The carrier spaces of the representation are the antisymmetric
++ tensors of the n-fold tensor product.
createGenericMatrix : NNI -> M P R
++ createGenericMatrix(m) creates a square matrix of dimension $k$
++ whose entry at the i-th row and j-th column is the
++ indeterminate $x[i,j]$ (double subscripted).
symmetricTensors : (M R, PI) -> M R
++ symmetricTensors(a,n) applies to the m-by-m
++ square matrix $a$ the
++ irreducible, polynomial representation of the general linear
++ group $GL_m$
++ which corresponds to the partition $(n,0,...,0)$ of $n$.
++ Error: if $a$ is not a square matrix.
++ Note that this corresponds to the symmetrization of the
representation with the trivial representation of the symmetric
++ group Sn.
++ The carrier spaces of the representation are the symmetric
tensors of the n-fold tensor product.

++ representation with the trivial representation of the symmetric
tensor of the n-fold tensor product.

++ The carrier spaces of the representation are the symmetric
tensors of the n-fold tensor product.

tensorProduct : (L M R, PI) -> L M R
++ tensorProduct(la,n) applies to each m-by-m square matrix in the
++ list la the irreducible, polynomial representation
++ of the general linear group GLm
++ which corresponds
++ to the partition (n,0,...,0) of n.
++ Note that this corresponds to the symmetrization of the
++ representation with the trivial representation of the symmetric
group Sn.
++ The carrier spaces of the representation are the symmetric
tensors of the n-fold tensor product.
tensorProduct : (M R, M R) -> M R
++ tensorProduct(a,b) calculates the Kronecker product
++ of the matrices a and b.
++ Note that if each matrix corresponds to a group representation
++ (repr. of generators) of one group, then these matrices
++ correspond to the tensor product of the two representations.
tensorProduct : (L M R, L M R) -> L M R
++ tensorProduct([a1,...,ak],[b1,...,bk]) calculates the list of
++ Kronecker products of the matrices ai and bi
++ for \(1 \leq i \leq k\).
++ Note that if each list of matrices corresponds to a group
++ representation (repr. of generators) of one group, then these
++ matrices correspond to the tensor product of the two representations.
tensorProduct : M R -> M R
++ tensorProduct(a) calculates the Kronecker product
++ of the matrix a with itself.
tensorProduct : L M R -> L M R
++ tensorProduct([a1,...ak]) calculates the list of
++ Kronecker products of each matrix ai with itself
++ for \(1 \leq i \leq k\).
++ Note that if the list of matrices corresponds to a group
++ representation (repr. of generators) of one group, then these
++ matrices correspond to the tensor product of the representation
++ with itself.

permutationRepresentation : (PERM I, I) -> M I
++ permutationRepresentation(pi,n) returns the matrix
++ \((\delta(i),\pi(i))\) (Kronecker delta) for a permutation
++ \(\pi\) of \(\{1,2,...,n\}\).

permutationRepresentation : L I -> M I
++ permutationRepresentation(pi,n) returns the matrix
++ \((\delta(i),\pi(i))\) (Kronecker delta) if the permutation
++ \(\pi\) is in list notation and permutes \(\{1,2,...,n\}\).

permutationRepresentation : (L PERM I, I) -> L M I
++ permutationRepresentation([pi1,...,pik],n) returns the list
++ of matrices \[(\delta_{i,j},\pi_1(i)),\ldots,(\delta_{i,j},\pi_k(i))\]
++ (Kronecker delta) for the permutations \(\pi_1,\ldots,\pi_k\)
++ of \(\{1,2,\ldots,n\}\).

\textbf{permutationRepresentation : L L I \rightarrow L M I}
++ permutationRepresentation([\pi_1,\ldots,\pi_k],n) returns the list
++ of matrices \[\{(\delta_{i,j},\pi_1(i)),\ldots,(\delta_{i,j},\pi_k(i))\}\]
++ if the permutations \(\pi_1,\ldots,\pi_k\) are in
++ list notation and are permuting \(\{1,2,\ldots,n\}\).

private ==> add

-- import of domains and packages

import OutputForm

-- declaration of local functions:

calcCoef : (L I, M I) \rightarrow I
-- calcCoef(beta,C) calculates the term
-- \(\lvert S(\beta) \gamma S(\alpha) \rvert / \lvert S(\beta) \rvert\)

invContent : L I \rightarrow V I
-- invContent(alpha) calculates the weak monoton function \(f\) with
-- \(f : m \rightarrow n\) with invContent alpha. \(f\) is stored in the returned
-- vector

-- definition of local functions

calcCoef(beta,C) ==
  prod : I := 1
  for i in 1..maxIndex beta repeat
    prod := prod * multinomial(beta(i), entries row(C,i))$ICF
  prod

invContent(alpha) ==
  n : NNI := (+/alpha)::NNI
  f : V I := new(n,0)
  i : NNI := 1
  j : I := - 1
  for og in alpha repeat
    j := j + 1
    for k in 1..og repeat
      f(i) := j
      i := i + 1
    f
-- exported functions:

if R has commutative("*") then
  antisymmetricTensors ( a : M R , k : PI ) ==
  n := nrows a
  k = 1 => a
  k > n =>
    error("second parameter for antisymmetricTensors is too large")
  m := binomial(n,k)
  il := [subset(n,k,i) for i in 0..m-1]
  b := zero(m::NNI, m::NNI)
  for i in 1..m repeat
    for j in 1..m repeat
      c := zero(k,k)
      lr := il.i
      lt := il.j
      for r in 1..k repeat
        for t in 1..k repeat
          rr := lr.r
          tt := lt.t
          c := elt(a, rr, tt)
          setelt(c, r, t, elt(a, rr, tt))
        setelt(b, i, j, determinant c)
  b

if R has commutative("*") then
  antisymmetricTensors(la: L M R, k: PI) ==
    [antisymmetricTensors(ma,k) for ma in la]

symmetricTensors (a : M R, n : PI) ==
  m := nrows a
  m ^= ncols a =>
    error("Input to symmetricTensors is no square matrix")
  n = 1 => a
  dim := (binomial(m+n-1,n)::NNI)
  c := new(dim, dim, 0)
  f := new(n, 0)
  g := new(n, 0)
  nullMatrix := new(1, 0)
  colemanMatrix := M I

for i in 1..dim repeat
   -- unrankImproperPartitions1 starts counting from 0
   alpha := unrankImproperPartitions1(n,m,i-1)$SGCF
   f := invContent(alpha)
for j in 1..dim repeat
   -- unrankImproperPartitions1 starts counting from 0
   beta := unrankImproperPartitions1(n,m,j-1)$SGCF
   g := invContent(beta)
   colemanMatrix := nextColeman(alpha,beta,nullMatrix)$SGCF
   while colemanMatrix ^= nullMatrix repeat
      gamma := inverseColeman(alpha,beta,colemanMatrix)$SGCF
      help := calcCoef(beta,colemanMatrix)::R
      for k in 1..n repeat
         help := help * a((1+f k)::NNI, (1+g(gamma k))::NNI)
      c(i,j) := c(i,j) + help
      colemanMatrix := nextColeman(alpha,beta,colemanMatrix)$SGCF
   -- end of while
   -- end of j-loop
-- end of i-loop

c

symmetricTensors(la : L M R, k : PI) ==
   [symmetricTensors (ma, k) for ma in la]

tensorProduct(a: M R, b: M R) ==
   n : NNI := nrows a
   m : NNI := nrows b
   nc : NNI := ncols a
   mc : NNI := ncols b
   c : M R := zero(n * m, nc * mc)
   indexr : NNI := 1 -- row index
   for i in 1..n repeat
      for k in 1..m repeat
         indexc : NNI := 1 -- column index
         for j in 1..nc repeat
            for l in 1..mc repeat
               c(indexr,indexc) := a(i,j) * b(k,l)
               indexc := indexc + 1
         indexr := indexr + 1
   c

tensorProduct (la: L M R, lb: L M R) ==
   [tensorProduct(la.i, lb.i) for i in 1..maxIndex la]
\text{tensorProduct}(a : M \ R) == \text{tensorProduct}(a, a)

\text{tensorProduct}(la : L \ M \ R) ==
\text{tensorProduct}(la :: L \ M \ R, la :: L \ M \ R)

\text{permutationRepresentation} (p : \text{PERM} I, n : I) ==
\text{-- permutations are assumed to permute \{1,2,\ldots,n\}}
a : M I := \text{zero}(n :: \text{NNI}, n :: \text{NNI})
\text{for i in 1..n repeat}
\quad a(\text{eval}(p,i)$(\text{PERM} I),i) := 1
a

\text{permutationRepresentation} (p : L I) ==
\text{-- permutations are assumed to permute \{1,2,\ldots,n\}}
n : I := \#p
a : M I := \text{zero}(n::\text{NNI}, n::\text{NNI})
\text{for i in 1..n repeat}
\quad a(p.i,i) := 1
a

\text{permutationRepresentation} (\text{listperm} : L \text{PERM} I, n : I) ==
\text{-- permutations are assumed to permute \{1,2,\ldots,n\}}
[\text{permutationRepresentation}(\text{perm}, n) \text{ for perm in listperm}]

\text{permutationRepresentation} (\text{listperm} : L L I) ==
\text{-- permutations are assumed to permute \{1,2,\ldots,n\}}
[\text{permutationRepresentation} \text{perm for perm in listperm}]

\text{createGenericMatrix}(m) ==
\text{res} : M P R := \text{new}(m,m,0$(P R))
\text{for i in 1..m repeat}
\quad \text{for j in 1..m repeat}
\quad\quad \text{iof} : \text{OF} := \text{coerce}(i)$\text{Integer}
\quad\quad \text{jof} : \text{OF} := \text{coerce}(j)$\text{Integer}
\quad\quad \text{le} : L \text{OF} := \text{cons}(\text{iof}, \text{list jof})
\quad\quad \text{sy} : \text{Symbol} := \text{subscript}(x::\text{Symbol}, \text{le}$\text{Symbol}
\quad\quad \text{res}(i,j) := (\text{sy :: P R})
\text{res}

---

--- REP1.dotabb ---

"REP1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REP1"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"REP1" -> "ALIST"
package REP2 RepresentationPackage2

--- RepresentationPackage2.input ---

)set break resume
)sys rm -f RepresentationPackage2.output
)spool RepresentationPackage2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show RepresentationPackage2
--R
--R RepresentationPackage2(R: Ring) is a package constructor
--R Abbreviation for RepresentationPackage2 is REP2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for REP2
--R
--R----------------------------- Operations -----------------------------
--R areEquivalent? : (List(Matrix(R)),List(Matrix(R)),Boolean,Integer) -> Matrix(R) if R has FIELD
--R areEquivalent? : (List(Matrix(R)),List(Matrix(R))) -> Matrix(R) if R has FIELD
--R areEquivalent? : (List(Matrix(R)),List(Matrix(R)),Integer) -> Matrix(R) if R has FIELD
--R completeEchelonBasis : Vector(Vector(R)) -> Matrix(R)
--R createRandomElement : (List(Matrix(R)),Matrix(R)) -> Matrix(R)
--R cyclicSubmodule : (List(Matrix(R)),Vector(R)) -> Vector(Vector(R)) if R has EUCDOM
--R isAbsolutelyIrreducible? : (List(Matrix(R)),Integer) -> Boolean if R has FIELD
--R isAbsolutelyIrreducible? : List(Matrix(R)) -> Boolean if R has FIELD
--R meatAxe : (List(Matrix(R)),Boolean,PositiveInteger) -> List(List(Matrix(R))) if R has FIELD and R has FINITE
--R meatAxe : (List(Matrix(R)),Boolean,Integer,Integer) -> List(List(Matrix(R))) if R has FIELD and R has FINITE
--R meatAxe : (List(Matrix(R)),Integer) -> List(List(Matrix(R))) if R has FIELD and R has FINITE
--R scanOneDimSubspaces : (List(Vector(R)),Integer) -> Vector(R) if R has FIELD and R has FINITE
--R split : (List(Matrix(R)),List(List(Matrix(R)))) -> List(List(Matrix(R))) if R has FIELD
--R split : (List(Matrix(R)),Vector(Vector(R))) -> List(List(Matrix(R))) if R has FIELD
--R standardBasisOfCyclicSubmodule : (List(Matrix(R)),Vector(R)) -> Matrix(R) if R has EUCDOM
--R
--E 1

)spool
)lisp (bye)

---
RepresentationPackage2 provides functions for working with modular representations of finite groups and algebra. The routines in this package are created, using ideas of R. Parker, (the meat-Axe) to get smaller representations from bigger ones, i.e. finding sub- and factormodules, or to show, that such the representations are irreducible.

Note that most functions are randomized functions of Las Vegas type i.e. every answer is correct, but with small probability the algorithm fails to get an answer.

See Also:
- )show RepresentationPackage2

Exports:
- areEquivalent?
- completeEchelonBasis
- createRandomElement
- cyclicSubmodule
- isAbsolutelyIrreducible?
- meatAxe
- scanOneDimSubspaces
- split
- standardBasisOfCyclicSubmodule
)abbrev package REP2 RepresentationPackage2
++ Authors: Holger Gollan, Johannes Grabmeier
++ Date Created: 10 September 1987
++ Date Last Updated: 20 August 1990
++ Reference:
++ R. A. Parker: The Computer Calculation of Modular Characters
++ (The Meat-Axe), in M. D. Atkinson (Ed.), Computational Group Theory
++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++ their Realization in the Computer Algebra System Scratchpad,
++ Description:
++ RepresentationPackage2 provides functions for working with
++ modular representations of finite groups and algebra.
++ The routines in this package are created, using ideas of R. Parker,
++ (the meat-Axe) to get smaller representations from bigger ones,
++ i.e. finding sub- and factormodules, or to show, that such the
++ representations are irreducible.
++ Note that most functions are randomized functions of Las Vegas type
++ i.e. every answer is correct, but with small probability
++ the algorithm fails to get an answer.

RepresentationPackage2(R): public == private where

R : Ring
OF ==> OutputForm
I ==> Integer
L ==> List
SM ==> SquareMatrix
M ==> Matrix
NNI ==> NonNegativeInteger
V ==> Vector
PI ==> PositiveInteger
B ==> Boolean
RADIX ==> RadixExpansion

public == with

  completeEchelonBasis : V V R -> M R
  ++ completeEchelonBasis(lv) completes the basis lv assumed
  ++ to be in echelon form of a subspace of R**n (n the length
  ++ of all the vectors in lv with unit vectors to a basis of
  ++ R**n. It is assumed that the argument is not an empty
  ++ vector and that it is not the basis of the 0-subspace.
  ++ Note that the rows of the result correspond to the vectors
  ++ of the basis.
  createRandomElement : (L M R, M R) -> M R
  ++ createRandomElement(aG,x) creates a random element of the group
  ++ algebra generated by aG.
  -- randomWord : (L L I, L M) -> M R
---+ You can create your own 'random' matrix with "randomWord(lli, lm)".
---+ Each li in lli determines a product of matrices, the entries in li
---+ determine which matrix from lm is chosen. Finally we sum over all
---+ products. The result "sm" can be used to call split with (e.g.)
---+ second parameter "first nullSpace sm"

if R has EuclideanDomain then -- using rowEchelon

\begin{verbatim}
cyclicSubmodule : (L M R, V R) -> V V R
  ++ cyclicSubmodule(lm,v) generates a basis as follows.
  ++ It is assumed that the size n of the vector equals the number
  ++ of rows and columns of the matrices. Then the matrices generate
  ++ a subalgebra, say \spad{A}, of the algebra of all square matrices of
  ++ dimension n. V R is an \spad{A}-module in the natural way.
  ++ cyclicSubmodule(lm,v) generates the R-Basis of Av as
  ++ described in section 6 of R. A. Parker's "The Meat-Axe".
  ++ Note that in contrast to the description in "The Meat-Axe" and to
  ++ standardBasisOfCyclicSubmodule the result is in echelon form.

standardBasisOfCyclicSubmodule : (L M R, V R) -> M R
  ++ standardBasisOfCyclicSubmodule(lm,v) returns a matrix as follows.
  ++ It is assumed that the size n of the vector equals the number
  ++ of rows and columns of the matrices. Then the matrices generate
  ++ a subalgebra, say \spad{A},
  ++ of the algebra of all square matrices of
  ++ dimension n. V R is an \spad{A}-module in the natural way.
  ++ standardBasisOfCyclicSubmodule(lm,v) calculates a matrix whose
  ++ non-zero column vectors are the R-Basis of Av achieved
  ++ in the way as described in section 6
  ++ of R. A. Parker's "The Meat-Axe".
  ++ Note that in contrast to cyclicSubmodule, the result is not
  ++ in echelon form.
\end{verbatim}

if R has Field then -- only because of inverse in SM

\begin{verbatim}
areEquivalent? : (L M R, L M R, B, I) -> M R
  ++ areEquivalent?(aG0,aG1,randomelements,numberOfTries) tests
  ++ whether the two lists of matrices, all assumed of same
  ++ square shape, can be simultaneously conjugated by a non-singular
  ++ matrix. If these matrices represent the same group generators,
  ++ the representations are equivalent.
  ++ The algorithm tries
  ++ numberOfTries times to create elements in the
  ++ generated algebras in the same fashion. If their ranks differ,
  ++ they are not equivalent. If an
  ++ isomorphism is assumed, then
  ++ the kernel of an element of the first algebra
  ++ is mapped to the kernel of the corresponding element in the
  ++ second algebra. Now consider the one-dimensional ones.
  ++ If they generate the whole space (e.g. irreducibility !)
  ++ we use standardBasisOfCyclicSubmodule to create the
  ++ only possible transition matrix. The method checks whether the
  ++ matrix conjugates all corresponding matrices from aG1.
  ++ The way to choose the singular matrices is as in meatAxe.
  ++ If the two representations are equivalent, this routine
\end{verbatim}
++ returns the transformation matrix TM with
++ aG0.i * TM = TM * aG1.i for all i. If the representations
++ are not equivalent, a small 0-matrix is returned.
++ Note that the case
++ with different sets of group generators cannot be handled.
areEquivalent? : (L M R, L M R) -> M R
++ areEquivalent?(aG0,aG1) calls areEquivalent?(aG0,aG1,true,25).
++ Note that the choice of 25 was rather arbitrary.
areEquivalent? : (L M R, L M R, I) -> M R
++ areEquivalent?(aG0,aG1,numberOfTries) calls
++ areEquivalent?(aG0,aG1,true,25).
++ Note that the choice of 25 was rather arbitrary.
isAbsolutelyIrreducible? : (L M R, I) -> B
++ isAbsolutelyIrreducible?(aG, numberOfTries) uses
++ Norton’s irreducibility test to check for absolute
++ irreducibility, assuming if a one-dimensional kernel is found.
++ As no field extension changes create "new" elements
++ in a one-dimensional space, the criterium stays true
++ for every extension. The method looks for one-dimensionals only
++ by creating random elements (no fingerprints) since
++ a run of meatAxe would have proved absolute irreducibility
++ anyway.
isAbsolutelyIrreducible? : L M R -> B
++ isAbsolutelyIrreducible?(aG) calls
++ isAbsolutelyIrreducible?(aG,25).
++ Note that the choice of 25 was rather arbitrary.
split : (L M R, V R) -> L L M R
++ split(aG, vector) returns a subalgebra \spad{A} of all
++ square matrix of dimension n as a list of list of matrices,
++ generated by the list of matrices aG, where n denotes both
++ the size of vector as well as the dimension of each of the
++ square matrices.
++ V R is an A-module in the natural way.
++ split(aG, vector) then checks whether the cyclic submodule
++ generated by vector is a proper submodule of V R.
++ If successful, it returns a two-element list, which contains
++ first the list of the representations of the submodule,
++ then the list of the representations of the factor module.
++ If the vector generates the whole module, a one-element list
++ of the old representation is given.
++ Note that a later version this should call the other split.
split: (L M R, V V R) -> L L M R
++ split(aG,submodule) uses a proper submodule of R**n
++ to create the representations of the submodule and of
++ the factor module.
if (R has Finite) and (R has Field) then
meatAxe : (L M R, B, I, I) -> L L M R
++ meatAxe(aG,randomElements,numberOfTries, maxTests) returns
++ a 2-list of representations as follows.
++ All matrices of argument aG are assumed to be square
meatAxe : L M R \rightarrow L L M R
++ meatAxe(aG) calls meatAxe(aG,\text{false},25,7) returns
++ a 2-list of representations as follows.
++ All matrices of argument \text{\spad{aG}} are assumed to be square
++ and of
++ equal size. Then \text{\spad{aG}} generates a subalgebra,
++ say \text{\spad{A}}, of the algebra
++ of all square matrices of dimension \(n\). \(V R\) is an \(A\)-module
++ in the usual way.
++ meatAxe(aG) creates at most 25 random elements
++ of the algebra, tests
++ them for singularity. If singular, it tries at most 7
++ elements of its kernel to generate a proper submodule.
++ If successful a list which contains first the list of the
++ representations of the submodule, then a list of the
++ representations of the factor module is returned.
++ Otherwise, if we know that all the kernel is already
++ scanned, Norton’s irreducibility test can be used either
++ to prove irreducibility or to find the splitting.
++ Notes: the first 6 tries use Parker’s fingerprints.
++ Also, 7 covers the case of three-dimensional kernels over the field with 2 elements.

meatAxe : (L M R, \text{B}) \rightarrow L L M R
++ meatAxe(aG, \text{randomElements}) calls meatAxe(aG,\text{false},6,7),
++ only using Parker’s fingerprints, if \text{randomElements} is false.
++ If it is true, it calls meatAxe(aG,\text{true},25,7),
++ only using random elements.
++ Note that the choice of 25 was rather arbitrary.
++ Also, 7 covers the case of three-dimensional kernels over the field
++ with 2 elements.

meatAxe : (L M R, \text{PI}) \rightarrow L L M R
++ meatAxe(aG, \text{numberOfTries}) calls
++ meatAxe(aG,\text{true},\text{numberOfTries},7).
++ Notes: 7 covers the case of three-dimensional
++ kernels over the field with 2 elements.
scanOneDimSubspaces: (L V R, I) -> V R
++ scanOneDimSubspaces(basis,n) gives a canonical representative
++ of the n-th one-dimensional subspace of the vector space
++ generated by the elements of basis, all from R**n.
++ The coefficients of the representative are of shape
++ (0,...,0,1,*...,*), * in R. If the size of R
++ is q, then there are (q**n-1)/(q-1) of them.
++ We first reduce n modulo this number, then find the
++ largest i such that +/[q**i for i in 0..i-1] <= n.
++ Subtracting this sum of powers from n results in an
++ i-digit number to basis q. This fills the positions of the
++ stars.
-- would prefer to have (V V R,.... but nullSpace results
-- in L V R

private ==> add

-- import of domain and packages
import OutputForm

-- declarations and definitions of local variables and
-- local function

blockMultiply: (M R, M R, L I, I) -> M R
-- blockMultiply(a,b,li,n) assumes that a has n columns
-- and b has n rows, li is a sublist of the rows of a and
-- a sublist of the columns of b. The result is the
-- multiplication of the (li x n) part of a with the
-- (n x li) part of b. We need this, because just matrix
-- multiplying the parts would require extra storage.
blockMultiply(a, b, li, n) ==
matrix([[ +/[a(i,s) * b(s,j) for s in 1..n ]
for j in li ] for i in li])

fingerPrint: (NNI, M R, M R, M R) -> M R
-- is local, because one should know all the results for smaller i
fingerPrint (i : NNI, a : M R, b : M R, x : M R) ==
-- i > 2 only gives the correct result if the value of x from
-- the parameter list equals the result of fingerprint(i-1,...)
(i::PI) = 1 => x := a + b + a*b
(i::PI) = 2 => x := (x + a*b)*b
(i::PI) = 3 => x := a + b*x
(i::PI) = 4 => x := x + b
(i::PI) = 5 => x := x + a*b
(i::PI) = 6 => x := x - a + b*a
error "Sorry, but there are only 6 fingerprints!"
x
-- definition of exported functions

--randomWord(lli,lm) ==
-- -- we assume that all matrices are square of same size
-- numberOfMatrices := #lm
-- +[/*[lm.1+i rem numberOfMatrices) for i in li ] for li in lli]

completeEchelonBasis(basis) ==

dimensionOfSubmodule : NNI := #basis
n : NNI := # basis.1
indexOfVectorToBeScanned : NNI := 1
row : NNI := dimensionOfSubmodule

completedBasis : M R := zero(n, n)
for i in 1..dimensionOfSubmodule repeat
  completedBasis := setRow_!(completedBasis, i, basis.i)
if #basis <= n then
  newStart : NNI := 1
  for j in 1..n
    while indexOfVectorToBeScanned <= dimensionOfSubmodule repeat
      if basis.indexOfVectorToBeScanned.j = 0 then
        completedBasis(1+row,j) := 1 --put unit vector into basis
        row := row + 1
      else
        indexOfVectorToBeScanned := indexOfVectorToBeScanned + 1
        newStart : NNI := j + 1
      end if
    end while
  end for
completedBasis

createRandomElement(aG,algElt) ==
  numberOfGenerators : NNI := #aG
  -- randomIndex := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  algElt := algElt * aG.randomIndex
  -- randomIndxElement := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  algElt + aG.randomIndex

if R has EuclideanDomain then
  cyclicSubmodule (lm : L M R, v : V R) ==
    basis : M R := rowEchelon matrix list entries v
    -- normalizing the vector
    -- all these elements lie in the submodule generated by v
    furtherElts : L V R := [(lm.i*v)::V R for i in 1..maxIndex lm]
-- furtherElts has elements of the generated submodule. It will
-- will be checked whether they are in the span of the vectors
-- computed so far. Of course we stop if we have got the whole
-- space.
while (null furtherElts) and (nrows basis < #v) repeat
  w : V R := first furtherElts
  nextVector : M R := matrix list entries w -- normalizing the vector
  -- will the rank change if we add this nextVector
  -- to the basis so far computed?
  addedToBasis : M R := vertConcat(basis, nextVector)
  if rank addedToBasis ^= nrows basis then
    basis := rowEchelon addedToBasis -- add vector w to basis
    updateFurtherElts : L V R :=
      [(lm.i*w)::V R for i in 1..maxIndex lm]
    furtherElts := append (rest furtherElts, updateFurtherElts)
  else
    -- the vector w lies in the span of matrix, no updating
    -- of the basis
    furtherElts := rest furtherElts
  vector [row(basis, i) for i in 1..maxRowIndex basis]

standardBasisOfCyclicSubmodule (lm : L M R, v : V R) ==
  dim := #v
  standardBasis : L L R := list(entries v)
  basis : M R := rowEchelon matrix list entries v
  -- normalizing the vector
  -- all these elements lie in the submodule generated by v
  furtherElts : L V R := [(lm.i*v)::V R for i in 1..maxIndex lm]
  -- furtherElts has elements of the generated submodule. It will
  -- will be checked whether they are in the span of the vectors
  -- computed so far. Of course we stop if we have got the whole
  -- space.
  while (null furtherElts) and (nrows basis < #v) repeat
    w : V R := first furtherElts
    nextVector : M R := matrix list entries w -- normalizing the vector
    -- will the rank change if we add this nextVector
    -- to the basis so far computed?
    addedToBasis : M R := vertConcat(basis, nextVector)
    if rank addedToBasis ^= nrows basis then
      standardBasis := cons(entries w, standardBasis)
      basis := rowEchelon addedToBasis -- add vector w to basis
      updateFurtherElts : L V R :=
        [lm.i*w for i in 1..maxIndex lm]
      furtherElts := append (rest furtherElts, updateFurtherElts)
    else
      -- the vector w lies in the span of matrix, therefore
      -- no updating of matrix
      furtherElts := rest furtherElts
    vector [row(basis, i) for i in 1..maxRowIndex basis]
  transpose matrix standardBasis
if R has Field then -- only because of inverse in Matrix

-- as conditional local functions, *internal have to be here

splitInternal: (L M R, V R, B) -> L L M R

n : I := # vector -- R-rank of representation module =
  -- degree of representation
submodule : V V R := cyclicSubmodule (algebraGenerators,vector)
rankOfSubmodule : I := # submodule -- R-Rank of submodule
submoduleRepresentation : L M R := nil()
factormoduleRepresentation : L M R := nil()
if n ^= rankOfSubmodule then
  messagePrint " A proper cyclic submodule is found."
  if doSplitting? then -- no else !!
    submoduleIndices : L I := [i for i in 1..rankOfSubmodule]
factormoduleIndices : L I := [i for i in (1+rankOfSubmodule)..n]
  transitionMatrix : M R := _
  transpose completeEchelonBasis submodule
messagePrint " Transition matrix computed"
inverseTransitionMatrix : M R := _
  autoCoerce(inverse transitionMatrix)$Union(M R,"failed")
messagePrint " The inverse of the transition matrix computed"
messagePrint " Now transform the matrices"
for i in 1..maxIndex algebraGenerators repeat
  helpMatrix : M R := inverseTransitionMatrix * algebraGenerators.i
  -- in order to not create extra space and regarding the fact
  -- that we only want the two blocks in the main diagonal we
  -- multiply with the aid of the local function blockMultiply
  submoduleRepresentation := cons( blockMultiply( _
    helpMatrix,transitionMatrix,submoduleIndices,n), _
    submoduleRepresentation)
factormoduleRepresentation := cons( blockMultiply( _
    helpMatrix,transitionMatrix,factormoduleIndices,n), _
    factormoduleRepresentation)
[reverse submoduleRepresentation, reverse _
  factormoduleRepresentation]
else -- representation is irreducible
  messagePrint " The generated cyclic submodule was not proper"
[algebraGenerators]

irreducibilityTestInternal: (L M R, M R, B) -> L L M R
irreducibilityTestInternal(algebraGenerators, _
  singularMatrix,split?) ==

algebraGeneratorsTranspose : L M R := [transpose _


algebraGenerators.j for j in 1..maxIndex algebraGenerators]
x : M R := transpose singularMatrix
messagePrint " We know that all the cyclic submodules generated by all"
messagePrint " non-trivial element of the singular matrix under view are"
messagePrint " not proper, hence Norton's irreducibility test can be done:
-- actually we only would need one (!) non-trivial element from
-- the kernel of xt, such an element must exist as the transpose
-- of a singular matrix is of course singular. Question: Can
-- we get it more easily from the kernel of x = singularMatrix?
kernel : L V R := nullSpace xt
result : L L M R := _
    splitInternal(algebraGeneratorsTranspose,first kernel,split?)
if null rest result then -- this means first kernel generates
    -- the whole module
if 1 = #kernel then
      messagePrint " Representation is absolutely irreducible"
else
      messagePrint " Representation is irreducible, but we don't know "
      messagePrint " whether it is absolutely irreducible"
else
    if split? then
      messagePrint " Representation is not irreducible and it will be split:"
      -- these are the dual representations, so calculate the
      -- dual to get the desired result, i.e. "transpose inverse"
      -- improvements??
      for i in 1..maxIndex result repeat
        for j in 1..maxIndex (result.i) repeat
          mat : M R := result.i.j
          result.i.j := _
            transpose autoCoerce(inverse mat)$Union(M R,"failed")
        else
          messagePrint " Representation is not irreducible, use meatAxe to split"
          -- if "split?" then dual representation interchange factor
          -- and submodules, hence reverse
          reverse result

areEquivalent? (aG0, aG1) ==
    areEquivalent? (aG0, aG1, true, 25)

areEquivalent? (aG0, aG1, numberOfTries) ==
    areEquivalent? (aG0, aG1, true, numberOfTries)

areEquivalent? (aG0, aG1, randomelements, numberOfTries) ==
result : B := false
transitionM : M R := zero(1, 1)
numberOfGenerators : NNI := #aG0
-- need a start value for creating random matrices:
-- if we switch to randomelements later, we take the last
-- fingerprint.
if randomelements then  -- random should not be from I
  --randomIndex : I := randnum numberOfGenerators
  randomIndex := 1+(random()$Integer rem numberOfGenerators)
  x0 : M R := aG0.randomIndex
  x1 : M R := aG1.randomIndex
n : NNI := #row(x0,1)  -- degree of representation
foundResult := false
for i in 1..numberOfTries until foundResult repeat
  -- try to create a non-singular element of the algebra
  -- generated by "aG". If only two generators,
  -- i < 7 and not "randomelements" use Parker's fingerprints
  -- i >= 7 create random elements recursively:
  -- x_i+1 := x_i * mr1 + mr2, where mr1 and mr2 are randomly
  -- chosen elements form "aG".
  if i = 7 then randomelements := true
  if randomelements then
    --randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
    x0 := x0 * aG0.randomIndex
    x1 := x1 * aG1.randomIndex
  else
    x0 := fingerprint (i, aG0.0, aG0.1 ,x0)
    x1 := fingerprint (i, aG1.0, aG1.1 ,x1)
  -- test singularity of x0 and x1
  rk0 : NNI := rank x0
  rk1 : NNI := rank x1
  rk0 ^= rk1 =>
    messagePrint "Dimensions of kernels differ"
    foundResult := true
    result := false
  -- can assume dimensions are equal
  rk0 ^= n - 1 =>
    -- not of any use here if kernel not one-dimensional
    if randomelements then
      messagePrint "Random element in generated algebra does"
      messagePrint " not have a one-dimensional kernel"
    else
      messagePrint "Fingerprint element in generated algebra does"
      messagePrint " not have a one-dimensional kernel"
  -- can assume dimensions are equal and equal to n-1
if randomized elements then
  messagePrint "Random element in generated algebra has"
  messagePrint "one-dimensional kernel"
else
  messagePrint "Fingerprint element in generated algebra has"
  messagePrint "one-dimensional kernel"
kernel0 : L V R := nullSpace x0
kernel1 : L V R := nullSpace x1
baseChange0 : M R := standardBasisOfCyclicSubmodule(_
aG0,kernel0.1)
baseChange1 : M R := standardBasisOfCyclicSubmodule(_
aG1,kernel1.1)
(ncols baseChange0) ^= (ncols baseChange1) =>
  messagePrint "Dimensions of generated cyclic submodules differ"
  foundResult := true
  result := false
-- can assume that dimensions of cyclic submodules are equal
(ncols baseChange0) = n => -- full dimension
  transitionM := baseChange0 * _
    autoCoerce(inverse baseChange1)$Union(M R,"failed")
  foundResult := true
  result := true
for j in 1..numberOfGenerators while result repeat
  if (aG0.j*transitionM) ^= (transitionM*aG1.j) then
    result := false
    transitionM := zero(1,1)
    messagePrint "There is no isomorphism, as the only possible one"
    messagePrint "fails to do the necessary base change"
-- can assume that dimensions of cyclic submodules are not "n"
  messagePrint "Generated cyclic submodules have equal, but not full"
  messagePrint "dimension, hence we can not draw any conclusion"
-- here ends the for-loop
if not foundResult then
  messagePrint ""
  messagePrint "Can neither prove equivalence nor inequivalence."
  messagePrint "Try again."
else
  if result then
    messagePrint ""
    messagePrint "Representations are equivalent."
  else
    messagePrint ""
    messagePrint "Representations are not equivalent."
  transitionM

isAbsolutelyIrreducible?(aG) == isAbsolutelyIrreducible?(aG,25)

isAbsolutelyIrreducible?(aG, numberOfTries) ==
result : B := false

numberOfGenerators : NNI := #aG

-- need a start value for creating random matrices:
-- randomIndex : I := randnum numberOfGenerators
randomIndex := 1+(random()$Integer rem numberOfGenerators)

x : M R := aG.randomIndex

n : NNI := #row(x,1) -- degree of representation

foundResult : B := false

for i in 1..numberOfTries until foundResult repeat

-- try to create a non-singular element of the algebra
-- generated by "aG", dimension of its kernel being 1.
-- create random elements recursively:
-- x_i+1 := x_i * mr1 + mr2, where mr1 and mr2 are randomly
-- chosen elements form "aG".
-- randomIndex := randnum numberOfGenerators
randomIndex := 1+(random()$Integer rem numberOfGenerators)

x := x * aG.randomIndex

x := x + aG.randomIndex

-- test whether rank of x is n-1
rk : NNI := rank x

if rk = n - 1 then

foundResult := true

messagePrint "Random element in generated algebra has"

messagePrint " one-dimensional kernel"

kernel : L V R := nullSpace x

if n=#cyclicSubmodule(aG, first kernel) then

result := (irreducibilityTestInternal(aG,x,false)).1 ^= nil()$(L M R)

-- result := not null? first irreducibilityTestInternal(aG,x,false) -- this don't

else -- we found a proper submodule

result := false

--split(aG,kernel.1) -- to get the splitting

else -- not of any use here if kernel not one-dimensional

messagePrint "Random element in generated algebra does"

messagePrint " not have a one-dimensional kernel"

-- here ends the for-loop

if not foundResult then

messagePrint "We have not found a one-dimensional kernel so far,"

messagePrint " as we do a random search you could try again"

else

-- if not result then

-- messagePrint "Representation is not irreducible."

-- else

-- messagePrint "Representation is irreducible."

result

--split(algebraGenerators: L M R, vector: V R) ==
splitInternal(algebraGenerators, vector, true)

split(algebraGenerators : L M R, submodule: V V R) == -- not zero submodule
n : NNI := #submodule.1 -- R-rank of representation module =
   -- degree of representation
rankOfSubmodule : I := (#submodule) :: I -- R-Rank of submodule
submoduleRepresentation : L M R := nil()
factormoduleRepresentation : L M R := nil()
submoduleIndices : L I := [i for i in 1..rankOfSubmodule]
factormoduleIndices : L I := [i for i in (1+rankOfSubmodule)..(n::I)]
transitionMatrix : M R := _
   transpose completeEchelonBasis submodule
messagePrint " Transition matrix computed"
inverseTransitionMatrix : M R :=
   autoCoerce(inverse transitionMatrix)$Union(M R,"failed")
messagePrint " The inverse of the transition matrix computed"
messagePrint " Now transform the matrices"
for i in 1..maxIndex algebraGenerators repeat
   helpMatrix : M R := inverseTransitionMatrix * algebraGenerators.i
   -- in order to not create extra space and regarding the fact
   -- that we only want the two blocks in the main diagonal we
   -- multiply with the aid of the local function blockMultiply
   submoduleRepresentation := cons( blockMultiply(_, helpMatrix,transitionMatrix,submoduleIndices,n), _
   submoduleRepresentation)
   factormoduleRepresentation := cons( blockMultiply(_, helpMatrix,transitionMatrix,factormoduleIndices,n), _
   factormoduleRepresentation)
cons(reverse submoduleRepresentation, list( reverse _
   factormoduleRepresentation)::(L L M R))

-- the following is "under" "if R has Field", as there are compiler
-- problems with conditionally defined local functions, i.e. it
-- doesn't know, that "FiniteField" has "Field".

-- we are scanning through the vectorspaces
if (R has Finite) and (R has Field) then

meatAxe(algebraGenerators, randomelements, numberOfTries, _
   maxTests) ==
numberOfGenerators : NNI := #algebraGenerators
result : L L M R := nil()$(L L M R)
q : PI := size()$R:PI
-- need a start value for creating random matrices:
-- if we switch to randomelements later, we take the last
-- fingerprint.
if randomelements then -- random should not be from I

randomIndex : I := randnum numberOfGenerators
randomIndex := 1+(random()$Integer rem numberOfGenerators)
x : M R := algebraGenerators.randomIndex

foundResult : B := false
for i in 1..numberOfTries until foundResult repeat
  -- try to create a non-singular element of the algebra
  -- generated by "algebraGenerators". If only two generators,
  -- i < 7 and not "randomelements" use Parker's fingerprints
  -- i >= 7 create random elements recursively:
  -- x_i+1 := x_i * mr1 + mr2, where mr1 and mr2 are randomly
  -- chosen elements form "algebraGenerators".
  if i = 7 then randomelements := true
  if randomelements then
    randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
x := x * algebraGenerators.randomIndex
    randomIndex := randnum numberOfGenerators
    randomIndex := 1+(random()$Integer rem numberOfGenerators)
x := x + algebraGenerators.randomIndex
  else
    x := fingerPrint (i, algebraGenerators.1,_,
    algebraGenerators.2 , x)
  -- test singularity of x
  n : NNI := #row(x, 1) -- degree of representation
  if (rank x) ^= n then -- x singular
    if randomelements then
      messagePrint "Random element in generated algebra is singular"
    else
      messagePrint "Fingerprint element in generated algebra is singular"
    kernel : L V R := nullSpace x
      -- the first number is the maximal number of one dimensional
      -- subspaces of the kernel, the second is a user given
      -- constant
    numberOfOneDimSubspacesInKernel : I := (q**(#kernel)-1)quo(q-1)
    numberOfTests : I := min(numberOfOneDimSubspacesInKernel, maxTests)
    for j in 1..numberOfTests repeat
      -- we create an element in the kernel, there is a good
      -- probability for it to generate a proper submodule, the
      -- called "split" does the further work:
      result := _
      split(algebraGenerators,scanOneDimSubspaces(kernel,j))
      -- we had "not null rest result" directly in the following
      -- if .. then, but the statement there foundResult := true
      -- didn't work properly
      foundResult := not null rest result
      if foundResult then
        leave -- inner for-loop
        -- finish here with result
      else -- no proper submodule
-- we were not successful, i.e. gen. submodule was
-- not proper, if the whole kernel is already scanned,
-- Norton's irreducibility test is used now.
if (j+1)>numberOfOneDimSubspacesInKernel then
  -- we know that all the cyclic submodules generated
  -- by all non-trivial elements of the kernel are proper.
  foundResult := true
  result : L L M R := irreducibilityTestInternal (_
    algebraGenerators,x,true)
  leave -- inner for-loop
-- here ends the inner for-loop
else -- x non-singular
  if randomelements then
    messagePrint "Random element in generated algebra is non-singular"
  else
    messagePrint "Fingerprint element in generated algebra is non-singular"
-- here ends the outer for-loop
if not foundResult then
  result : L L M R := [nil()$(L M R), nil()$(L M R)]
  messagePrint ""
  messagePrint "Sorry, no result, try meatAxe(...,true)"
  messagePrint " or consider using an extension field."
  result

meatAxe (algebraGenerators) ==
  meatAxe(algebraGenerators, false, 25, 7)

meatAxe (algebraGenerators, randomElements?) ==
  randomElements? => meatAxe (algebraGenerators, true, 25, 7)
  meatAxe(algebraGenerators, false, 6, 7)

meatAxe (algebraGenerators:L M R, numberOfTries:PI) ==
  meatAxe (algebraGenerators, true, numberOfTries, 7)

scanOneDimSubspaces(basis,n) ==
  -- "dimension" of subspace generated by "basis"
  dim : NNI := #basis
  -- "dimension of the whole space:
  nn : NNI := #(basis.1)
  q : NNI := size()$R
  -- number of all one-dimensional subspaces:
  nred : I := n rem ((q**dim -1) quo (q-1))
  pos : I := nred
  i : I := 0
  for i in 0..dim-1 while nred >= 0 repeat
pos := nred
nred := nred - (q**i)
i := if i = 0 then 0 else i-1
coefficients : V R := new(dim,0$R)
coefficients.(dim-i) := 1$R
iR : L I := wholeRagits(pos::RADIX q)
for j in 1..(maxIndex iR) repeat
  coefficients.(dim-((#iR)::I) +j) := index((iR.j+(q::I))::PI)$R
result : V R := new(nn,0)
for i in 1..maxIndex coefficients repeat
  newAdd : V R := coefficients.i * basis.i
  for j in 1..nn repeat
    result.j := result.j + newAdd.j
result

——

— REP2.dotabb —

"REP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=REP2"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"REP2" -> "IVECTOR"

——

package RESLATC ResolveLatticeCompletion

— ResolveLatticeCompletion.input —

)set break resume
)sys rm -f ResolveLatticeCompletion.output
)spool ResolveLatticeCompletion.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show ResolveLatticeCompletion
--R
--R ResolveLatticeCompletion(S: Type) is a package constructor
--R Abbreviation for ResolveLatticeCompletion is RESLATC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RESLATC
--R
---R------------------------------------------------- Operations -------------------------------
---R coerce : S -> Void coerce : Exit -> S
---E 1

)spool
)lisp (bye)

——
— ResolveLatticeCompletion.help —
——

ResolveLatticeCompletion examples

This package provides coercions for the special types Exit and Void.

See Also:
o )show ResolveLatticeCompletion

——

ResolveLatticeCompletion (RESLATC)

Exports:
coerce

—— package RESLATC ResolveLatticeCompletion ——

)abbrev package RESLATC ResolveLatticeCompletion
++ Author: Stephen M. Watt
++ Date Created: 1986
++ Date Last Updated: May 30, 1991
++ Description:
++ This package provides coercions for the special types \spadtype{Exit}
++ and \spadtype{Void}.

ResolveLatticeCompletion(S: Type): with
   coerce: S -> Void
     + This coercion throws all information about s away.
     + This coercion allows values of any type to appear
     + in contexts where they will not be used.
     + For example, it allows the resolution of different types in
       + the \spad{then} and \spad{else} branches when an \spad{if}
       + is in a context where the resulting value is not used.
   coerce: Exit -> S
     + This coercion is
     + used for formal type correctness when a function will not
     + return directly to its caller.

== add
   coerce(s: S): Void == void()
   coerce(e: Exit): S ==
     error "Bug: Should not be able to obtain value of type Exit"

package RETSOL RetractSolvePackage

— RetractSolvePackage.input —

)set break resume
)sys rm -f RetractSolvePackage.output
)spool RetractSolvePackage.output
)set message test on
)set message auto off
)clear all
RetractSolvePackage (RETSOL)

RetractSolvePackage is an interface to SystemSolvePackage that attempts to retract the coefficients of the equations before solving.

See Also:
o )show RetractSolvePackage
Exports:
solveRetract

— package RETSOL RetractSolvePackage —

)abbrev package RETSOL RetractSolvePackage
++ Author: Manuel Bronstein
++ Date Created: 31 October 1991
++ Date Last Updated: 31 October 1991
++ Description:
++ RetractSolvePackage is an interface to \spadtype{SystemSolvePackage}
++ that attempts to retract the coefficients of the equations before
++ solving.

RetractSolvePackage(Q, R): Exports == Implementation where
  Q: IntegralDomain
  R: Join(IntegralDomain, RetractableTo Q)

PQ ==> Polynomial Q
FQ ==> Fraction PQ
SY ==> Symbol
P ==> Polynomial R
F ==> Fraction P
EQ ==> Equation
SSP ==> SystemSolvePackage

Exports ==> with
  solveRetract: (List P, List SY) -> List List EQ F
  + solveRetract(lp, lv) finds the solutions of the list lp of
  + rational functions with respect to the list of symbols lv.
  + The function tries to retract all the coefficients of the equations
  + to Q before solving if possible.

Implementation ==> add
LEQQ2F : List EQ FQ -> List EQ F
FQ2F : FQ -> F
PQ2P : PQ -> P
QIfCan : List P -> Union(List FQ, "failed")
PQIfCan: P -> Union(FQ, "failed")

PQ2P p == map((q:Q):R +-> q::R)$PolynomialFunctions2(Q, R)
FQ2F f == PQ2P numer f / PQ2P denom f
LEQQ2F l == [equation(FQ2F lhs eq, FQ2F rhs eq) for eq in l]

solveRetract(lp, lv) ==
  (u := QIfCan lp) case "failed" =>
    solve((p::F for p in lp)$List(F), lv)$SSP(R)
  [LEQQ2F l for l in solve(u::List(FQ), lv)$SSP(Q)]
package RFP RootsFindingPackage

-- RootsFindingPackage.input --

)set break resume
)sys rm -f RootsFindingPackage.output
)spool RootsFindingPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show RootsFindingPackage
--R
--R RootsFindingPackage(K: Field) is a package constructor
--R Abbreviation for RootsFindingPackage is RFP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RFP
--R
--R----------------------------------------------- Operations -------------------------------
--R foundZeroes : () -> List(K) setFoundZeroes : List(K) -> List(K)
--R distinguishedCommonRootsOf : (List(SparseUnivariatePolynomial(K)),K) -> Record(zeros: List(K),extDegree: Integer)
--R distinguishedRootsOf : (SparseUnivariatePolynomial(K),K) -> Record(zeros: List(K),extDegree: Integer)
--E 1

)spool
)lisp (bye)

——

— RootsFindingPackage.help —

====================================================================
RootsFindingPackage examples
====================================================================

This package finds all the roots of a polynomial. If the constant field is not large enough then it returns the list of found zeros and the degree of the extension need to find the other roots missing. If the return degree is 1 then all the roots have been found. If 0 is return for the extension degree then there are an infinite number of zeros, that is you ask for the zeroes of 0. In the case of infinite field a list of all found zeros is kept and for each other call of a function that finds zeroes, a check is made on that list; this is to keep a kind of "canonical" representation of the elements.

See Also:
  o )show RootsFindingPackage

——
RootsFindingPackage (RFP)

Exports:
distinguishedCommonRootsOf distinguishedRootsOf foundZeros setFoundZeros

— package RFP RootsFindingPackage —

)abbrev package RFP RootsFindingPackage
++ Authors: G. Hache
++ Date Created: 6 Oct 1994
++ Date Last Updated: May 2010 by Tim Daly
++ Description:
++ This pacackage finds all the roots of a polynomial. If the constant field is
++ not large enough then it returns the list of found zeros and the degree
++ of the extension need to find the other roots missing. If the return
++ degree is 1 then all the roots have been found. If 0 is return
++ for the extension degree then there are an infinite number of zeros,
++ that is you ask for the zeroes of 0. In the case of infinite field
++ a list of all found zeros is kept and for each other call of a function
++ that finds zeroes, a check is made on that list; this is to keep
++ a kind of "canonical" representation of the elements.
RootsFindingPackage(K):P==T where
K:Field

LIST ==> List
INT ==> Integer
NNI ==> NonNegativeInteger
MFINFACF ==> MultFiniteFactorize
FFFACTSE ==> FiniteFieldFactorizationWithSizeParseBySideEffect
SUP ==> SparseUnivariatePolynomial
REC ==> Record(zeros:List(K),extDegree:INT)

P== with

distinguishedRootsOf: (SUP(K),K) -> REC
++ distinguishedRootsOf returns a record consisting of a list of zeros
++ of the input polynomial followed by the smallest extension degree
++ needed to find all the zeros. If K has
++ \spad{PseudoAlgebraicClosureOfFiniteFieldCategory} or
++ \spad{PseudoAlgebraicClosureOfRationalNumberCategory} then
++ a root is created for each irreducible factor, and only these
++ roots are returns and not their conjugate.

distinguishedCommonRootsOf: (List SUP(K),K) -> REC
++ distinguishedCommonRootsOf returns the common zeros of a list of
++ polynomial. It returns a record as in distinguishedRootsOf. If 0
++ is returned as extension degree then there are an infinite number
++ of common zeros (in this case, the polynomial 0 was given in the
++ list of input polynomials).

foundZeroes: () -> List K
++ foundZeroes returns the list of already
++ found zeros by the functions
++ distinguishedRootsOf and
++ distinguishedCommonRootsOf.

setFoundZeroes: List K -> List K
++ setFoundZeroes sets the list of foundZeroes to the given one.

T== add
-- signature of local function
zeroOfLinearPoly: SUP(K) -> K
-- local variable
listOfAllZeros:List(K):=empty()

foundZeroes:=listOfAllZeros

if K has PseudoAlgebraicClosureOfPerfectFieldCategory then
  distinguishedRootsOf(polyZero, theExtension) ==
  --PRECONDITION: setExtension! is called in K to set the extension to
  --the extension of factorization
  zero?(polyZero) =>
  [empty(),0]
  listOfZeros:List(K):=distinguishedRootsOf(polyZero,theExtension)$K
  [listOfZeros,1]

if K has FiniteFieldCategory and _
  ^(K has PseudoAlgebraicClosureOfFiniteFieldCategory) then
  distinguishedRootsOf(polyZero,dummy)==
  zero?(polyZero) => [empty(),0]
  factorpolyZero:=factor(polyZero)$FFFFACTSE(K,SUP(K))
  listOfFactor:=factorList(factorpolyZero)
  listFact:= [pol.fctr for pol in listOfFactor | one?(degree(pol))]
  degExt:INT:=
  lcm([degree(poly) for poly in listFact]) pretend LIST(INT))
  listOfZeros:List(K):=removeDuplicates_
  [zeroOfLinearPoly(poly) for poly in listFact | one?(degree(poly))]
if K has QuotientFieldCategory(Integer) and _
(K has PseudoAlgebraicClosureOfRationalNumberCategory) then
  distinguishedRootsOf(polyZero,dummy)==
  zero?(polyZero) => [empty(),0]
  factorpolyZero:=factor(polyZero)$RationalFactorize(SUP(K))
  listOfFactor:=factorList(factorpolyZero)
  listFact:=[pol.fctr for pol in listOfFactor]
  degExt:INT:=
    lcm([degree(poly) for poly in listFact]) pretend LIST(INT)
  listOfZeros:List(K):=
    [zeroOfLinearPoly(poly) for poly in listFact | one?(degree(poly))]
  [listOfZeros,degExt]

distinguishedCommonRootsOf(listOfPoly1,theExtension)==
  listOfPoly:List(SUP(K)):=
    [pol for pol in listOfPoly1 | ~zero?(pol)]
  empty?(listOfPoly) ==> [empty(),0]
  reco:= distinguishedRootsOf(gcd(listOfPoly),theExtension)
  listOfZeros:= reco.zeros
  degExt:INT:= reco.extDegree
  [listOfZeros,degExt]

zeroOfLinearPoly(pol)==
  ~one?(degree(pol)) => error "the polynomial is not linear"
  listCoef:List(K):=coefficients(pol)
  one?(#listCoef) => 0
  - last(listCoef) / first(listCoef)

setFoundZeros(setlist)==
  oldListOfAllZeroes:= copy listOfAllZeros
  listOfAllZeros:=setlist
  oldListOfAllZeros
Chapter 20

Chapter S

package SAERFFC SAERationalFunctionAlgFactor

— SAERationalFunctionAlgFactor.input —

)set break resume
)sys rm -f SAERationalFunctionAlgFactor.output
)spool SAERationalFunctionAlgFactor.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SAERationalFunctionAlgFactor
--R
--R SAERationalFunctionAlgFactor(UP: UnivariatePolynomialCategory(Fraction(Polynomial(Integer))),SAE: Join(Field,CharacteristicZero,MonogenicAlgebra(Fraction(Polynomial(Integer)),UP)),UPA: UnivariatePolynomialCategory(SAE)) is a package constructor
--R Abbreviation for SAERationalFunctionAlgFactor is SAERFFC
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SAERFFC
--R
--R----------------------------------------------- Operations -----------------------------------------------
--R factor : UPA -> Factored(UPA)
--R
--E 1

)spool
)lisp (bye)

——

— SAERationalFunctionAlgFactor.help —

3785
Factorization of univariate polynomials with coefficients in an algebraic extension of \texttt{Fraction Polynomial Integer}.

See Also:
- \texttt{)show SAERationalFunctionAlgFactor}

---

SAERationalFunctionAlgFactor (SAERFFC)

RFFACT

⇓

ALIST

Exports:

- \texttt{factor}

---

\texttt{)abbrev package SAERFFC SAERationalFunctionAlgFactor}

++ Author: Patrizia Gianni
++ Description:
++ Factorization of univariate polynomials with coefficients in an algebraic extension of \texttt{Fraction Polynomial Integer}.

SAERationalFunctionAlgFactor(UP, SAE, UPA): Exports == Implementation where

\texttt{UP} : UnivariatePolynomialCategory Fraction Polynomial Integer
\texttt{SAE} : Join(Field, CharacteristicZero,
          MonogenicAlgebra(Fraction Polynomial Integer, UP))
\texttt{UPA} : UnivariatePolynomialCategory SAE

Exports ==> with
  \texttt{factor}: UPA \rightarrow \texttt{Factored UPA}
++ factor(p) returns a prime factorisation of p.

Implementation ==> add

factor(q) ==
factor(q, factor$RationalFunctionFactor(UP)
$$InnerAlgFactor(Fraction Polynomial Integer, UP, SAE, UPA)

———

— SAERFFC.dotabb —

"SAERFFC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SAERFFC"]
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]
"SAERFFC" -> "MONOGEN"

———

package FORMULA1 ScriptFormulaFormat1

— ScriptFormulaFormat1.input —

)set break resume
)sys rm -f ScriptFormulaFormat1.output
)spool ScriptFormulaFormat1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ScriptFormulaFormat1
--R
--R ScriptFormulaFormat1(S: SetCategory) is a package constructor
--R Abbreviation for ScriptFormulaFormat1 is FORMULA1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for FORMULA1
--R
--R----------------------------- Operations -----------------------------
--R coerce : S -> ScriptFormulaFormat
--R
--E 1

)spool
)lisp (bye)
ScriptFormulaFormat1 (FORMULA1)

Exports:
coerce

— package FORMULA1 ScriptFormulaFormat1 —

)abbrev package FORMULA1 ScriptFormulaFormat1
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1990
++ References:
++ SCRIPT Mathematical Formula Formatter User’s Guide, SH20-6453,
++ IBM Corporation, Publishing Systems Information Development,
++ Dept. G68, P.O. Box 1900, Boulder, Colorado, USA 80301-9191.
++ Description:
++ \spadtype{ScriptFormulaFormat1} provides a utility coercion for
++ changing to SCRIPT formula format anything that has a coercion to
++ the standard output format.

ScriptFormulaFormat1(S : SetCategory): public == private where
  public == with
    coerce: S -> ScriptFormulaFormat()
    coerce(s) provides a direct coercion from an expression s of domain S
    to SCRIPT formula format. This allows the user to skip the step of
    first manually coercing the object to standard output format
    before it is coerced to SCRIPT formula format.

  private == add
    import ScriptFormulaFormat()
    coerce(s : S): ScriptFormulaFormat ==
      coerce(s :: OutputForm)$ScriptFormulaFormat

— FORMULA1.dotabb —

"FORMULA1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=FORMULA1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"FORMULA1" -> "BASTYPE"
"FORMULA1" -> "KOERCE"

— SegmentBindingFunctions2.input —

)set break resume
)sys rm -f SegmentBindingFunctions2.output
)spool SegmentBindingFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SegmentBindingFunctions2
--R
--R SegmentBindingFunctions2(R: Type,S: Type) is a package constructor
--R Abbreviation for SegmentBindingFunctions2 is SEGBIND2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SEGBIND2
--R
--R---------------------------------------------------------- Operations --------------------------------
--R map : ((R -> S),SegmentBinding(R)) -> SegmentBinding(S)
--R
--E 1

)spool
)lisp (bye)

---

SegmentBindingFunctions2.help

====================================================================
SegmentBindingFunctions2 examples
====================================================================

This package provides operations for mapping functions onto SegmentBinding.

See Also:
  o )show SegmentBindingFunctions2

---

SegmentBindingFunctions2 (SEGBIND2)

Exports:
    map

    -- package SEGBIND2 SegmentBindingFunctions2 --
package SEG2 SegmentFunctions2

SegmentFunctions2(R: Type, S: Type): with
map: (R -> S, SegmentBinding R) -> SegmentBinding S
  ++ map(f,v=a..b) returns the value given by \spad{v=f(a)..f(b)}.
  ++
  == add
  map(f, b) ==
  equation(variable b, map(f, segment b)$SegmentFunctions2(R, S))
— SegmentFunctions2.help —

SegmentFunctions2 examples

This package provides operations for mapping functions onto segments.

See Also:
- )show SegmentFunctions2

SegmentFunctions2 (SEG2)

Exports:
- map

— package SEG2 SegmentFunctions2 —

)abbrev package SEG2 SegmentFunctions2
++ Date Last Updated: June 4, 1991
++ Description:
This package provides operations for mapping functions onto segments.

SegmentFunctions2(R:Type, S:Type): public == private where
  public ==> with
    map: (R -> S, Segment R) -> Segment S
    ++ map(f,l..h) returns a new segment \spad{f(l)..f(h)}.
  if R has OrderedRing then
    map: (R -> S, Segment R) -> List S
    ++ map(f,s) expands the segment s, applying \spad{f} to each
    ++ value. For example, if \spad{s = l..h by k}, then the list
    ++ \spad{\{f(l), f(l+k),..., f(lN)\}} is computed, where
    ++ \spad{1N <= h < 1N+k}.

private ==> add
  map(f : R->S, r : Segment R): Segment S ==
    SEGMENT(f lo r,f hi r)$Segment(S)
  if R has OrderedRing then
    map(f : R->S, r : Segment R): List S ==
      lr := nil()$List(S)
      l := lo r
      h := hi r
      inc := (incr r)::R
      if inc > 0 then
        while l <= h repeat
          lr := concat(f(l), lr)
          l := l + inc
      else
        while l >= h repeat
          lr := concat(f(l), lr)
          l := l + inc
      reverse_! lr

--- SEG2.dotabb ---

"SEG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SEG2"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"SEG2" -> "FLAGG"
package SAEFACT SimpleAlgebraicExtensionAlgFactor

--- SimpleAlgebraicExtensionAlgFactor.input ---

)set break resume
/sys rm -f SimpleAlgebraicExtensionAlgFactor.output
/spool SimpleAlgebraicExtensionAlgFactor.output
)set message test on
)set message auto off
)clear all

--) 1 of 1
)show SimpleAlgebraicExtensionAlgFactor
--)R
--)R SimpleAlgebraicExtensionAlgFactor(UP: UnivariatePolynomialCategory(Fraction(Integer)),SAE: Join(Field,CharacteristicZero,MonogenicAlgebra(Fraction(Integer),UP)) is a package constructor
--)R Abbreviation for SimpleAlgebraicExtensionAlgFactor is SAEFACT
--)R This constructor is exposed in this frame.
--)R Issue )edit bookvol10.4.pamphlet to see algebra source code for SAEFACT
--)R
--)----------------------------------- Operations -----------------------------------
--)R factor : UPA -> Factored(UPA)
--)R
--)E 1

)spool
)lisp (bye)

---

--- SimpleAlgebraicExtensionAlgFactor.help ---

====================================================================
SimpleAlgebraicExtensionAlgFactor examples
====================================================================

Factorization of univariate polynomials with coefficients in an algebraic extension of the rational numbers (Fraction Integer).

See Also:
  o )show SimpleAlgebraicExtensionAlgFactor

---
SimpleAlgebraicExtensionAlgFactor (SAEFAC)

Exports:

- factor

---

\( \text{factor} \) of univariate polynomials with coefficients in an
algebraic extension of the rational numbers (\spadtype{Fraction Integer}).

---

\( \text{factor} (p) \) returns a prime factorisation of \( p \).

Implementation ==

\[ \text{factor} (q) == \]
\[ \text{factor}(q, \text{factor}\$\text{RationalFactorize}(UP)) \]
\[ \text{$\text{InnerAlgFactor}(\text{Fraction Integer, UP, SAE, UPA})$} \]
package SIMPAN SimplifyAlgebraicNumberConvertPackage

— SimplifyAlgebraicNumberConvertPackage.input —

)set break resume
)sys rm -f SimplifyAlgebraicNumberConvertPackage.output
)spool SimplifyAlgebraicNumberConvertPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SimplifyAlgebraicNumberConvertPackage
--R
--R SimplifyAlgebraicNumberConvertPackage is a package constructor
--R Abbreviation for SimplifyAlgebraicNumberConvertPackage is SIMPAN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SIMPAN
--R
--R----------------------------------- Operations -----------------------------------
--R simplify : AlgebraicNumber -> Expression(Integer)
--R
--E 1

)spool
)lisp (bye)

— SimplifyAlgebraicNumberConvertPackage.help —

====================================================================
SimplifyAlgebraicNumberConvertPackage examples
====================================================================

Package to allow simplify to be called on AlgebraicNumbers by converting to EXPR(INT)

See Also:
  o )show SimplifyAlgebraicNumberConvertPackage
SimplifyAlgebraicNumberConvertPackage (SIMPAN)

Exports:
simplify

— package SIMPAN SimplifyAlgebraicNumberConvertPackage —

)abbrev package SIMPAN SimplifyAlgebraicNumberConvertPackage
++ Description:
++ Package to allow simplify to be called on AlgebraicNumbers
++ by converting to EXPR(INT)

SimplifyAlgebraicNumberConvertPackage(): with
  simplify: AlgebraicNumber -> Expression(Integer)
  ++ simplify(an) applies simplifications to an
== add
  simplify(a:AlgebraicNumber) ==
    simplify(a::Expression(Integer))$TranscendentalManipulations(Integer, Expression Integer)

— SIMPAN.dotabb —

"SIMPAN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SIMPAN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SIMPAN" -> "FS"
package SMITH SmithNormalForm

— SmithNormalForm.input —

)set break resume
)sys rm -f SmithNormalForm.output
)spool SmithNormalForm.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
)show SmithNormalForm
--R
--R SmithNormalForm(R: EuclideanDomain,Row: FiniteLinearAggregate(R),Col: FiniteLinearAggregate(R)) is a package constructor
--R Abbreviation for SmithNormalForm is SMITH
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SMITH
--R
--R------------------------------- Operations --------------------------------
--R hermite : M -> M  smith : M -> M
--R completeHermite : M -> Record(Hermite: M,eqMat: M)
--R completeSmith : M -> Record(Smith: M,leftEqMat: M,rightEqMat: M)
--R diophantineSystem : (M,Col) -> Record(particular: Union(Col,"failed"),basis: List(Col))
--R
--E 1

)spool
)lisp (bye)

— SmithNormalForm.help —

====================================================================
SmithNormalForm examples
====================================================================

SmithNormalForm is a package which provides some standard canonical forms for matrices.

See Also:
  o )show SmithNormalForm
SmithNormalForm (SMITH)

Exports:
  completeHermite completeSmith diophantineSystem hermite smith

— package SMITH SmithNormalForm —

)abbrev package SMITH SmithNormalForm
++ Author: Patrizia Gianni
++ Date Created: October 1992
++ Description:
++ \spadfunc{SmithNormalForm} is a package
++ which provides some standard canonical forms for matrices.

SmithNormalForm(R,Row,Col,M) : Exports == Implementation where

  R : EuclideanDomain
  Row : FiniteLinearAggregate R
  Col : FiniteLinearAggregate R
  M : MatrixCategory(R,Row,Col)

  I ==> Integer
  NNI ==> NonNegativeInteger
  HermiteForm ==> Record(Hermite:M,eqMat:M)
  SmithForm ==> Record(Smith : M, leftEqMat : M, rightEqMat : M)
  PartialV ==> Union(Col, "failed")
  Both ==> Record(particular: PartialV, basis: List Col)

Exports == with
  hermite : M -> M
    ++ \spad{hermite(m)} returns the Hermite normal form of the
    ++ matrix m.
  completeHermite : M -> HermiteForm
    ++ \spad{completeHermite} returns a record that contains
    ++ the Hermite normal form H of the matrix and the equivalence matrix
    ++ U such that U*m = H
  smith : M -> M
++ \spad{smith(m)} returns the Smith Normal form of the matrix m.

completeSmith : M -> SmithForm
  ++ \spad{completeSmith} returns a record that contains
  ++ the Smith normal form H of the matrix and the left and right
  ++ equivalence matrices U and V such that U*m*v = H

diophantineSystem : (M,Col) -> Both
  ++ \spad{diophantineSystem(A,B)} returns a particular integer solution and
  ++ an integer basis of the equation \spad{AX = B}.

Implementation == add
MATCAT1 == MatrixCategoryFunctions2(R,Row,Col,M,QF,Row2,Col2,M2)
MATCAT2 == MatrixCategoryFunctions2(QF,Row2,Col2,M2,R,Row,Col,M)
QF     ==> Fraction R
Row2   ==> Vector QF
Col2   ==> Vector QF
M2     ==> Matrix QF

----- Local Functions -----
elRow1  : (M,I,I) -> M
elRow2  : (M,R,I,I) -> M
elColumn2 : (M,R,I,I) -> M
isDiagonal? : M -> Boolean
ijDivide : (SmithForm ,I,I) -> SmithForm
lastStep : SmithForm -> SmithForm
test1  : (M,Col,NNI) -> Union(NNI , "failed")
test2  : (M, Col,NNI,NNI) -> Union( Col, "failed")

-- inconsistent system : case 0 = c --
test1(sm:M,b:Col,m1 : NNI) : Union(NNI , "failed") ==
  km:=m1
  while zero? sm(km,km) repeat
    if not zero?(b(km)) then return "failed"
    km:= (km - 1) :: NNI
  km

if Col has shallowlyMutable then
  test2(sm : M ,b : Col, n1:NNI,dk:NNI) : Union( Col, "failed") ==
    -- test divisibility --
    sol:Col := new(n1,0)
    for k in 1..dk repeat
      if (c:=(b(k) exquo sm(k,k))) case "failed" then return "failed"
      sol(k):= c::R
    sol

-- test if the matrix is diagonal or pseudo-diagonal --
isDiagonal?(m : M) : Boolean ==
  m1:= nrows m
  n1:= ncols m
  for i in 1..m1 repeat
for j in 1..n1 | (j ^= i) repeat
  if not zero?(m(i,j)) then return false
true

-- elementary operation of first kind: exchange two rows --
elRow1(m:M,i:I,j:I) : M ==
  vec:=row(m,i)
  setRow!(m,i,row(m,j))
  setRow!(m,j,vec)
  m

-- elementary operation of second kind: add to row i--
-- a*row j (i ^= j) --
elRow2(m : M,a:R,i:I,j:I) : M ==
  vec:= map(x +-> a*x,row(m,j))
  setRow!(m,i,vec)
  m

-- elementary operation of second kind: add to column i --
-- a*column j (i ^= j) --
elColumn2(m : M,a:R,i:I,j:I) : M ==
  vec:= map(x +-> a*x,column(m,j))
  setColumn!(m,i,vec)
  m

-- modify SmithForm in such a way that the term m(i,i) --
-- divides the term m(j,j). m is diagonal --
ijDivide(sf : SmithForm , i : I,j : I) : SmithForm ==
  m:=sf.Smith
  mii:=m(i,i)
  mjj:=m(j,j)
  extGcd:=extendedEuclidean(mii,mjj)
  d := extGcd.generator
  mii:=(mii exquo d)::R
  mjj := (mjj exquo d) :: R
  -- add to row j extGcd.coef1*row i --
  lMat:=elRow2(sf.leftEqMat,extGcd.coef1,j,i)
  -- switch rows i and j --
  lMat:=elRow1(lMat,i,j)
  -- add to row j -mii*row i --
  lMat := elRow2(lMat,-mii,j,i)
--  lMat := ijModify(mii,mjj,extGcd.coef1,extGcd.coef2,sf.leftEqMat,i,j)
  m(j,j):= m(i,i) * mjj
  m(i,i):= d
  -- add to column i extGcd.coef2 * column j --
  rMat := elColumn2(sf.rightEqMat,extGcd.coef2,i,j)
  -- add to column j -mjj*column i --
  rMat:=elColumn2(rMat,-mjj,j,i)
--  multiply by -1 column j --
setColumn!(rMat,j,\map(x \mapsto -1\times,\text{column}(rMat,j)))

\[[\text{m,1Mat,rMat}]\]

-- given a diagonal matrix compute its Smith form --

\text{lastStep}(sf : \text{SmithForm}) : \text{SmithForm} ==
\text{m:=sf.Smith}
\text{m1:=min(nrows m,ncols m)}
for \text{i in 1..m1 while (mii:=m(i,i)) ^=0 repeat}
  for \text{j in i+1..m1 repeat}
    if (m(j,j) exquo mii) case "failed" then return
    \text{lastStep(ijDivide(sf,i,j))}

sf

-- given m and t row-equivalent matrices, with t in upper triangular --

\text{findEqMat(m : M,t : M) : Record(Hermite : M, eqMat : M) ==}
\text{m1:=nrows m}
\text{n1:=ncols m}
"and"/[\text{zero? t(m1,j) for j in 1..n1}] => -- there are 0 rows
  if "and"/[\text{zero? t(1,j) for j in 1..n1}]
    then return [m,\text{scalarMatrix(m1,1)}] -- m is the zero matrix
\text{mm:=horizConcat(m,\text{scalarMatrix(m1,1)})}
\text{mmh:=rowEchelon mm}
[\text{subMatrix(mmh,1,m1,1,n1), subMatrix(mmh,1,m1,n1+1,n1+m1)}]
\text{u:M:=zero(m1,m1)}
\text{j:=1}
while \text{t(1,j)=0 repeat j:=j+1} -- there are 0 columns
\text{t1:=copy t}
\text{mm:=copy m}
if \text{j>1 then}
  \text{t1:=subMatrix(t,1,m1,j,n1)}
  \text{mm:=subMatrix(m,1,m1,j,n1)}
\text{t11:=t1(1,1)}
for \text{i in 1..m1 repeat}
  \text{u(i,1) := (mm(i,1) exquo t11) \map R}
for \text{j in 2..m1 repeat}
  \text{j0:=j}
  while \text{zero?(tjj:=t1(j,j0)) repeat j0:=j0+1}
  \text{u(i,j) := ((mm(i,j0)-(*\map{+}/[u(i,k)*t1(k,j0) for k in 1..(j-1)]])) exquo tjj)\map R}
\text{u1:M2:= \map(x \mapsto x::\text{QF},u)\map\text{MATCAT1}}
[t,\map(\text{retract}$\text{QF}$,(\text{inverse u1})\map\text{M2}\map\text{MATCAT2})]

--- Hermite normal form of m ---
\text{hermite(m:M) : M == rowEchelon m}

-- Hermite normal form and equivalence matrix --
\text{completeHermite(m : M) : Record(Hermite : M, eqMat : M) ==}
\text{findEqMat(m,rowEchelon m)
\input{section10.4}

\input{section10.2}

\begin{verbatim}
smith(m : M) : M == completeSmith(m).Smith

completeSmith(m : M) : Record(Smith : M, leftEqMat : M, rightEqMat : M) ==
cml:=completeHermite m
leftm:=cml.eqMat
m1:=cml.Hermite
isDiagonal? m1 => lastStep([m1,leftm,scalarMatrix(ncols m,1)])
nr:=nrows m
rightm:= transpose cml.eqMat
m1:=cml.Hermite
isDiagonal? m1 =>
cm2:=lastStep([m1,leftm,rightm])
nrows(m:=cm2.Smith) = nr => cm2
[transpose m,cm2.leftEqMat, cm2.rightEqMat]

cm2:=completeSmith m1
cm2:=lastStep([cm2.Smith,transpose(cm2.rightEqMat)*leftm,
rightm*transpose(cm2.leftEqMat)])
nrows(m:=cm2.Smith) = nr => cm2
[transpose m, cm2.leftEqMat, cm2.rightEqMat]

-- Find the solution in R of the linear system mX = b --
diophantineSystem(m : M, b : Col) : Both ==
sf:=completeSmith m
sm:=sf.Smith
m1:=nrows sm
lm:=sf.leftEqMat
b1:Col:= lm* b
(t1:=test1(sm,b1,m1)) case "failed" => ["failed",empty()]
dk:=t1 :: NNI
n1:=ncols sm
(t2:=test2(sm,b1,n1,dk)) case "failed" => ["failed",empty()]
rm := sf.rightEqMat
sol:=rm*(t2 :: Col) -- particular solution
dk = n1 => [sol,list new(n1,0)]
lsol:List Col := [column(rm,i) for i in (dk+1)..n1]
[sol,lsol]

----------

-- SMITH.dotabb --

"SMITH" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SMITH"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SMITH" -> "PFECAT"

----------
\end{verbatim}
package SCACHE SortedCache

--- SortedCache.input ---

)set break resume
)sys rm -f SortedCache.output
)spool SortedCache.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SortedCache
--R
--R SortedCache(S: CachableSet) is a package constructor
--R Abbreviation for SortedCache is SCACHE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SCACHE
--R
--R------------------------------- Operations --------------------------------
--R cache : () -> List(S)  clearCache : () -> Void
--R enterInCache : (S,(S -> Boolean)) -> S
--R enterInCache : (S,((S,S) -> Integer)) -> S
--R
--E 1

)spool
)lisp (bye)

--- SortedCache.help ---

====================================================================
SortedCache examples
====================================================================

A sorted cache of a cachable set S is a dynamic structure that keeps
the elements of S sorted and assigns an integer to each element of S
once it is in the cache. This way, equality and ordering on S are
tested directly on the integers associated with the elements of S,
once they have been entered in the cache.

See Also:
o )show SortedCache
SortedCache (SCACHE)

Exports:
cache clearCache enterInCache

— package SCACHE SortedCache —

)abbrev package SCACHE SortedCache
++ Author: Manuel Bronstein
++ Date Created: 31 Oct 1988
++ Date Last Updated: 14 May 1991
++ Description:
++ A sorted cache of a cachable set S is a dynamic structure that
++ keeps the elements of S sorted and assigns an integer to each
++ element of S once it is in the cache. This way, equality and ordering
++ on S are tested directly on the integers associated with the elements
++ of S, once they have been entered in the cache.

SortedCache(S:CachableSet): Exports == Implementation where
N ==> NonNegativeInteger
DIFF ==> 1024

Exports ==> with
  clearCache : () -> Void
      ++ clearCache() empties the cache.
  cache : () -> List S
      ++ cache() returns the current cache as a list.
  enterInCache: (S, S -> Boolean) -> S
      ++ enterInCache(x, f) enters x in the cache, calling \spad{f(y)} to
      ++ determine whether x is equal to y. It returns x with an integer
      ++ associated with it.
  enterInCache: (S, (S, S) -> Integer) -> S
      ++ enterInCache(x, f) enters x in the cache, calling \spad{f(x, y)} to
      ++ determine whether \spad{x < y (f(x,y) < 0), x = y (f(x,y) = 0)}, or
++ \spad{x > y (f(x,y) > 0)}. 
++ It returns \code{x} with an integer associated with it.

Implementation \impladd
\spad{shiftCache : (List S, N) -> Void
insertInCache: (List S, List S, S, N) -> S}

In the \code{cache} is defined:
\spad{cach := [nil()]$Record(cche:List S)}

\spad{cache() == cach.cche}

\spad{shiftCache(l, n) ==
for x in l repeat setPosition(x, n + position x)
void}

\spad{clearCache() ==
for x in cache repeat setPosition(x, 0)
cach.cche := nil()
void}

\spad{enterInCache(x:S, equal?:S -> Boolean) ==
scan := cache()
while not null scan repeat
  equal?(y := first scan) ==
    setPosition(x, position y)
    return y
  scan := rest scan
setPosition(x, 1 + #cache())
cach.cche := concat(cache(), x)
x}

\spad{enterInCache(x:S, triage:(S, S) -> Integer) ==
scan := cache()
pos:N:= 0
for i in 1..#scan repeat
  zero?(n := triage(x, y := first scan)) ==
    setPosition(x, position y)
    return y
  n<0 => return insertInCache(first(cache()),(i-1)::N),scan,x,pos
  scan := rest scan
  pos := position y
setPosition(x, pos + DIFF)
cach.cche := concat(cache(), x)
x}

\spad{insertInCache(before, after, x, pos) ==
if ((pos+1) = position first after) then shiftCache(after, DIFF)
setPosition(x, pos + ((position first after) - pos)::N quo 2))
cach.cche := concat(before, concat(x, after))
x}
package SORTPAK SortPackage

--- S 1 of 1 ---
)
show SortPackage
--R
--R SortPackage(S: Type,A)where
--R A: IndexedAggregate(Integer,S)with
--R finiteAggregate
--R shallowlyMutable is a package constructor
--R Abbreviation for SortPackage is SORTPAK
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SORTPAK
--R
--R---------------------------------- Operations ----------------------------------
--R bubbleSort!: A -> A if S has ORDSET
--R bubbleSort!: (A,((S,S) -> Boolean)) -> A
--R insertionSort!: (A,((S,S) -> Boolean)) -> A
--R insertionSort!: A -> A if S has ORDSET
--R
--E 1

)spool
)lisp (bye)
This package exports sorting algorithms

See Also:
  o `)show SortPackage`
++ bubbleSort!(a,f) \undocumented
insertionSort_!: (A, (S,S) -> Boolean) -> A
++ insertionSort!(a,f) \undocumented
if S has OrderedSet then
  bubbleSort_!: A -> A
  ++ bubbleSort!(a) \undocumented
  insertionSort_!: A -> A
  ++ insertionSort! \undocumented

Implementation == add
bubbleSort_!(m,f) ==
n := #m
for i in 1..(n-1) repeat
  for j in n..(i+1) by -1 repeat
    if f(m.j,m.(j-1)) then swap_!(m,j,j-1)
m
insertionSort_!(m,f) ==
for i in 2..#m repeat
  j := i
  while j > 1 and f(m.j,m.(j-1)) repeat
    swap_!(m,j,j-1)
    j := (j - 1) pretend PositiveInteger
  m
if S has OrderedSet then
  bubbleSort_!(m) == bubbleSort_!(m,_<$S)
insertionSort_!(m) == insertionSort_!(m,_<$S)
if A has UnaryRecursiveAggregate(S) then
  bubbleSort_!(m,fn) ==
    empty? m => m
    l := m
    while not empty? (r := l.rest) repeat
      r := bubbleSort_!(r,fn)
      x := l.first
      if fn(r.first,x) then
        l.first := r.first
        r.first := x
        l.rest := r
        l := l.rest
      m

——

— SORTPAK.dotabb —

"SORTPAK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SORTPAK"]
"IXAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=IXAGG"]
"EVALAB" [color="#4488FF",href="bookvol10.2.pdf#nameddest=EVALAB"]
"SORTPAK" -> "IXAGG"
package SUP2 SparseUnivariatePolynomialFunctions2

— SparseUnivariatePolynomialFunctions2.input —

)set break resume
)sys rm -f SparseUnivariatePolynomialFunctions2.output
)spool SparseUnivariatePolynomialFunctions2.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show SparseUnivariatePolynomialFunctions2
--R
--R SparseUnivariatePolynomialFunctions2(R: Ring,S: Ring) is a package constructor
--R Abbreviation for SparseUnivariatePolynomialFunctions2 is SUP2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SUP2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((R -> S),SparseUnivariatePolynomial(R)) -> SparseUnivariatePolynomial(S)
--R
--E 1

)spool
)lisp (bye)

— SparseUnivariatePolynomialFunctions2.help —

SparseUnivariatePolynomialFunctions2 examples

This package lifts a mapping from coefficient rings R to S to a mapping from sparse univariate polynomial over R to a sparse univariate polynomial over S. Note that the mapping is assumed to send zero to zero, since it will only be applied to the non-zero coefficients of the polynomial.
See Also:
- )show SparseUnivariatePolynomialFunctions2

---

SparseUnivariatePolynomialFunctions2 (SUP2) package

Exports:
- map

--- package SUP2 SparseUnivariatePolynomialFunctions2 ---

)abbrev package SUP2 SparseUnivariatePolynomialFunctions2
++ Description:
++ This package lifts a mapping from coefficient rings \( R \) to \( S \) to
++ a mapping from sparse univariate polynomial over \( R \) to
++ a sparse univariate polynomial over \( S \).
++ Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

SparseUnivariatePolynomialFunctions2(R:Ring, S:Ring): with
  map:(R->S,SparseUnivariatePolynomial R) -> SparseUnivariatePolynomial S
  ++ map(func, poly) creates a new polynomial by applying func to
  ++ every non-zero coefficient of the polynomial poly.
  == add
  map(f,p) == map(f,p)$UnivariatePolynomialCategoryFunctions2(R, SparseUnivariatePolynomial R, S, SparseUnivariatePolynomial S)

--- SUP2.dotabb ---
package SPECOUT SpecialOutputPackage

— SpecialOutputPackage.input —

)set break resume
)sys rm -f SpecialOutputPackage.output
)spool SpecialOutputPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SpecialOutputPackage

--R
--R SpecialOutputPackage is a package constructor
--R Abbreviation for SpecialOutputPackage is SPECOUT
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SPECOUT
--R
--R-------------------------------- Operations --------------------------------
--R outputAsFortran : OutputForm -> Void
--R outputAsScript : OutputForm -> Void
--R outputAsTex : OutputForm -> Void
--R outputAsFortran : (String,OutputForm) -> Void
--R outputAsFortran : List(OutputForm) -> Void
--R outputAsScript : List(OutputForm) -> Void
--R outputAsTex : List(OutputForm) -> Void

--R

)spool
)lisp (bye)

— SpecialOutputPackage.help —

====================================================================
SpecialOutputPackage examples
====================================================================

SpecialOutputPackage allows FORTRAN, TeX and Script Formula Formatter output from programs.

See Also:
o )show SpecialOutputPackage

SpecialOutputPackage (SPECOUT)

Exports:
outputAsTex outputAsFortran outputAsScript

— package SPECOUT SpecialOutputPackage —

)abbrev package SPECOUT SpecialOutputPackage
++ Author: Stephen M. Watt
++ Date Created: September 1986
++ Date Last Updated: May 23, 1991
++ Description:
++ SpecialOutputPackage allows FORTRAN, TeX and
++ Script Formula Formatter output from programs.

SpecialOutputPackage: public == private where
public == with
    outputAsFortran: (String,OutputForm) -> Void
      ++ outputAsFortran(v,o) sends output v = o in FORTRAN format
      ++ to the destination defined by \spad{set output fortran}.
    outputAsFortran: OutputForm -> Void
      ++ outputAsFortran(o) sends output o in FORTRAN format.
outputAsScript: OutputForm -> Void
++ outputAsScript(o) sends output o in Script Formula Formatter format
++ to the destination defined by \spadsyscom{set output formula}.
outputAsTex: OutputForm -> Void
++ outputAsTex(o) sends output o in Tex format to the destination
++ defined by \spadsyscom{set output tex}.
outputAsFortran: List OutputForm -> Void
++ outputAsFortran(l) sends (for each expression in the list l)
++ output in FORTRAN format to the destination defined by
++ \spadsyscom{set output fortran}.
outputAsScript: List OutputForm -> Void
++ outputAsScript(l) sends (for each expression in the list l)
++ output in Script Formula Formatter format to the destination defined.
++ by \spadsyscom{set output formula}.
outputAsTex: List OutputForm -> Void
++ outputAsTex(l) sends (for each expression in the list l)
++ output in Tex format to the destination as defined by
++ \spadsyscom{set output tex}.

private == add
e : OutputForm
l : List OutputForm
var : String
--ExpressionPackage()

juxtaposeTerms: List OutputForm -> OutputForm
juxtaposeTerms 1 == blankSeparate 1

outputAsFortran e ==
dispfortexp$Lisp e
void()$Void

outputAsFortran(var,e) ==
e := var::Symbol::OutputForm = e
dispfortexp(e)$Lisp
void()$Void

outputAsFortran l ==
dispfortexp$Lisp juxtaposeTerms 1
void()$Void

outputAsScript e ==
formulaFormat$Lisp e
void()$Void

outputAsScript l ==
formulaFormat$Lisp juxtaposeTerms 1
void()$Void

outputAsTex e ==
package SFQCMPK SquareFreeQuasiComponentPackage

--- SquareFreeQuasiComponentPackage.input ---

)set break resume
/sys rm -f SquareFreeQuasiComponentPackage.output
/spool SquareFreeQuasiComponentPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SquareFreeQuasiComponentPackage

--R
--R SquareFreeQuasiComponentPackage(R: GcdDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategory(R,E,V)) is a package constructor
--R Abbreviation for SquareFreeQuasiComponentPackage is SFQCMPK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SFQCMPK
--R
--R----------------------------------- Operations -----------------------------------
--R algebraicSort : List(TS) -> List(TS) moreAlgebraic? : (TS,TS) -> Boolean
--R stopTable! : () -> Void subTriSet? : (TS,TS) -> Boolean
--R branchIfCan : (List(P),TS,List(P),List(P),Boolean,Boolean,Boolean,Boolean) -> Union(Record(eq: List(P),tower: TS,ineq: List(P)),"failed")
--R infRittWu? : (List(P),List(P)) -> Boolean
--R internalInfRittWu? : (List(P),List(P)) -> Boolean
--R internalSubPolSet? : (List(P),List(P)) -> Boolean
--R internalSubQuasiComponent? : (TS,TS) -> Union(Boolean,"failed")
A internal package for removing redundant quasi-components and redundant branches when decomposing a variety by means of quasi-components of regular triangular sets.

See Also:
- )show SquareFreeQuasiComponentPackage

---

SquareFreeQuasiComponentPackage (SFQCMPK)
Exports:
    algebraicSort    branchIfCan
    infRittWu?      internalInfRittWu?
    internalSubPolSet?      internalSubQuasiComponent?
    moreAlgebraic?       prepareDecompose
    removeSuperfluousCases removeSuperfluousQuasiComponents
    startTable!         stopTable!
    subCase?           subPolSet?
    subQuasiComponent?        subTriSet?
    supDimElseRittWu?

<table>
<thead>
<tr>
<th>SFQCMPK.dotabb</th>
</tr>
</thead>
</table>

"SFQCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFQCMPK"]

package SFQCMPK SquareFreeQuasiComponentPackage

SFQCMPK.package SFQCMPK SquareFreeQuasiComponentPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ References :
++ [1] D. LAZARD "A new method for solving algebraic systems of
++ Tech. Report (PoSSo project)
++ d'extensions simples et resolution des systemes d'equations
++ Description:
++ A internal package for removing redundant quasi-components and redundant
++ branches when decomposing a variety by means of quasi-components
++ of regular triangular sets.

SquareFreeQuasiComponentPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
Key ==> Record(left:TS, right:TS)
Entry ==> Boolean
H ==> TabulatedComputationPackage(Key, Entry)
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
SQUAREFREE ==> SquareFreeRegularTriangularSetCategory(R,E,V,P)

Exports == with
  startTable!: (S,S,S) -> Void
    ++ \axiom{startTableGcd!(s1,s2,s3)}
    ++ is an internal subroutine, exported only for developement.
  stopTable!: () -> Void
    ++ \axiom{stopTableGcd!()}
    ++ is an internal subroutine, exported only for developement.
  supDimElseRittWu?: (TS,TS) -> Boolean
    ++ \axiom{supDimElseRittWu(ts,us)} returns true iff \axiom{ts}
    ++ has less elements than \axiom{us} otherwise if \axiom{ts}
    ++ has higher rank than \axiom{us} w.r.t. Riit and Wu ordering.
  algebraicSort: Split -> Split
    ++ \axiom{algebraicSort(lts)} sorts \axiom{lts} w.r.t
    ++ supDimElseRittWu from QuasiComponentPackage.
  moreAlgebraic?: (TS,TS) -> Boolean
    ++ \axiom{moreAlgebraic?(ts,us)} returns false iff \axiom{ts}
    ++ and \axiom{us} are both empty, or \axiom{ts}
    ++ has less elements than \axiom{us}, or some variable is
    ++ algebraic w.r.t. \axiom{us} and is not w.r.t. \axiom{ts}.
  subTriSet?: (TS,TS) -> Boolean
    ++ \axiom{subTriSet?(ts,us)} returns true iff \axiom{ts}
    ++ is a sub-set of \axiom{us}.
  subPolSet?: (LP, LP) -> Boolean
    ++ \axiom{subPolSet?(lp1,lp2)} returns true iff \axiom{lp1}
    ++ is a sub-set of \axiom{lp2}.
  internalSubPolSet?: (LP, LP) -> Boolean
    ++ \axiom{internalSubPolSet?(lp1,lp2)} returns true iff \axiom{lp1}
    ++ is a sub-set of \axiom{lp2} assuming that these lists are sorted
    ++ increasingly w.r.t.
    ++ infRittWu? from RecursivePolynomialCategory.
  internalInfRittWu?: (LP, LP) -> Boolean
    ++ \axiom{internalInfRittWu?(lp1,lp2)}
    ++ is an internal subroutine, exported only for developement.
  infRittWu?: (LP, LP) -> Boolean
    ++ \axiom{infRittWu?(lp1,lp2)}
++ is an internal subroutine, exported only for development.
internalSubQuasiComponent?: (TS,TS) -> Union(Boolean,"failed")
++ \texttt{internalSubQuasiComponent?(ts,us)} returns a boolean
++ \texttt{spad(b)} value if the fact the regular zero set of \texttt{axiom(us)}
++ contains that of \texttt{axiom(ts)} can be decided (and in that case
++ \texttt{axiom(b)} gives this inclusion) otherwise returns \texttt{axiom("failed")}.
subQuasiComponent?: (TS,TS) -> Boolean
++ \texttt{subQuasiComponent?(ts,us)} returns true iff
++ \texttt{internalSubQuasiComponent?(ts,us)} from QuasiComponentPackage
++ returns true.
subQuasiComponent?: (TS,Split) -> Boolean
++ \texttt{subQuasiComponent?(ts,us)} holds for one
++ \texttt{spad(us)} in \texttt{axiom(lus)}.
removeSuperfluousQuasiComponents: Split -> Split
++ \texttt{removeSuperfluousQuasiComponents(lts)} removes from
++ \texttt{axiom(lts)} any \texttt{spad(ts)} such that
++ \texttt{axiom(subQuasiComponent?(ts,us))} holds for
++ another \texttt{spad(us)} in \texttt{axiom(lts)}.
subCase?: (LpWT,LpWT) -> Boolean
++ \texttt{subCase?(lpwt1,lpwt2)}
++ is an internal subroutine, exported only for development.
removeSuperfluousCases: List LpWT -> List LpWT
++ \texttt{removeSuperfluousCases(llpwt)}
++ is an internal subroutine, exported only for development.
prepareDecompose: (LP, List(TS),B,B) -> List Branch
++ \texttt{prepareDecompose(lp,lts,b1,b2)}
++ is an internal subroutine, exported only for development.
branchIfCan: (LP,TS,LP,B,B,B,B,B) -> Union(Branch,"failed")
++ \texttt{branchIfCan(leq,ts,lineq,b1,b2,b3,b4,b5)}
++ is an internal subroutine, exported only for development.

Implementation == add

squareFreeFactors(lp: LP): LP ==
lsflp: LP := []
for p in lp repeat
  lsfp := squareFreeFactors(p)$polsetpack
  lsflp := concat(lsfp,lsflp)
sort(infRittWu?,removeDuplicates lsfpl)

startTable!(ok: S, ko: S, domainName: S): Void ==
  initTable!()$H
  if (not empty? ok) and (not empty? ko) then printInfo!(ok,ko)$H
  if (not empty? domainName) then startStats!(domainName)$H
  void()

stopTable!(): Void ==
  if makingStats?()$H then printStats!()$H
  clearTable!()$H
supDimElseRittWu? \((ts:TS,us:TS)\): Boolean ==
  #ts < #us => true
  #ts > #us => false
lp1 :LP := members(ts)
lp2 :LP := members(us)
while (not empty? lp1) and (not infRittWu?(first(lp2),first(lp1)))
  repeat
    lp1 := rest lp1
    lp2 := rest lp2
  not empty? lp1

algebraicSort \((lts:Split)\): Split ==
  lts := removeDuplicates lts
  sort(supDimElseRittWu?,lts)

moreAlgebraic?\((ts:TS,us:TS)\): Boolean ==
  empty? ts => empty? us
  empty? us => true
  #ts < #us => false
  for p in (members us) repeat
    not algebraic?(mvar(p),ts) => return false
  true

subTriSet?\((ts:TS,us:TS)\): Boolean ==
  empty? ts => true
  empty? us => false
  mvar(ts) > mvar(us) => false
  mvar(ts) < mvar(us) => subTriSet?(ts,rest(us)::TS)
  first(ts)::P = first(us)::P => subTriSet?(rest(ts)::TS,rest(us)::TS)
  false

internalSubPolSet?\((lp1: LP, lp2: LP)\): Boolean ==
  empty? lp1 => true
  empty? lp2 => false
  associates?(first lp1, first lp2) =>
    internalSubPolSet?(rest lp1, rest lp2)
  infRittWu?(first lp1, first lp2) => false
  internalSubPolSet?(lp1,lp2)

subPolSet?\((lp1: LP, lp2: LP)\): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalSubPolSet?(lp1,lp2)

infRittWu?\((lp1: LP, lp2: LP)\): Boolean ==
  lp1 := sort(infRittWu?, lp1)
  lp2 := sort(infRittWu?, lp2)
  internalInfRittWu?(lp1,lp2)
internalInfRittWu?(lp1: LP, lp2: LP): Boolean ==
  empty? lp1 => not empty? lp2
  empty? lp2 => false
  infRittWu?(first lp1, first lp2)$P => true
  infRittWu?(first lp2, first lp1)$P => false
  infRittWu?(rest lp1, rest lp2)$

subCase? (lpwt1:LpWT,lpwt2:LpWT): Boolean ==
  -- ASSUME lpwt.{1,2}.val is sorted w.r.t. infRittWu?
  not internalSubPolSet?(lpwt2.val, lpwt1.val) => false
  subQuasiComponent?(lpwt1.tower,lpwt2.tower)

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P) then

  internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
  subTriSet?(us,ts) => true
  not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
  for p in (members us) repeat
    mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
      return("failed":Union(Boolean,"failed"))
  for p in (members us) repeat
    not zero? initiallyReduce(p,ts) =>
      return("failed":Union(Boolean,"failed"))
  lsfp := squareFreeFactors(initials us)
  for p in lsfp repeat
    b: B := invertible?(p,ts)$TS
    not b =>
      return(false::Union(Boolean,"failed"))
  true::Union(Boolean,"failed")

else

  internalSubQuasiComponent?(ts:TS,us:TS): Union(Boolean,"failed") ==
  subTriSet?(us,ts) => true
  not moreAlgebraic?(ts,us) => false::Union(Boolean,"failed")
  for p in (members us) repeat
    mdeg(p) < mdeg(select(ts,mvar(p))::P) =>
      return("failed":Union(Boolean,"failed"))
  for p in (members us) repeat
    not zero? reduceByQuasiMonic(p,ts) =>
      return("failed":Union(Boolean,"failed"))
  true::Union(Boolean,"failed")

subQuasiComponent?(ts:TS,us:TS): Boolean ==
  k: Key := [ts, us]
  e := extractIfCan(k)$H
  e case Entry => e::Entry
  ubf: Union(Boolean,"failed") := internalSubQuasiComponent?(ts,us)
  b: Boolean := (ubf case Boolean) and (ubf::Boolean)
insert!(k,b)$H
b

subQuasiComponent?(ts:TS,lus:Split): Boolean ==
  for us in lus repeat
    subQuasiComponent?(ts,us)@B => return true
  false

removeSuperfluousCases (cases:List LpWT) ==
  #cases < 2 => cases
toSee :=
    sort((x:LpWT,y:LpWT):Boolean +-> supDimElseRitt Wu?(x.tower,y.tower), cases)
lpwt1,lpwt2 : LpWT
toSave,headmaxcases,maxcases,copymaxcases : List LpWT
while not empty? toSee repeat
  lpwt1 := first toSee
toSee := rest toSee
toSave := []
  for lpwt2 in toSee repeat
    if subCase?(lpwt1,lpwt2)
      then
        lpwt1 := lpwt2
    else
      if not subCase?(lpwt2,lpwt1)
        then
          toSave := cons(lpwt2,toSave)
  if empty? maxcases
    then
      headmaxcases := [lpwt1]
      maxcases := headmaxcases
    else
      copymaxcases := maxcases
      while (not empty? copymaxcases) and _
        (not subCase?(lpwt1,first(copymaxcases))) repeat
        copymaxcases := rest copymaxcases
      if empty? copymaxcases
        then
          setrest!(headmaxcases,[lpwt1])
      headmaxcases := rest headmaxcases
toSee := reverse toSave
maxcases

removeSuperfluousQuasiComponents(lts: Split): Split ==
lts := removeDuplicates lts
#lts < 2 => lts
toSee := algebraicSort lts
toSave,headmaxlts,maxlts,copymaxlts : Split
while not empty? toSee repeat
  ts := first toSee
toSee := rest toSee
toListSave := []
for us in toSee repeat
  if subQuasiComponent?(ts,us)@B
    then
      ts := us
    else
      if not subQuasiComponent?(us,ts)@B
        then
          toListSave := cons(us,toListSave)
  if empty? maxlts
    then
      headmaxlts := [ts]
      maxlts := headmaxlts
    else
      copymaxlts := maxlts
      while (not empty? copymaxlts) and
          (not subQuasiComponent?(ts,first(copymaxlts))@B) repeat
        copymaxlts := rest copymaxlts
      if empty? copymaxlts
        then
          setrest!(headmaxlts,[ts])
          headmaxlts := rest headmaxlts
      toSee := reverse tolistSave
      algebraicSort maxlts

removeAssociates (lp:LP):LP ==
  removeDuplicates [primitivePart(p) for p in lp]

branchIfCan(leq: LP, ts: TS, lineq: LP, b1:B, b2:B, b3:B, b4:B, b5:B):UBF ==
  -- ASSUME pols in leq are squarefree and mainly primitive
  -- if b1 then CLEAN UP leq
  -- if b2 then CLEAN UP lineq
  -- if b3 then SEARCH for ZERO in lineq with leq
  -- if b4 then SEARCH for ZERO in lineq with ts
  -- if b5 then SEARCH for ONE in leq with lineq
  if b1
    then
      leq := removeAssociates(leq)
      leq := remove(zero?,leq)
      any?(ground?,leq) =>
        return("failed"::Union(Branch,"failed"))
  if b2
    then
      any?(zero?,lineq) =>
        return("failed"::Union(Branch,"failed"))
      lineq := removeRedundantFactors(lineq)$polsetpack
  if b3
    then
      ps: PS := construct(leq)$PS
for q in lineq repeat
    zero? remainder(q,ps).polnum =>
        return("failed":Union(Branch,"failed"))
(empty? leq) or (empty? lineq) => ([leq, ts, lineq]$Branch)::UBF
if b4
    then
        for q in lineq repeat
            zero? initiallyReduce(q,ts) =>
                return("failed":Union(Branch,"failed"))
if b5
    then
        newleq: LP := []
        for p in leq repeat
            for q in lineq repeat
                if mvar(p) = mvar(q)
                    then
                        g := gcd(p,q)
                        newp := (p exquo g)::P
                        ground? newp =>
                            return("failed":Union(Branch,"failed"))
                        newleq := cons(newp,newleq)
                    else
                        newleq := cons(p,newleq)
        leq := newleq
        leq := sort(infRittWu?, removeDuplicates leq)
        ([leq, ts, lineq]$Branch)::UBF
prepareDecompose(lp: LP, lts: List(TS), b1: B, b2: B): List Branch ==
-- if b1 then REMOVE REDUNDANT COMPONENTS in lts
-- if b2 then SPLIT the input system with squareFree
lp := sort(infRittWu?, remove(zero?,removeAssociates(lp)))
any?(ground?,lp) => []
empty? lts => []
if b1 then lts := removeSuperfluousQuasiComponents lts
not b2 =>
    [[lp,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
toSee: List Branch
lq: LP := []
toSee := [[lq,ts,squareFreeFactors(initials ts)]$Branch for ts in lts]
empty? lp => toSee
for p in lp repeat
    lsfp := squareFreeFactors(p)$polsetpack
    branches: List Branch := []
    lq := []
    for f in lsfp repeat
        for branch in toSee repeat
            leq : LP := branch.eq
ts := branch.tower
        lineq : LP := branch.ineq
        ubf1: UBF := branchIfCan(leq,ts,lq,false,false,true,true,true)@UBF
ubf1 case "failed" => "leave"
ubf2: UBF :=
    branchIfCan([f],ts,lineq,false,false,true,true,true)@UBF
ubf2 case "failed" => "leave"
leq := sort(infRittWu?,removeDuplicates concat(ubf1.eq,ubf2.eq))
lineq :=
    sort(infRittWu?,removeDuplicates concat(ubf1.ineq,ubf2.ineq))
newBranch :=
    branchIfCan(leq,ts,lineq,false,false,false,false,false)
branches:= cons(newBranch::Branch,branches)
lq := cons(f,lq)
toSee := branches
sort((x,y) +-> supDimElseRittWu?(x.tower,y.tower),toSee)

———

— SFQCMPK.dotabb ——

"SFQCMPK" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFQCMPK"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SFQCMPK" -> "SFRTCAT"

———

package SRDCMPK SquareFreeRegularSetDecompositionPackage

— SquareFreeRegularSetDecompositionPackage.input —

)set break resume
)sys rm -f SquareFreeRegularSetDecompositionPackage.output
)spool SquareFreeRegularSetDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SquareFreeRegularSetDecompositionPackage

--R
--R SquareFreeRegularSetDecompositionPackage(R: GcdDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategory(R,E,V),TS: SquareFreeRegularTriangularSetCategory(R,E,V,P)) is a package constructor
--R Abbreviation for SquareFreeRegularSetDecompositionPackage is SRDCMPK
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SRDCMPK
--R
--R---------------------------------------- Operations----------------------------------------
--R KrullNumber : (List(P),List(TS)) -> NonNegativeInteger
--R algebraicDecompose : (P,TS) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))
--R convert : Record(val: List(P),tower: TS) -> String
--R decompose : (List(P),List(TS),Boolean,Boolean) -> List(TS)
--R decompose : (List(P),List(TS),Boolean,Boolean,Boolean,Boolean,Boolean) -> List(TS)
--R internalDecompose : (P,TS,NonNegativeInteger,Boolean) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))
--R internalDecompose : (P,TS,NonNegativeInteger) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))
--R internalDecompose : (P,TS) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))
--R numberOfVariables : (List(P),List(TS)) -> NonNegativeInteger
--R printInfo : (List(Record(val: List(P),tower: TS)),NonNegativeInteger) -> Void
--R transcendentalDecompose : (P,TS,NonNegativeInteger) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))
--R transcendentalDecompose : (P,TS) -> Record(done: List(TS),todo: List(Record(val: List(P),tower: TS)))
--R upDateBranches : (List(P),List(TS),List(Record(val: List(P),tower: TS)),Record(done: List(TS),todo: List(TS)))

)spool
)lisp (bye)

——

---------

SquareFreeRegularSetDecompositionPackage.help

====================================================================
SquareFreeRegularSetDecompositionPackage examples
====================================================================

A package providing a new algorithm for solving polynomial systems by means of regular chains. Two ways of solving are provided: in the sense of Zariski closure (like in Kalkbrener's algorithm) or in the sense of the regular zeros (like in Wu, Wang or Lazard-Moreno methods). This algorithm is valid for any type of regular set. It does not care about the way a polynomial is added in an regular set, or how two quasi-components are compared (by an inclusion-test), or how the invertibility test is made in the tower of simple extensions associated with a regular set.

These operations are realized respectively by the domain TS and the packages QCMPPK(R,E,V,P,TS) and RSETGCD(R,E,V,P,TS).

The same way it does not care about the way univariate polynomial gcds (with coefficients in the tower of simple extensions associated with a regular set) are computed. The only requirement is that these gcds need to have invertible initials (normalized or not).

WARNING. There is no need for a user to call directly any operation of this package since they can be accessed by the domain \axiomType{TS}. 

Thus, the operations of this package are not documented.

See Also:
o )show SquareFreeRegularSetDecompositionPackage

---

SquareFreeRegularSetDecompositionPackage (SRDCMPK)

Exports:

- algebraicDecompose
- convert
- decompose
- internalDecompose
- KrullNumber
- numberOfVariables
- printInfo
- transcendentalDecompose
- upDateBranches

---

)abbrev package SRDCMPK SquareFreeRegularSetDecompositionPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 12/16/1998
++ References :
++ Description:
++ A package providing a new algorithm for solving polynomial systems
++ by means of regular chains. Two ways of solving are provided:
++ in the sense of Zariski closure (like in Kalkbrener’s algorithm)
++ or in the sense of the regular zeros (like in Wu, Wang or Lazard-
++ Moreno methods). This algorithm is valid for any type
++ of regular set. It does not care about the way a polynomial is
++ added in an regular set, or how two quasi-components are compared
++ (by an inclusion-test), or how the invertibility test is made in
++ the tower of simple extensions associated with a regular set.
These operations are realized respectively by the domain \spad{TS} and the packages \spad{QCMPPK(R,E,V,P,TS)} and \spad{RSETGCD(R,E,V,P,TS)}. The same way it does not care about the way univariate polynomial gcds (with coefficients in the tower of simple extensions associated with a regular set) are computed. The only requirement is that these gcds need to have invertible initials (normalized or not).

**WARNING.** There is no need for a user to call directly any operation of this package since they can be accessed by the domain \spad{axiomType{TS}}. Thus, the operations of this package are not documented.

\begin{verbatim}
SquareFreeRegularSetDecompositionPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : SquareFreeRegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
LP ==> List P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Wip ==> Record(done: Split, todo: List LpWT)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,TS)
regsetgcdpack ==> SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS)

Exports == with

KrullNumber: (LP, Split) -> N
numberOfVariables: (LP, Split) -> N
algebraicDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
transcendentalDecompose: (P,TS,N) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS,N,B) -> Record(done: Split, todo: List LpWT)
internalDecompose: (P,TS) -> Record(done: Split, todo: List LpWT)
decompose: (LP, Split, B, B) -> Split
decompose: (LP, Split, B, B, B, B) -> Split
upDateBranches: (LP,Split,List LpWT,Wip,N) -> List LpWT
convert: Record(val: List P,tower: TS) -> String
printInfo: (List Record(val: List P,tower: TS), N) -> Void
\end{verbatim}
Implementation == add

KrullNumber(lp: LP, lts: Split): N ==
  ln: List N := [#(ts) for ts in lts]
  n := #lp + reduce(max,ln)

numberOfVariables(lp: LP, lts: Split): N ==
  lv: List V := variables([lp$PS])
  for ts in lts repeat lv := concat(variables(ts), lv)
  # removeDuplicates(lv)

  ground? p =>
    error " in algebraicDecompose$REGSET: should never happen !"
  v := mvar(p); n := #ts
  ts_v_- := collectUnder(ts,v)
  ts_v_+ := collectUpper(ts,v)
  ts_v := select(ts,v)::P
  lgwt: List PWT
  if mdeg(p) < mdeg(ts_v)
    then
      lgwt := stoseInternalLastSubResultant(ts_v,p,ts_v_-,true,false)$regsetgcdpack
    else
      lgwt := stoseInternalLastSubResultant(p,ts_v,ts_v_-,true,false)$regsetgcdpack
  lts: Split := []
  llpwt: List LpWT := []
  for gwt in lgwt repeat
    g := gwt.val; us := gwt.tower
    zero? g =>
      error " in algebraicDecompose$REGSET: should never happen !!!!"
    ground? g => "leave"
    h := leadingCoefficient(g,v)
    lus := augment(members(ts_v_+),augment(ts_v,us)$TS)$TS
    lsfp := squareFreeFactors(h)$polsetpack
    for f in lsfp repeat
      ground? f => "leave"
      for vs in lus repeat
        llpwt := cons([[f,p],vs]$LpWT, llpwt)
    n < #us =>
      error " in algebraicDecompose$REGSET: should never happen !!!!"
    mvar(g) = v =>
      lts := concat(augment(members(ts_v_+),augment(g,us)$TS)$TS,$TS,lts)
  [lts,llpwt]

  lts: Split
  if #ts < bound
    then
      lts := augment(p,ts)$TS
    else
   
   lts := augment(p,ts)$TS
   llpwt := List LpWT := []
   [lts,llpwt]
   
   clos? => internalDecompose(p,ts,bound)
   internalDecompose(p,ts)
   
   -- ASSUME p not constant
   llpwt := List LpWT := []
   lts: Split := []
   -- EITHER mvar(p) is null
   if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
   then
      llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
      p := (p exquo lmp)::P
   ip := squareFreePart init(p); tp := tail p
   p := mainPrimitivePart p
   -- OR init(p) is null or not
   lbwt: List BWT := stoseInvertible?_sqfreg(ip,ts)$regsetgcdpack
   for bwt in lbwt repeat
      bwt.val =>
      if algebraic?(mvar(p),bwt.tower)
      then
         rsl := algebraicDecompose(p,bwt.tower)
      else
         rsl := transcendentalDecompose(p,bwt.tower,bound)
      lts := concat(rsl.done,lts)
      llpwt := concat(rsl.todo,llpwt)
      (not ground? ip) =>
         zero? tp => llpwt := cons([[ip,bwt.tower]$LpWT, llpwt])
         (not ground? tp) => llpwt := cons([[ip,tp,bwt.tower]$LpWT, llpwt])
         riv := removeZero(ip,bwt.tower)
         (zero? riv) =>
         zero? tp => lts := cons(bwt.tower,lts)
         (not ground? tp) => llpwt := cons([[tp,bwt.tower]$LpWT, llpwt])
         llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
      [lts,llpwt]
   -- ASSUME p not constant
   llpwt := List LpWT := []
   lts: Split := []
   -- EITHER mvar(p) is null
if (not zero? tail(p)) and (not ground? (lmp := leastMonomial(p)))
then
  llpwt := cons([[mvar(p)::P],ts]$LpWT,llpwt)
p := (p exquo lmp)::P
ip := squareFreePart init(p); tp := tail p
p := mainPrimitivePart p
-- OR init(p) is null or not
lbwt: List BWT := stoseInvertible?_sqfreg(ip,ts)$regsetgcdpack
for bwt in lbwt repeat
  bwt.val =>
    if algebraic?(mvar(p),bwt.tower)
      then
        rsl := algebraicDecompose(p,bwt.tower)
      else
        rsl := transcendentalDecompose(p,bwt.tower)
lts := concat(rsl.done,lts)
  llpwt := concat(rsl.todo,llpwt)
  (not ground? ip) =>
    zero? tp => llpwt := cons([[ip],bwt.tower]$LpWT, llpwt)
  (not ground? tp) => llpwt := cons([[ip,tp],bwt.tower]$LpWT, llpwt)
  riv := removeZero(ip,bwt.tower)
  (zero? riv) =>
    zero? tp => lts := cons(bwt.tower,lts)
  (not ground? tp) => llpwt := cons([[tp],bwt.tower]$LpWT, llpwt)
  llpwt := cons([[riv * mainMonomial(p) + tp],bwt.tower]$LpWT, llpwt)
[lts,llpwt]
decompose(lp,lts,false,false,clos?,true,info?)
convert(lpwt: LpWT): String ==
  ls: List String := ["<", string((#(lpwt.val))::Z), ",", string((#(lpwt.tower))::Z), ">"]
  concat ls
printInfo(toSee: List LpWT, n: N): Void ==
  lpwt := first toSee
  s: String := concat ["[", string((#toSee)::Z), ",", convert(lpwt)@String]
  m: N := #(lpwt.val)
toSee := rest toSee
  for lpwt in toSee repeat
    m := m + #(lpwt.val)
    s := concat [s, ",", convert(lpwt)@String]
  s := concat [s, " -> |", string(m::Z), "; {", string(n::Z),"}]"
  iprint(s)$iprintpack
  void()
  -- if cleanW? then REMOVE REDUNDANT COMPONENTS in lts
  -- if sqfr? then SPLIT the system with SQUARE-FREE FACTORIZATION
  -- if clos? then SOLVE in the closure sense
-- if rem? then REDUCE the current p by using remainder
-- if info? then PRINT info
empty? lp => lts
branches: List Branch := prepareDecompose(lp,lts,cleanW?,sqfr?)$quasicomppack
empty? branches => []
toSee: List LpWT := [[br.eq,br.tower]$LpWT for br in branches]
toSave: Split := []
if clos? then bound := KrullNumber(lp,lts) else bound := numberOfVariables(lp,lts)
while (not empty? toSee) repeat
  if info? then printInfo(toSee,#toSave)
  lpwt := first toSee; toSee := rest toSee
  lp := lpwt.val; ts := lpwt.tower
  empty? lp =>
    toSave := cons(ts, toSave)
  p := first lp; lp := rest lp
  if rem? and (not ground? p) and (not empty? ts) then
    p := remainder(p,ts).polnum
  p := removeZero(p,ts)
  zero? p => toSee := cons([lp,ts]$LpWT, toSee)
  ground? p => "leave"
  rsl := internalDecompose(p,ts,bound,clos?)
toSee := upDateBranches(lp,toSave,toSee,rsl,bound)
removeSuperfluousQuasiComponents(toSave)$quasicomppack

newBranches: List LpWT := wip.todo
newComponents: Split := wip.done
branches1, branches2: List LpWT
branches1 := []; branches2 := []
for branch in newBranches repeat
  us := branch.tower
  #us > n => "leave"
  newleq := sort(infRittWu?,concat(leq,branch.val))
  --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
  --any?(ground?,foo) => "leave"
  branches1 := cons([newleq,us]$LpWT, branches1)
for us in newComponents repeat
  #us > n => "leave"
  subQuasiComponent?(us,lts)$quasicomppack => "leave"
  newleq := leq
  --foo := rewriteSetWithReduction(newleq,us,initiallyReduce,initiallyReduced?)
  --any?(ground?,foo) => "leave"
  branches2 := cons([leq,us]$LpWT, branches2)
empty? branches1 =>
  empty? branches2 => current
  concat(branches2, current)
branches := concat [branches2, branches1, current]
-- branches := concat(branches, current)
removeSuperfluousCases(branches)$quasicomppack
package SFRGCD SquareFreeRegularTriangularSetGcdPackage

-- SquareFreeRegularTriangularSetGcdPackage.input —

)set break resume
)sys rm -f SquareFreeRegularTriangularSetGcdPackage.output
)spool SquareFreeRegularTriangularSetGcdPackage.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show SquareFreeRegularTriangularSetGcdPackage

--R
--R SquareFreeRegularTriangularSetGcdPackage(R: GcdDomain,E: OrderedAbelianMonoidSup,V: OrderedSet,P: RecursivePolynomialCategory(R,E,V),TS: RegularTriangularSetCategory(R,E,V,P)) is a package constructor
--R Abbreviation for SquareFreeRegularTriangularSetGcdPackage is SFRGCD
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SFRGCD
--R
--R-------------------------------------- Operations --------------------------------------
--R stopTableGcd! : () -> Void
--R stopTableInvSet! : () -> Void
--R stoseInvertible? : (P,TS) -> Boolean
--R startTableGcd! : (String,String,String) -> Void
--R startTableInvSet! : (String,String,String) -> Void
--R stoseIntegralLastSubResultant : (P,P,TS) -> List(Record(val: P,tower: TS))
--R stoseInternalLastSubResultant : (P,TS,Boolean,Boolean) -> List(Record(val: P,tower: TS))
--R stoseInternalLastSubResultant : (List(Record(val: List(P),tower: TS)),V,Boolean) -> List(Record(val: P,tower: TS))
--R stoseInvertible? : (P,TS) -> List(Record(val: Boolean,tower: TS))
--R stoseInvertible?reg : (P,TS) -> List(Record(val: Boolean,tower: TS))
--R stoseInvertible?sqfreg : (P,TS) -> List(Record(val: Boolean,tower: TS))
--R stoseInvertibleSet : (P,TS) -> List(TS)
--R stoseInvertibleSetreg : (P,TS) -> List(TS)
--R stoseInvertibleSetsqfreg : (P,TS) -> List(TS)
--R stoseLastSubResultant : (P,P,TS) -> List(Record(val: P,tower: TS))
--R stosePrepareSubResAlgo : (P,P,TS) -> List(Record(val: List(P),tower: TS))
--R stoseSquareFreePart : (P,TS) -> List(Record(val: P,tower: TS))
--E 1

)spool
)lisp (bye)

——

--- SquareFreeRegularTriangularSetGcdPackage.help ---

====================================================================
SquareFreeRegularTriangularSetGcdPackage examples
====================================================================

A internal package for computing gcds and resultants of univariate
polynomials with coefficients in a tower of simple extensions of a field.
There is no need to use directly this package since its main operations are
available from TS.

See Also:
o )show SquareFreeRegularTriangularSetGcdPackage

——

SquareFreeRegularTriangularSetGcdPackage (SFRGCD)
Exports:
stopTableGcd! stopTableInvSet!
startTableGcd! startTableInvSet!
stoseIntegralLastSubResultant stoseInternalLastSubResultant
stoseInvertible? stoseInvertible?reg
stoseInvertible?qsfreg stoseInvertibleSet
stoseInvertibleSetreg stoseInvertibleSetsqfreg
stoseLastSubResultant stosePrepareSubResAlgo
stoseSquareFreePart

— package SFRGCD SquareFreeRegularTriangularSetGcdPackage —

)abbrev package SFRGCD SquareFreeRegularTriangularSetGcdPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/23/1998
++ Date Last Updated: 10/01/1998
++ References :
++ [1] M. MORENO MAZA and R. RIOBOO "Computations of gcd over
++ algebraic towers of simple extensions" In proceedings of AAECC11
++ d'extensions simples et resolution des systemes d'equations
++ Description:
++ A internal package for computing gcds and resultants of univariate
++ polynomials with coefficients in a tower of simple extensions of a field.
++ There is no need to use directly this package since its main operations are
++ available from \spad{TS}.

SquareFreeRegularTriangularSetGcdPackage(R,E,V,P,TS): Exports == Implementation where

R : GcdDomain
E : OrderedAbelianMonoidSup
V : OrderedSet
P : RecursivePolynomialCategory(R,E,V)
TS : RegularTriangularSetCategory(R,E,V,P)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
S ==> String
LP ==> List P
PtoP ==> P -> P
PS ==> GeneralPolynomialSet(R,E,V,P)
PWT ==> Record(val : P, tower : TS)
BWT ==> Record(val : Boolean, tower : TS)
LpWT ==> Record(val : (List P), tower : TS)
Branch ==> Record(eq: List P, tower: TS, ineq: List P)
UBF ==> Union(Branch,"failed")
Split ==> List TS
KeyGcd ==> Record(arg1: P, arg2: P, arg3: TS, arg4: B)
EntryGcd ==> List PWT
HGcd ==> TabulatedComputationPackage(KeyGcd, EntryGcd)
KeyInvSet ==> Record(arg1: P, arg3: TS)
EntryInvSet ==> List TS
HInvSet ==> TabulatedComputationPackage(KeyInvSet, EntryInvSet)
iprintpack ==> InternalPrintPackage()
polsetpack ==> PolynomialSetUtilitiesPackage(R,E,V,P)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E,V,P,TS)

SQUAREFREE ==> SquareFreeRegularTriangularSetCategory(R,E,V,P)

Exports == with
  startTableGcd!: (S,S,S) -> Void
  stopTableGcd!: () -> Void
  startTableInvSet!: (S,S,S) -> Void
  stopTableInvSet!: () -> Void
  stosePrepareSubResAlgo: (P,P,TS) -> List LpWT
  stoseInternalLastSubResultant: (P,P,TS,B,B) -> List PWT
  stoseInternalLastSubResultant: (List LpWT,V,B) -> List PWT
  stoseIntegralLastSubResultant: (P,P,TS) -> List PWT
  stoseLastSubResultant: (P,P,TS) -> List PWT
  stoseInvertible?: (P,TS) -> B
  stoseInvertible?_sqfreg: (P,TS) -> List BWT
  stoseInvertibleSet_sqfreg: (P,TS) -> Split
  stoseInvertible?_reg: (P,TS) -> List BWT
  stoseInvertibleSet_reg: (P,TS) -> Split
  stoseInvertible?: (P,TS) -> List BWT
  stoseInvertibleSet: (P,TS) -> Split
  stoseSquareFreePart: (P,TS) -> List PWT

Implementation == add

  startTableGcd!(ok: S, ko: S, domainName: S): Void ==
    initTable!()$HGcd
    printInfo!(ok,ko)$HGcd
    startStats!(domainName)$HGcd
    void()

  stopTableGcd!(): Void ==
    if makingStats?()$HGcd then printStats!()$HGcd
    clearTable!()$HGcd

  startTableInvSet!(ok: S, ko: S, domainName: S): Void ==
    initTable!()$HInvSet
    printInfo!(ok,ko)$HInvSet
    startStats!(domainName)$HInvSet
void()

stopTableInvSet!: Void ==
  if makingStats?()$HInvSet then printStats!()$HInvSet
  clearTable!()$HInvSet

  q := primitivePart initiallyReduce(p,ts)
  zero? q => false
  normalized?(q,ts) => true
  v := mvar(q)
  not algebraic?(v,ts) =>
    toCheck: List BWT := stoseInvertible?(p,ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val = false => return false
    return true
  ts_v := select(ts,v)::P
  ts_v_- := collectUnder(ts,v)
  lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,true)
  for gwt in lgwt repeat
    g := gwt.val;
    (not ground? g) and (mvar(g) = v) =>
      return false
  true

  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME init(p1) invertible modulo ts !!!!
  toSee: List LpWT := [[[p1,p2],ts]$LpWT]
  toSave: List LpWT := []
  v := mvar(p1)
  while (not empty? toSee) repeat
    lpwt := first toSee; toSee := rest toSee
    p1 := lpwt.val.1; p2 := lpwt.val.2
    ts := lpwt.tower
    lbwt := stoseInvertible?(leadingCoefficient(p2,v),ts)@(List BWT)
    for bwt in lbwt repeat
      (bwt.val = true) and (degree(p2,v) > 0) =>
        p3 := prem(p1, -p2)
        s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
        toSave := cons([[p2,p3,s],bwt.tower]$LpWT,toSave)
        -- p2 := initiallyReduce(p2,bwt.tower)
        newp2 := primitivePart initiallyReduce(p2,bwt.tower)
        (bwt.val = true) =>
          -- toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
          toSave := cons([[p2,0,1],bwt.tower]$LpWT,toSave)
          -- zero? p2 =>
          zero? newp2 =>
            toSave := cons([[p1,0,1],bwt.tower]$LpWT,toSave)
        -- toSee := cons([[p1,p2],bwt.tower]$LpWT,toSee)
toSee := cons(\[\[p1, newp2\], bwt.tower\]$LpWT, toSee)

toSave

stoseIntegralLastSubResultant(p1:P, p2:P, ts:TS) : List PWT ==
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- ASSUME p1 and p2 have no algebraic coefficients
  lsr := lastSubResultant(p1, p2)
  ground?(lsr) => \[\[lsr, ts\]$PWT
  mvar(lsr) < mvar(p1) => \[\[lsr, ts\]$PWT
  gi1i2 := gcd(init(p1), init(p2))
  ex: Union(P, "failed") := (gi1i2 * lsr) exquo$P init(lsr)
  ex case "failed" => \[\[lsr, ts\]$PWT
  \[\[ex::P, ts\]$PWT

stoseInternalLastSubResultant(p1:P, p2:P, ts:TS, b1:B, b2:B) : List PWT ==
  -- ASSUME mvar(p1) = mvar(p2) > mvar(ts) and mdeg(p1) >= mdeg(p2)
  -- if b1 ASSUME init(p2) invertible w.r.t. ts
  -- if b2 BREAK with the first non-trivial gcd
  k: KeyGcd := \[p1, p2, ts, b2\]
  e := extractIfCan(k)$HGcd
  e case EntryGcd => e::EntryGcd
  toSave: List PWT
  empty? ts =>
    toSave := stoseIntegralLastSubResultant(p1, p2, ts)
    insert!(k, toSave)$HGcd
    return toSave
  toSee: List LpWT
  if b1
    then
      p3 := prem(p1, -p2)
      s: P := init(p2)**(mdeg(p1) - mdeg(p2))::N
      toSee := \[\[p2, p3, s, ts\]$LpWT
    else
      toSee := stosePrepareSubResAlgo(p1, p2, ts)
    toSave := stoseInternalLastSubResultant(toSee, mvar(p1), b2)
    insert!(k, toSave)$HGcd
    toSave
  stoseInternalLastSubResultant(llpwt: List LpWT, v:V, b2:B) : List PWT ==
  toReturn: List PWT := []; toSee: List LpWT;
  while (not empty? llpwt) repeat
    toSee := llpwt; llpwt := []
    -- CONSIDER FIRST the vanishing current last subresultant
    for lpwt in toSee repeat
      p1 := lpwt.val.1;
      p2 := lpwt.val.2;
      s := lpwt.val.3;
      ts := lpwt.tower
      lbwt := stoseInvertible?(leadingCoefficient(p2, v), ts)@(List BWT)
      for bwt in lbwt repeat
bwt.val = false =>
  toReturn := cons([p1,bwt.tower]$PWT, toReturn)

b2 and positive?(degree(p1,v)) => return toReturn

llpwt := cons([[p1,p2,s],bwt.tower]$LpWT, llpwt)

empty? llpwt => "leave"

-- CONSIDER NOW the branches where the computations continue

toSee := llpwt; llpwt := []

lpwt := first toSee; toSee := rest toSee

p1 := lpwt.val.1; p2 := lpwt.val.2; s := lpwt.val.3

delta: N := (mdeg(p1) - degree(p2,v))::N

p3: P := LazardQuotient2(p2, leadingCoefficient(p2,v), s, delta)

zero?(degree(p3,v)) =>
  toReturn := cons([p3,lpwt.tower]$PWT, toReturn)

for lpwt in toSee repeat
  toReturn := cons([p3,lpwt.tower]$PWT, toReturn)

(p1, p2) := (p3, next_subResultant2(p1, p2, p3, s))

s := leadingCoefficient(p1,v)

llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)

for lpwt in toSee repeat
  llpwt := cons([[p1,p2,s],lpwt.tower]$LpWT, llpwt)

return

stoseInternalLastSubResultant(p1,P,p2:P,ts:TS): List PWT ==

  ground? p1 =>
    error "in stoseInternalLastSubResultantElseSplit$SFRGCD : bad #1"

  ground? p2 =>
    error "in stoseInternalLastSubResultantElseSplit$SFRGCD : bad #2"

  not (mvar(p2) = mvar(p1)) =>
    error "in stoseInternalLastSubResultantElseSplit$SFRGCD : bad #2"

  algebraic?(mvar(p1),ts) =>
    error "in stoseInternalLastSubResultantElseSplit$SFRGCD : bad #1"

  not initiallyReduced?(p1,ts) =>
    error "in stoseInternalLastSubResultantElseSplit$SFRGCD : bad #1"

  not initiallyReduced?(p2,ts) =>
    error "in stoseInternalLastSubResultantElseSplit$SFRGCD : bad #2"

  purelyTranscendental?(p1,ts) and purelyTranscendental?(p2,ts) =>
    stoseIntegralLastSubResultant(p1,p2,ts)

if mdeg(p1) < mdeg(p2) then
  (p1, p2) := (p2, p1)

if odd?(mdeg(p1)) and odd?(mdeg(p2)) then p2 := - p2

stoseInternalLastSubResultant(p1,p2,ts,false,false)


  -- ASSUME p is not constant and mvar(p) > mvar(ts)
  -- ASSUME init(p) is invertible w.r.t. ts
  -- ASSUME p is mainly primitive

  one? mdeg(p) => [[p,ts]$PWT]

  mdeg(p) = 1 => [[p,ts]$PWT]

  v := mvar(p)$P

  q: P := mainPrimitivePart D(p,v)
licate: List PWT := stoseInternalLastSubResultant(p,q,ts,true,false)
lwp := List PWT := []
sfp : P
for gwt in lgwt repeat
g := gwt.val; us := gwt.tower
(ground? g) or (mvar(g) < v) =>
lwp := cons([p,us],lwp)
g := mainPrimitivePart g
sfp := lazyPquo(p,g)
sfp := mainPrimitivePart stronglyReduce(sfp,us)
lwp := cons([sfp,us],lwp)
lwp

stoseInvertible?_sqfreg(p:P,ts:TS): List BWT ==
--iprint("+")$iprintpack
q := primitivePart initiallyReduce(p,ts)
zero? q => [[false,ts]$BWT]
normalized?(q,ts) => [[true,ts]$BWT]
v := mvar(q)
not algebraic?(v,ts) =>
lbw := List BWT := []
toCheck: List BWT := stoseInvertible?_sqfreg(init(q),ts)@(List BWT)
for bwt in toCheck repeat
  bwt.val => lbw := cons(bwt,lbw)
  newq := removeZero(q,bwt.tower)
  zero? newq => lbw := cons(bwt,lbw)
  lbw :=
    concat(stoseInvertible?_sqfreg(newq,bwt.tower)@(List BWT), lbw)
return lbw
ts_v := select(ts,v)::P
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
lbw := List BWT := []
lts, lts_g, lts_h: Split
for gwt in lgwt repeat
g := gwt.val; ts := gwt.tower
(ground? g) or (mvar(g) < v) =>
lts := augment(ts_v-ts)$TS
  lts := augment(members(ts_v_+),lts)$TS
for ts in lts repeat
  lbw := cons([true, ts]$BWT,lbw)
g := mainPrimitivePart g
lts_g := augment(g,ts)$TS
lts_g := augment(members(ts_v_+),lts_g)$TS
-- USE stoseInternalAugment with parameters ??
for ts_g in lts_g repeat
  lbwt := cons([false, ts_g]$BWT, lbwt)
  h := lazyPquo(ts_v, g)
  (ground? h) or (mvar(h) < v) => "leave"
  h := mainPrimitivePart h
  lts_h := augment(h, ts)$TS
  lts_h := augment(members(ts_v_+), lts_h)$TS
  -- USE stoseInternalAugment with parameters ??
  for ts_h in lts_h repeat
    lbwt := cons([true, ts_h]$BWT, lbwt)
    sort((x, y) +-> x.val < y.val, lbwt)

  -- iprint("*")$iprintpack
  k: KeyInvSet := [p, ts]
  e := extractIfCan(k)$HInvSet
  e case EntryInvSet => e::EntryInvSet
  q := primitivePart initiallyReduce(p, ts)
  zero? q => []
  normalized?(q, ts) => [ts]
  v := mvar(q)
  toSave: Split := []
  not algebraic?(v, ts) =>
  toCheck: List BWT := stoseInvertible?_sqfreg(init(q), ts)@(List BWT)
  for bwt in toCheck repeat
    bwt.val => toSave := cons(bwt.tower, toSave)
    newq := removeZero(q, bwt.tower)
    zero? newq => "leave"
    toSave := concat(stoseInvertibleSet_sqfreg(newq, bwt.tower), toSave)
  toSave := removeDuplicates toSave
  return algebraicSort(toSave)$quasicomppack

for gwt in lgwt repeat
  g := gwt.val; ts := gwt.tower
  (ground? g) or (mvar(g) < v) =>
    lts := augment(ts_v, ts)$TS
    lts := augment(members(ts_v_+), lts)$TS
    toSave := concat(lts, toSave)
  g := mainPrimitivePart g
  h := lazyPquo(ts_v, g)
  h := mainPrimitivePart h
  (ground? h) or (mvar(h) < v) => "leave"
  lts_h := augment(h, ts)$TS
  lts_h := augment(members(ts_v_+), lts_h)$TS
  toSave := concat(lts_h, toSave)
  toSave := algebraicSort(toSave)$quasicomppack
\begin{verbatim}
insert!(k, toSave)$HInvSet
toSave

stoseInvertible?_reg(p:P, ts:TS): List BWT ==
  -- iprint("-")$iprintpack
  q := primitivePart initiallyReduce(p, ts)
  zero? q => [[false, ts]$BWT]
  normalized?(q, ts) => [[true, ts]$BWT]
  v := mvar(q)
  not algebraic?(v, ts) =>
    lbwt: List BWT := []
    toCheck: List BWT := stoseInvertible?_reg(init(q), ts)@(List BWT)
    for bwt in toCheck repeat
      bwt.val => lbwt := cons(bwt, lbwt)
      newq := removeZero(q, bwt.tower)
      zero? newq => lbwt := cons(bwt, lbwt)
      lbwt :=
        concat(stoseInvertible?_reg(newq, bwt.tower)@(List BWT), lbwt)
    return lbwt

  ts_v := select(ts, v)::P
  ts_v_- := collectUnder(ts, v)
  ts_v_+ := collectUpper(ts, v)
  lgwt := stoseInternalLastSubResultant(ts_v, q, ts_v_-, false, false)
  lbwt: List BWT := []
  lts, lts_g, lts_h: Split
  for gwt in lgwt repeat
    g := gwt.val; ts := gwt.tower
    (ground? g) or (mvar(g) < v) =>
      lts := augment(ts_v, ts)$TS
      lts := augment(members(ts_v_+), lts)$TS
      for ts in lts repeat
        lbwt := cons([[true, ts]$BWT], lbwt)
    g := mainPrimitivePart g
    lts_g := augment(g, ts)$TS
    lts_g := augment(members(ts_v_+), lts_g)$TS
    -- USE internalAugment with parameters ??
    for ts_g in lts_g repeat
      lbwt := cons([[false, ts_g]$BWT], lbwt)
  h := lazyPquo(ts_v, g)
  (ground? h) or (mvar(h) < v) => "leave"
  h := mainPrimitivePart h
  lts_h := augment(h, ts)$TS
  lts_h := augment(members(ts_v_+), lts_h)$TS
  -- USE internalAugment with parameters ??
  for ts_h in lts_h repeat
    inv := stoseInvertible?_reg(q, ts_h)@(List BWT)
    lbwt := concat([bwt for bwt in inv | bwt.val], lbwt)
  sort((x, y) +-> x.val < y.val, lbwt)

\end{verbatim}
--iprint("/")$iprintpack
k: KeyInvSet := [p,ts]
e := extractIfCan(k)$HInvSet
e case EntryInvSet => e::EntryInvSet
q := primitivePart initiallyReduce(p,ts)
zero? q => []
normalized?(q,ts) => [ts]
v := mvar(q)
toSave: Split := []
not algebraic?(v,ts) =>
toCheck: List BWT := stoseInvertible?_reg(init(q),ts)@(List BWT)
for bwt in toCheck repeat
  bwt.val => toSave := cons(bwt.tower,toSave)
  newq := removeZero(q,bwt.tower)
  zero? newq => "leave"
  toSave := concat(stoseInvertibleSet_reg(newq,bwt.tower), toSave)
toSave := removeDuplicates toSave
return algebraicSort(toSave)$quasicomppack

ts_v := select(ts,v)::P
ts_v_- := collectUnder(ts,v)
ts_v_+ := collectUpper(ts,v)
lgwt := stoseInternalLastSubResultant(ts_v,q,ts_v_-,false,false)
lts, lts_h: Split
for gwt in lgwt repeat
  g := gwt.val; ts := gwt.tower
  (ground? g) or (mvar(g) < v) =>
lts := augment(ts_v,ts)$TS
  lts := augment(members(ts_v_+),lts)$TS
  toSave := concat(lts,toSave)
g := mainPrimitivePart g
h := lazyPquo(ts_v,g)
h := mainPrimitivePart h
(ground? h) or (mvar(h) < v) => "leave"
lts_h := augment(h,ts)$TS
lts_h := augment(members(ts_v_+),lts_h)$TS
for ts_h in lts_h repeat
  inv := stoseInvertibleSet_reg(q,ts_h)
  toSave := removeDuplicates concat(inv,toSave)
toSave := algebraicSort(toSave)$quasicomppack
insert!(k,toSave)$HInvSet
toSave

if TS has SquareFreeRegularTriangularSetCategory(R,E,V,P) then
else
stoseInvertible?\(p: P, ts: TS\): List BWT == stoseInvertible?_reg\(p, ts\)

stoseInvertibleSet\(p: P, ts: TS\): Split == stoseInvertibleSet_reg\(p, ts\)

---

--- SFRGCD.dotabb ---

"SFRGCD" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SFRGCD"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"SFRGCD" -> "SFRTCAT"

---

package MATSTOR StorageEfficientMatrixOperations

--- StorageEfficientMatrixOperations.input ---

)set break resume
)sys rm -f StorageEfficientMatrixOperations.output
)spool StorageEfficientMatrixOperations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StorageEfficientMatrixOperations
--R
--R StorageEfficientMatrixOperations\(R: \text{Ring}\) is a package constructor
--R Abbreviation for StorageEfficientMatrixOperations is MATSTOR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for MATSTOR
--R
--R------------------------------ Operations --------------------------------
--R \(\otimes\) : \(\text{Matrix}(R), \text{NonNegativeInteger}\) -> \text{Matrix}(R)
--R copy! : \(\text{Matrix}(R), \text{Matrix}(R)\) -> \text{Matrix}(R)
--R leftScalarTimes! : \(\text{Matrix}(R), R, \text{Matrix}(R)\) -> \text{Matrix}(R)
--R minus! : \(\text{Matrix}(R), \text{Matrix}(R)\) -> \text{Matrix}(R)
--R minus! : \(\text{Matrix}(R), \text{Matrix}(R), \text{Matrix}(R)\) -> \text{Matrix}(R)
--R plus! : \(\text{Matrix}(R), \text{Matrix}(R), \text{Matrix}(R)\) -> \text{Matrix}(R)
--R power! : \(\text{Matrix}(R), \text{Matrix}(R), \text{Matrix}(R), \text{Matrix}(R), \text{NonNegativeInteger}\) -> \text{Matrix}(R)
--R rightScalarTimes! : \(\text{Matrix}(R), \text{Matrix}(R), \text{Matrix}(R), R\) -> \text{Matrix}(R)
--R times! : \(\text{Matrix}(R), \text{Matrix}(R), \text{Matrix}(R)\) -> \text{Matrix}(R)
This package provides standard arithmetic operations on matrices. The functions in this package store the results of computations in existing matrices, rather than creating new matrices. This package works only for matrices of type Matrix and uses the internal representation of this type.

See Also:
o )show StorageEfficientMatrixOperations
CHAPTER 20. CHAPTER S

\abbrev package MATSTOR StorageEfficientMatrixOperations
++ Author: Clifton J. Williamson
++ Date Created: 18 July 1990
++ Date Last Updated: 18 July 1990
++ Description:
++ This package provides standard arithmetic operations on matrices.
++ The functions in this package store the results of computations
++ in existing matrices, rather than creating new matrices. This
++ package works only for matrices of type Matrix and uses the
++ internal representation of this type.

StorageEfficientMatrixOperations(R): Exports == Implementation where
R : Ring
M ==> Matrix R
NNI ==> NonNegativeInteger
ARR ==> PrimitiveArray R
REP ==> PrimitiveArray PrimitiveArray R

Exports ==> with
  copy_! : (M,M) -> M
    ++ \spad{copy!(c,a)} copies the matrix \spad{a} into the matrix c.
    ++ Error: if \spad{a} and c do not have the same
    ++ dimensions.
  plus_! : (M,M,M) -> M
    ++ \spad{plus!(c,a,b)} computes the matrix sum \spad{a + b} and stores the
    ++ result in the matrix c.
    ++ Error: if \spad{a}, b, and c do not have the same dimensions.
  minus_! : (M,M) -> M
    ++ \spad{minus!(c,a)} computes \spad{-a} and stores the result in the
    ++ matrix c.
    ++ Error: if a and c do not have the same dimensions.
  minus_! : (M,M,M) -> M
    ++ \spad{minus!(c,a,b)} computes the matrix difference \spad{a - b}
    ++ and stores the result in the matrix c.
    ++ Error: if \spad{a}, b, and c do not have the same dimensions.
  leftScalarTimes_! : (M,R,M) -> M
    ++ \spad{leftScalarTimes!(c,r,a)} computes \spad{r * a} and stores the result in the
    ++ matrix c.
    ++ Error: if \spad{a} and c do not have the same dimensions.
  rightScalarTimes_! : (M,M,R) -> M
    ++ \spad{rightScalarTimes!(c,a,r)} computes the scalar product
    ++ \spad{r * a} and stores the result in the matrix c.
    ++ Error: if \spad{a} and c do not have the same dimensions.
  times_! : (M,M,M) -> M
    ++ \spad{times!(c,a,b)} computes the matrix product \spad{a * b}
    ++ and stores the result in the matrix c.
    ++ Error: if \spad{a}, b, and c do not have
    ++ compatible dimensions.
  power_! : (M,M,M,NNI) -> M
    ++ \spad{power!(a,b,c,m,n)} computes \spad{m ** n} and stores the result in
++ \spad{a}. The matrices \spad{b} and \spad{c} are used to store intermediate results.
++ Error: if \spad{a}, \spad{b}, \spad{c}, and \spad{m} are not square
++ and of the same dimensions.

```spad
"**" : (M,NNI) -> M
++ \spad{\text{x ** n}} computes the n-th power
++ of a square matrix. The power \text{n} is assumed greater than 1.
```

Implementation ==> add

```spad
rep : M -> REP
rep m == m pretend REP

\text{copy!} (c,a) ==
\text{m := nrows a; n := ncols a}
\text{not((nrows c) = m and (ncols c) = n) =>}
\text{error "copy!: matrices of incompatible dimensions"}
\text{aa := rep a; cc := rep c}
\text{for i in 0..(m-1) repeat}
\text{aRow := qelt(aa,i); cRow := qelt(cc,i)}
\text{for j in 0..(n-1) repeat}
\text{qsetelt_!(cRow,j,qelt(aRow,j))}
\text{c}

\text{plus!} (c,a,b) ==
\text{m := nrows a; n := ncols a}
\text{not((nrows b) = m and (ncols b) = n) =>}
\text{error "plus!: matrices of incompatible dimensions"}
\text{not((nrows c) = m and (ncols c) = n) =>}
\text{error "plus!: matrices of incompatible dimensions"}
\text{aa := rep a; bb := rep b; cc := rep c}
\text{for i in 0..(m-1) repeat}
\text{aRow := qelt(aa,i); bRow := qelt(bb,i); cRow := qelt(cc,i)}
\text{for j in 0..(n-1) repeat}
\text{qsetelt_!(cRow,j,qelt(aRow,j) + qelt(bRow,j))}
\text{c}

\text{minus!} (c,a) ==
\text{m := nrows a; n := ncols a}
\text{not((nrows c) = m and (ncols c) = n) =>}
\text{error "minus!: matrices of incompatible dimensions"}
\text{aa := rep a; cc := rep c}
\text{for i in 0..(m-1) repeat}
\text{aRow := qelt(aa,i); cRow := qelt(cc,i)}
\text{for j in 0..(n-1) repeat}
\text{qsetelt_!(cRow,j,-qelt(aRow,j))}
\text{c}

\text{minus!} (c,a,b) ==
\text{m := nrows a; n := ncols a}
\text{not((nrows b) = m and (ncols b) = n) =>}
CHAPTER 20. CHAPTER S

error "minus!: matrices of incompatible dimensions"
not((nrows c) = m and (ncols c) = n) =>
  error "minus!: matrices of incompatible dimensions"
aa := rep a; bb := rep b; cc := rep c
for i in 0..(m-1) repeat
  aRow := qelt(aa,i); bRow := qelt(bb,i); cRow := qelt(cc,i)
  for j in 0..(n-1) repeat
    qsetelt_!(cRow,j,qelt(aRow,j) - qelt(bRow,j))
c
leftScalarTimes_!(c,r,a) ==
  m := nrows a; n := ncols a
  not((nrows c) = m and (ncols c) = n) =>
    error "leftScalarTimes!: matrices of incompatible dimensions"
aa := rep a; cc := rep c
  for i in 0..(m-1) repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
      qsetelt_!(cRow,j,r * qelt(aRow,j))
c
rightScalarTimes_!(c,a,r) ==
  m := nrows a; n := ncols a
  not((nrows c) = m and (ncols c) = n) =>
    error "rightScalarTimes!: matrices of incompatible dimensions"
aa := rep a; cc := rep c
  for i in 0..(m-1) repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    for j in 0..(n-1) repeat
      qsetelt_!(cRow,j,qelt(aRow,j) * r)
c
copyCol_!: (ARR,REP,Integer,Integer) -> ARR
  copyCol_!(bCol,bb,j,n1) ==
    for i in 0..n1 repeat qsetelt_!(bCol,i,qelt(qelt(bb,i),j))
times_!(c,a,b) ==
  m := nrows a; n := ncols a; p := ncols b
  not((nrows b) = n and (nrows c) = m and (ncols c) = p) =>
    error "times!: matrices of incompatible dimensions"
aa := rep a; bb := rep b; cc := rep c
  bCol : ARR := new(n,0)
  m1 := (m :: Integer) - 1; n1 := (n :: Integer) - 1
  for j in 0..(p-1) repeat
    copyCol_!(bCol,bb,j,n1)
  for i in 0..m1 repeat
    aRow := qelt(aa,i); cRow := qelt(cc,i)
    sum : R := 0
    for k in 0..n1 repeat
      sum := sum + qelt(aRow,k) * qelt(bCol,k)
qsetelt_!(cRow,j,sum)
c

\text{power}_!(a,b,c,m,p) ==
\text{mm} := \text{nrows } a; \text{nn} := \text{ncols } a
\text{not} (\text{mm} = \text{nn}) =>
\hspace{1em} \text{error "power!: matrix must be square"}
\text{not} ((\text{nrows } b) = \text{mm} \text{ and } (\text{ncols } b) = \text{nn}) =>
\hspace{1em} \text{error "power!: matrices of incompatible dimensions"}
\text{not} ((\text{nrows } c) = \text{mm} \text{ and } (\text{ncols } c) = \text{nn}) =>
\hspace{1em} \text{error "power!: matrices of incompatible dimensions"}
\text{not} ((\text{nrows } m) = \text{mm} \text{ and } (\text{ncols } m) = \text{nn}) =>
\hspace{1em} \text{error "power!: matrices of incompatible dimensions"}
\text{flag} := \text{false}
\text{copy}_!(b,m)
\text{repeat}
\hspace{1em} \text{if odd? } p \text{ then}
\hspace{2em} \text{flag} =>
\hspace{3em} \text{times}_!(c,b,a)
\hspace{3em} \text{copy}_!(a,c)
\hspace{3em} \text{flag} := \text{true}
\hspace{3em} \text{copy}_!(a,b)
\hspace{1em} \text{-- one? } p => \text{return } a
\hspace{1em} (p = 1) => \text{return } a
\hspace{1em} p := p \text{ quo } 2
\hspace{1em} \text{times}_!(c,b,b)
\hspace{1em} \text{copy}_!(b,c)

\text{m} ** \text{n} ==
\text{not square? } m \Rightarrow \text{error "**: matrix must be square"}
a := \text{copy } m; b := \text{copy } m; c := \text{copy } m
\text{power}_!(a,b,c,m,n)
package STREAM1 StreamFunctions1

— StreamFunctions1.input —

)set break resume
)sys rm -f StreamFunctions1.output
)spool StreamFunctions1.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamFunctions1
--R
--R StreamFunctions1(S: Type) is a package constructor
--R Abbreviation for StreamFunctions1 is STREAM1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STREAM1
--R
--R----------------------------------- Operations -----------------------------------
--R concat : Stream(Stream(S)) -> Stream(S)
--R
--Z 1

)spool
)lisp (bye)

— StreamFunctions1.help —

====================================================================
StreamFunctions1 examples
====================================================================

Functions defined on streams with entries in one set.

See Also:
  o )show StreamFunctions1
StreamFunctions1 (STREAM1)

Exports:
concat

--- package STREAM1 StreamFunctions1 ---

)abbrev package STREAM1 StreamFunctions1
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Description:
++ Functions defined on streams with entries in one set.

StreamFunctions1(S:Type): Exports == Implementation where
ST  ==> Stream
Exports ==> with
concat: ST ST S -> ST S
  ++ concat(u) returns the left-to-right concatenation of the
  ++ streams in u. Note that \spad{concat(u) = reduce(concat,u)}.
  ++
  ++X m:=\[i for i in 10..]    
  ++X n:=\[j for j in 1.. | prime? j]  
  ++X p:=\[m,n]:Stream(Stream(PositiveInteger))  
  ++X concat(p)
Implementation ==> add
concat z == delay
  empty? z => empty()
  empty?(x := frst z) => concat rst z
  concat(frst x,concat(rst x,concat rst z))
package STREAM2 StreamFunctions2

--- StreamFunctions2.input ---

)set break resume
)sys rm -f StreamFunctions2.output
)spool StreamFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamFunctions2
--R
--R StreamFunctions2(A: Type,B: Type) is a package constructor
--R Abbreviation for StreamFunctions2 is STREAM2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STREAM2
--R
--R------------------------------- Operations --------------------------------
--R map : ((A -> B),Stream(A)) -> Stream(B)
--R reduce : (B,((A,B) -> B),Stream(A)) -> B
--R scan : (B,((A,B) -> B),Stream(A)) -> Stream(B)
--R
--E 1

)spool
)lisp (bye)

--- StreamFunctions2.help ---

====================================================================
StreamFunctions2 examples
====================================================================
Functions defined on streams with entries in two sets.

See Also:
- )show StreamFunctions2

StreamFunctions2 (STREAM2)

Exports:
- map
- reduce
- scan

--- package STREAM2 StreamFunctions2 ---

)abbrev package STREAM2 StreamFunctions2
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Description:
++ Functions defined on streams with entries in two sets.

StreamFunctions2(A:Type,B:Type): Exports == Implementation where
  ST   => Stream

Exports => with
  map: ((A -> B),ST A) -> ST B
  ++ map(f,s) returns a stream whose elements are the function f applied
  ++ to the corresponding elements of s.
  ++ Note that \spad{map(f,[x0,x1,x2,...]) = [f(x0),f(x1),f(x2),...]}.
  ++
  ++X m:=[i for i in 1..]
  ++X f(i:PositiveInteger):PositiveInteger==i**2
+++ map(f,m)

scan: (B,((A,B) -> B),ST A) -> ST B
++ scan(b,h,[x0,x1,x2,...]) returns \( \text{spad}\{y0,y1,y2,...\} \), where
++ \( \text{spad}\{y0 = h(x0,b)\}, \)
++ \( \text{spad}\{y1 = h(x1,y0)\}, \text{spad}\{\ldots\} \)
++ \( \text{spad}\{yn = h(xn,y(n-1))\}. \)
++
++X m:=[i for i in 1..]::Stream(Integer)
++X f(i:Integer,j:Integer):Integer==i+j
++X scan(1,f,m)

reduce: (B,(A,B) -> B),ST A) -> B
++ reduce(b,f,u), where u is a finite stream \( \text{spad}\{x0,x1,...,xn\} \),
++ returns the value \( \text{spad}\{r(n)\} \) computed as follows:
++ \( \text{spad}\{r0 = f(x0,b)\}, \)
++ \( r1 = f(x1,r0),\ldots, \)
++ \( r(n) = f(xn,r(n-1))\}. \)
++
++X m:=[i for i in 1..300]::Stream(Integer)
++X f(i:Integer,j:Integer):Integer==i+j
++X reduce(1,f,m)

-- rreduce: (B,(A,B) -> B),ST A) -> B
-- ++ reduce(b,h,[x0,x1,...,xn]) = h(x1,h(x2(...,h(x(n-1),h(xn,b))..))
-- reshape: (ST B,ST A) -> ST B
-- ++ reshape(y,x) = y

Implementation ==> add

mapp: (A -> B,ST A) -> ST B
mapp(f,x)== delay
  empty? x => empty()
  concat(f frst x, map(f,rst x))

map(f,x) ==
  explicitlyEmpty? x => empty()
  eq?(x,rst x) => repeating([f frst x])
  mapp(f, x)

-- reshape(y,x) == y

scan(b,h,x) == delay
  empty? x => empty()
  c := h(frst x,b)
  concat(c,scan(c,h,rst x))

reduce(b,h,x) ==
  empty? x => b
  reduce(h(frst x,b),h,rst x)
-- rreduce(b,h,x) ==
-- empty? x => b
-- h(frst x,rreduce(b,h,rst x))

---

-- STREAM2.dotabb ---

"STREAM2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STREAM2"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"STREAM2" -> "TYPE"

---

package STREAM3 StreamFunctions3

--- StreamFunctions3.input ---

)set break resume
)sys rm -f StreamFunctions3.output
)spool StreamFunctions3.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamFunctions3

--R StreamFunctions3(A: Type,B: Type,C: Type) is a package constructor
--R Abbreviation for StreamFunctions3 is STREAM3
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STREAM3
--R
--R-------------------------------- Operations --------------------------------
--R map : (((A,B) -> C),Stream(A),Stream(B)) -> Stream(C)
--R
--E 1

)spool
)lisp (bye)

---

--- StreamFunctions3.help ---
StreamFunctions3 examples

Functions defined on streams with entries in three sets.

See Also:
  o )show StreamFunctions3

StreamFunctions3 (STREAM3)

Exports:
  map

— package STREAM3 StreamFunctions3 —

)abbrev package STREAM3 StreamFunctions3
++ Authors: Burge, Watt; updated by Clifton J. Williamson
++ Date Created: July 1986
++ Date Last Updated: 29 January 1990
++ Description:
++ Functions defined on streams with entries in three sets.

StreamFunctions3(A,B,C): Exports == Implementation where
  A : Type
  B : Type
  C : Type
  ST ==> Stream

Exports ==> with
  map: ((A,B) -> C,ST A,ST B) -> ST C
++ map(f,st1,st2) returns the stream whose elements are the
++ function f applied to the corresponding elements of st1 and st2.
++ \spad{map(f,[x0,x1,x2,...],[y0,y1,y2,...]) = [f(x0,y0),f(x1,y1),...]}.
++
++ S
++ X m:=[i for i in 1..]::Stream(Integer)
++ X n:=[i for i in 1..]::Stream(Integer)
++ X f(i:Integer,j:Integer):Integer == i+j
++ X map(f,m,n)

Implementation ==> add

mapp:((A,B) -> C,ST A,ST B) -> ST C
mapp(g,x,y) == delay
   empty? x or empty? y => empty()
   concat(g(frst x,frst y), map(g,rst x,rst y))

map(g,x,y) ==
   explicitlyEmpty? x => empty()
   eq?(x,rst x) => map(z +-> g(frst x,z)$StreamFunctions2(B,C)
   explicitlyEmpty? y => empty()
   eq?(y,rst y) => map(z +-> g(z,frst y),x)$StreamFunctions2(A,C)
   mapp(g,x,y)

— STREAM3.dotabb —

"STREAM3" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STREAM3"]
"TYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=TYPE"]
"STREAM3" -> "TYPE"

— package STINPROD StreamInfiniteProduct —

)set break resume
)sys rm -f StreamInfiniteProduct.output
)spool StreamInfiniteProduct.output
)set message test on
)set message auto off
)clear all
---S 1 of 1
)show StreamInfiniteProduct
--R
--R StreamInfiniteProduct(Coef: Join(IntegralDomain,CharacteristicZero)) is a package constructor
--R Abbreviation for StreamInfiniteProduct is STINPROD
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STINPROD
--R
--R---------------------------------------------------------- Operations --------------------------------
--R evenInfiniteProduct : Stream(Coef) -> Stream(Coef)
--R generalInfiniteProduct : (Stream(Coef),Integer,Integer) -> Stream(Coef)
--R infiniteProduct : Stream(Coef) -> Stream(Coef)
--R oddInfiniteProduct : Stream(Coef) -> Stream(Coef)
--R
--E 1

)spool
)lisp (bye)

— StreamInfiniteProduct.help —

====================================================================
StreamInfiniteProduct examples
====================================================================

This package computes infinite products of Taylor series over an
integral domain of characteristic 0. Here Taylor series are
represented by streams of Taylor coefficients.

See Also:
o )show StreamInfiniteProduct

———
StreamInfiniteProduct (STINPROD)

Exports:
- evenInfiniteProduct
- generalInfiniteProduct
- infiniteProduct
- oddInfiniteProduct

--- package STINPROD StreamInfiniteProduct ---

)abbrev package STINPROD StreamInfiniteProduct
++ Author: Clifton J. Williamson
++ Date Created: 23 February 1990
++ Date Last Updated: 23 February 1990
++ Description:
++ This package computes infinite products of Taylor series over an
++ integral domain of characteristic 0. Here Taylor series are
++ represented by streams of Taylor coefficients.

StreamInfiniteProduct(Coef):Exports == Implementation where
  Coef:Join(IntegralDomain,CharacteristicZero)
  I ==> Integer
  QF ==> Fraction
  ST ==> Stream

Exports ==> with

  infiniteProduct:ST Coef -> ST Coef
  ++ infiniteProduct(f(x)) computes \(\prod(n=1,2,3...,f(x^n))\).
  ++ The series \(f(x)\) should have constant coefficient 1.
  evenInfiniteProduct:ST Coef -> ST Coef
  ++ evenInfiniteProduct(f(x)) computes \(\prod(n=2,4,6...,f(x^n))\).
  ++ The series \(f(x)\) should have constant coefficient 1.
  oddInfiniteProduct:ST Coef -> ST Coef
  ++ oddInfiniteProduct(f(x)) computes \(\prod(n=1,3,5...,f(x^n))\).
  ++ The series \(f(x)\) should have constant coefficient 1.
  generalInfiniteProduct: (ST Coef,I,I) -> ST Coef
  ++ generalInfiniteProduct(f(x),a,d) computes
  ++ \(\prod(n=a,a+d,a+2*d,...,f(x^n))\).
CHAPTER 20. CHAPTER S

++ The series \spad{f(x)} should have constant coefficient 1.

Implementation ==> add

if Coef has Field then

import StreamTaylorSeriesOperations(Coef)
import StreamTranscendentalFunctions(Coef)

infiniteProduct st == exp lambert log st
evenInfiniteProduct st == exp evenlambert log st
oddInfiniteProduct st == exp oddlambert log st
generalInfiniteProduct(st,a,d) == exp generalLambert(log st,a,d)

else

import StreamTaylorSeriesOperations(QF Coef)
import StreamTranscendentalFunctions(QF Coef)

applyOverQF:(ST QF Coef -> ST QF Coef,ST Coef) -> ST Coef
applyOverQF(f,st) ==
    stQF := map(z1 +-> z1::QF(Coef),st)$StreamFunctions2(Coef,QF Coef)
    map(z1 +-> retract(z1)@Coef,f stQF)$StreamFunctions2(QF Coef,Coef)

infiniteProduct st == applyOverQF(z1 +-> exp lambert log z1,st)
evenInfiniteProduct st == applyOverQF(z1 +-> exp evenlambert log z1,st)
oddInfiniteProduct st == applyOverQF(z1 +-> exp oddlambert log z1,st)
generalInfiniteProduct(st,a,d) ==
    applyOverQF(z1 +-> exp generalLambert(log z1,a,d),st)

package STTAYLOR StreamTaylorSeriesOperations

— StreamTaylorSeriesOperations.input —

)set break resume
package STTAYLOR STREAMTAYLOR SERIES OPERATIONS

)sys rm -f StreamTaylorSeriesOperations.output
)spool StreamTaylorSeriesOperations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamTaylorSeriesOperations

--R
--R StreamTaylorSeriesOperations(A: Ring) is a package constructor
--R Abbreviation for StreamTaylorSeriesOperations is STTAYLOR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STTAYLOR
--R
--R---------------------------------- Operations ---------------------------------
--R ?*? : (A,Stream(A)) -> Stream(A)  ?*? : (Stream(A),A) -> Stream(A)
--R +  : Stream(A) -> Stream(A)  coerce : A -> Stream(A)
--R deriv : Stream(A) -> Stream(A)  eval : (Stream(A),A) -> Stream(A)
--R evenlambert : Stream(A) -> Stream(A)  int : A -> Stream(A)
--R lagrange : Stream(A) -> Stream(A)  lambert : Stream(A) -> Stream(A)
--R monom : (A,Integer) -> Stream(A)  oddlambert : Stream(A) -> Stream(A)
--R revert : Stream(A) -> Stream(A)
--R ?*? : (Stream(A),Stream(A)) -> Stream(A)
--R ?+? : (Stream(A),Stream(A)) -> Stream(A)
--R /? : (Stream(A),Stream(A)) -> Stream(A)
--R addiag : Stream(Stream(A)) -> Stream(A)
--R compose : (Stream(A),Stream(A)) -> Stream(A)
--R exquo : (Stream(A),Stream(A)) -> Stream(A)  "failed"
--R gderiv : ((Integer -> A),Stream(A)) -> Stream(A)
--R generalLambert : (Stream(A),Integer,Integer) -> Stream(A)
--R integers : Integer -> Stream(Integer)
--R integrate : (A,Stream(A)) -> Stream(A) if A has ALGEBRA(FRAC(INT))
--R invmultisect : (Integer,Integer,Stream(A)) -> Stream(A)
--R lazyGintegrate : ((Integer -> A),A,()) -> Stream(A) if A has FIELD
--R lazyIntegrate : (A,()) -> Stream(A)) -> Stream(A) if A has ALGEBRA(FRAC(INT))
--R mapdiv : (Stream(A),Stream(A)) -> Stream(A) if A has FIELD
--R mapmult : (Stream(A),Stream(A)) -> Stream(A)
--R multisect : (Integer,Integer,Stream(A)) -> Stream(A)
--R nlde : Stream(Stream(A)) -> Stream(A) if A has ALGEBRA(FRAC(INT))
--R oddintegers : Integer -> Stream(Integer)
--R power : (A,Stream(A)) -> Stream(A) if A has FIELD
--R powern : (Fraction(Integer),Stream(A)) -> Stream(A) if A has ALGEBRA(FRAC(INT))
--R recip : Stream(A) -> Stream(A) if A has FIELD
--R
--E 1

)spool
)lisp (bye)
StreamTaylorSeriesOperations implements Taylor series arithmetic, where a Taylor series is represented by a stream of its coefficients.

Problems raising a UTS to a negative integer power.

The code in powern(rn,x) which raises an unnecessary error where no distinction between rational and integer powers are made.

The fix is easy. Since the problem does not exist in SUPS we can just take the definition there.

See Also:

- )show StreamTaylorSeriesOperations

StreamTaylorSeriesOperations (STTAYLOR)
Exports:

- `addiag`
- `coerce`
- `compose`
- `deriv`
- `eval`
- `evenlambert`
- `exquo`
- `gderiv`
- `generalLambert`
- `int`
- `integrates`
- `lagrange`
- `lambert`
- `lazyGintegrate`
- `lazyIntegrate`
- `mapdiv`
- `mapmult`
- `monom`
- `multisect`
- `nlde`
- `oddlambert`
- `oddlintegers`
- `power`
- `powern`
- `recip`
- `revert`
- `?*?`
- `?+?`
- `?-?`
- `?/?

— package STTAYLOR StreamTaylorSeriesOperations —

)abbrev package STTAYLOR StreamTaylorSeriesOperations
++ Author: William Burge, Stephen Watt, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 26 May 1994
++ Description:
++ StreamTaylorSeriesOperations implements Taylor series arithmetic,
++ where a Taylor series is represented by a stream of its coefficients.

StreamTaylorSeriesOperations(A):Exports == Implementation where
  A : Ring
  RN == Fraction Integer
  I == Integer
  NNI == NonNegativeInteger
  ST == Stream
  SP2 == StreamFunctions2
  SP3 == StreamFunctions3
  L == List
  LA == List A
  YS == Y$ParadoxicalCombinatorsForStreams(A)
  UN == Union(ST A, "failed")
Exports == with
  "+" : (ST A,ST A) -> ST A
  ++ a + b returns the power series sum of \spad{a} and \spad{b}:
  ++ \spad{[a0,a1,...] + [b0,b1,...] = [a0 + b0,a1 + b1,...]}\n  "-" : (ST A,ST A) -> ST A
  ++ a - b returns the power series difference of \spad{a} and \spad{b}:
  ++ \spad{[a0,a1,...] - [b0,b1,...] = [a0 - b0,a1 - b1,...]}\n  "-" : ST A -> ST A
  ++ - a returns the power series negative of \spad{a}:
  ++ \spad{- [a0,a1,...] = [- a0,- a1,...]}\n  "*" : (ST A,ST A) -> ST A
  ++ a * b returns the power series (Cauchy) product of \spad{a} and b:
  ++ \spad{[a0,a1,...]*[b0,b1,...] = [c0,c1,...]} where
  ++ \spad{ck = \sum(i + j = k,ai * bk)}.\n  "*" : (A,ST A) -> ST A
  ++ r * a returns the power series scalar multiplication of r by \spad{a}:
  ++ \spad{r*[a0,a1,...] = [r*a0,r*a1,...]}\n  "*" : (ST A,A) -> ST A
++ a * r returns the power series scalar multiplication of \spad{a} by r:
++ \spad{[a0,a1,...] * r = [a0 * r,a1 * r,...]}
"exquo" : (ST A,ST A) -> Union(ST A,"failed")
++ exquo(a,b) returns the power series quotient of \spad{a} by b,
++ if the quotient exists, and "failed" otherwise
"/"  : (ST A,ST A) -> ST A
++ a / b returns the power series quotient of \spad{a} by b.
++ An error message is returned if \spad{b} is not invertible.
++ This function is used in fixed point computations.
recip  : ST A -> UN
++ recip(a) returns the power series reciprocal of \spad{a}, or
++ "failed" if not possible.
monom  : (A,I) -> ST A
++ monom(deg,coef) is a monomial of degree deg with coefficient
++ coef.
integers  : I -> ST I
++ integers(n) returns \spad{[n,n+1,n+2,...]}.
oddintegers  : I -> ST I
++ oddintegers(n) returns \spad{[n,n+2,n+4,...]}.
int  : A -> ST A
++ int(r) returns \spad{[r,r+1,r+2,...]}, where r is a ring element.
mapmult  : (ST A,ST A) -> ST A
++ mapmult([a0,a1,..],[b0,b1,..])
++ returns \spad{[a0*b0,a1*b1,..]}.
deriv  : ST A -> ST A
++ deriv(a) returns the derivative of the power series with
++ respect to the power series variable. Thus
++ \spad{deriv([a0,a1,a2,...])} returns \spad{[a1,2 a2,3 a3,...]}.
gderiv  : (I -> A,ST A) -> ST A
++ gderiv(f,[a0,a1,a2,...]) returns
++ \spad{[f(0)*a0,f(1)*a1,f(2)*a2,..]}.
coerce  : A -> ST A
++ coerce(r) converts a ring element r to a stream with one element.
eval  : (ST A,A) -> ST A
++ eval(a,r) returns a stream of partial sums of the power series
++ \spad{a} evaluated at the power series variable equal to r.
compose  : (ST A,ST A) -> ST A
++ compose(a,b) composes the power series \spad{a} with
++ the power series b.
lagrange  : ST A -> ST A
++ lagrange(g) produces the power series for f where f is
++ implicitly defined as \spad{f(z) = z*g(f(z))}.
revert  : ST A -> ST A
++ revert(a) computes the inverse of a power series \spad{a}
++ with respect to composition.
++ the series should have constant coefficient 0 and first
++ order coefficient 1.
addiag  : ST ST A -> ST A
++ addiag(x) performs diagonal addition of a stream of streams. if x =
++ \spad{[[a<0,0>,a<0,1>,..],[a<1,0>,a<1,1>,..],[a<2,0>,a<2,1>,..],...]}
++ and \texttt{addiag(x)} = [b<0,b<1>,...], then b<k> = \sum(i+j=k,a<i,j>).

\texttt{lambert} : \texttt{ST A \rightarrow ST A}
++ \texttt{lambert(st)} computes \texttt{\{f(x) + f(x**2) + f(x**3) + \ldots\}}
++ if \texttt{st} is a stream representing \texttt{\{f(x)\}}.
++ This function is used for computing infinite products.
++ If \texttt{\{f(x)\}} is a power series with constant coefficient 1 then
++ \texttt{\prod(f(x**n),n = 1..infinity) = exp(lambert(log(f(x))))}.

\texttt{oddlambert} : \texttt{ST A \rightarrow ST A}
++ \texttt{oddlambert(st)} computes \texttt{\{f(x) + f(x**3) + f(x**5) + \ldots\}}
++ if \texttt{st} is a stream representing \texttt{\{f(x)\}}.
++ This function is used for computing infinite products.
++ If \texttt{f(x)} is a power series with constant coefficient 1 then
++ \texttt{\prod(f(x**(2*n-1)),n=1..infinity) = exp(oddlambert(log(f(x))))}.

\texttt{evenlambert} : \texttt{ST A \rightarrow ST A}
++ \texttt{evenlambert(st)} computes \texttt{\{f(x**2) + f(x**4) + f(x**6) + \ldots\}}
++ if \texttt{st} is a stream representing \texttt{\{f(x)\}}.
++ This function is used for computing infinite products.
++ If \texttt{\{f(x)\}} is a power series with constant coefficient 1 then
++ \texttt{\prod(f(x**(2*n)),n=1..infinity) = exp(evenlambert(log(f(x))))}.

\texttt{generalLambert} : \texttt{(ST A,I,I) \rightarrow ST A}
++ \texttt{generalLambert(f(x),a,d)} returns
++ \texttt{\{f(x**a) + f(x**(a + d)) + f(x**(a + 2 d)) + \ldots\}}.
++ \texttt{f(x)} should have zero constant
++ coefficient and \texttt{\{a\}} and \texttt{\{d\}} should be positive.

\texttt{multisect} : \texttt{(I,I,ST A) \rightarrow ST A}
++ \texttt{multisect(a,b,st)}
++ selects the coefficients of \texttt{\{x**((a+b)*n+a)\}},
++ and changes them to \texttt{\{x**n\}}.

\texttt{invmultisect} : \texttt{(I,I,ST A) \rightarrow ST A}
++ \texttt{invmultisect(a,b,st)} substitutes \texttt{\{x**((a+b)*n)\}} for \texttt{\{x**n\}}
++ and multiplies by \texttt{\{x**b\}}.

if \texttt{A} has \texttt{Algebra RN} then
\texttt{integrate} : \texttt{(A,ST A) \rightarrow ST A}
++ \texttt{integrate(r,a)} returns the integral of the power series \texttt{\{a\}}
++ with respect to the power series variable\texttt{integration} where
++ \texttt{r} denotes the constant of integration. Thus
++ \texttt{\{a\}} = \texttt{\{a0,a1,a2,...\}}.

\texttt{nlde} : \texttt{ST ST A \rightarrow ST A}
++ \texttt{nlde(u)} solves a
++ first order non-linear differential equation described by \texttt{u} of the
++ form \texttt{\{\{b<0,b<1>,...,\},[b<1,0>,b<1,1>,...]\}}.
++ the differential equation has the form
++ \texttt{\{y'=sum(i=0 to infinity,j=0 to infinity,b<i,j>* (x**i)*(y**j))\}}.

if \texttt{A} has \texttt{Field} then
\texttt{mapdiv} : \texttt{(ST A,ST A) \rightarrow ST A}
++ \texttt{mapdiv(r,f)} raises power series \texttt{f} to the power \texttt{r}.
mapdiv([a0,a1,...],[b0,b1,...]) returns \mapdiv{[\frac{a0}{b0},\frac{a1}{b1},...]}.

lazyGintegrate : (I -> A,A,() -> ST A) -> ST A
++ lazyGintegrate(f,r,g) is used for fixed point computations.
power : (A,ST A) -> ST A
++ power(a,f) returns the power series f raised to the power \pad{a}.

Implementation ==> add

--% definitions

zro: () -> ST A
-- returns a zero power series
zro() == empty()$ST(A)

--% arithmetic

x + y == delay
empty? y => x
empty? x => y
eq?(x,rst x) => map(z +-> frst x+z, y)
ep?(y,rst y) => map(z +-> frst y+z, x)
concat(frst x + frst y,rst x + rst y)

x - y == delay
empty? y => x
empty? x => -y
eq?(x,rst x) => map(z +-> frst x-z , y)
eq?(y,rst y) => map(z +-> z-frst y , x)
concat(frst x - frst y,rst x - rst y)

-y == map(z +-> -z, y)

(x:ST A) * (y:ST A) == delay
empty? y => zro()
empty? x => zro()
concat(frst x * frst y,frst x * rst y + rst x * y)

(s:A) * (x:ST A) ==
zero? s => zro()
map(z +-> s*z, x)

(x:ST A) * (s:A) ==
zero? s => zro()
map(z +-> z*s, x)
iDiv: (ST A,ST A,A) -> ST A
iDiv(x,y,ry0) == delay
empty? x => empty()
c0 := frst x * ry0
concat(c0, iDiv(rst x - c0 * rst y, y, ry0))

x exquo y ==
  for n in 1.. repeat
    n > 1000 => return "failed"
    empty? y => return "failed"
    empty? x => return empty()
    frst y = 0 =>
      frst x = 0 => (x := rst x; y := rst y)
      return "failed"
    leave "first entry in y is non-zero"
    (ry0 := recip frst y) case "failed" => "failed"
    empty? rst y => map(z +-> z*ry0 :: A), x)
  iDiv(x, y, ry0 :: A)

(x:ST A) / (y:ST A) == delay
  empty? y => error "/: division by zero"
  empty? x => empty()
  (ry0 := recip frst y) case "failed" =>
    error "/: second argument is not invertible"
  empty? rst y => map(z +-> z*(ry0::A), x)
  iDiv(x, y, ry0 :: A)

recip x ==
  empty? x => "failed"
  rh1 := recip frst x
  rh1 case "failed" => "failed"
  rh := rh1 :: A
  delay
  concat(rh, iDiv(- rh * rst x, x, rh))

--% coefficients

rp: (I, A) -> L A
-- rp(z, s) is a list of length z each of whose entries is s.
rp(z, s) ==
  z <= 0 => empty()
  concat(s, rp(z-1, s))

rpSt: (I, A) -> ST A
-- rpSt(z, s) is a stream of length z each of whose entries is s.
rpSt(z, s) == delay
  z <= 0 => empty()
  concat(s, rpSt(z-1, s))

monom(s, z) ==
  z < 0 => error "monom: cannot create monomial of negative degree"
  concat(rpSt(z, 0), concat(s, zro()))

--% some streams of integers
nnintegers: NNI -> ST NNI
nnintegers zz == generate(y +-> y+1, zz)
integers z     == generate(y +-> y+1, z)
oddintegers z == generate(y +-> y+2, z)
int s          == generate(y +-> y+1, s)

--% derivatives
mapmult(x,y) == delay
   empty? y => zro()
   empty? x => zro()
   concat(frst x * frst y,mapmult(rst x,rst y))
deriv x ==
   empty? x => zro()
   mapmult(int 1,rest x)
gderiv(f,x) ==
   empty? x => zro()
   mapmult(map(f,integers 0)$SP2(I,A),x)

--% coercions
coerce(s:A) ==
   zero? s => zro()
   concat(s,zro())

--% evaluations and compositions
eval(x,at) ==
   scan(0,(y,z) +-> y+z,mapmult(x,generate(y +-> at*y,1)))$SP2(A,A)
compose(x,y) == delay
   empty? y => concat(frst x,zro())
   not zero? frst y =>
      error "compose: 2nd argument should have 0 constant coefficient"
   empty? x => zro()
   concat(frst x,compose(rst x,y) * rst(y))

--% reversion
lagrangere:(ST A,ST A) -> ST A
lagrangere(x,c) == delay(concat(0,compose(x,c)))
lagrange x == YS(y +-> lagrangere(x,y))
revert x ==
   empty? x => error "revert should start 0,1,..."
   zero? frst x =>
      empty? rst x => error "revert: should start 0,1,..."
      one? frst rst x => lagrange(recip(rst x) :: (ST A))
\[(\text{frst rst } x) = 1 \Rightarrow \text{lagrange}\left(\text{recip}\left(\text{rst } x\right)\right) :: (\text{ST } A)\]
error "revert:should start 0,1,..."

\% Lambert functions

\text{addiag}(\text{ststa}:\text{ST } \text{ST } A) = \text{delay}
\quad \text{empty? } \text{ststa} \Rightarrow \text{zro()}
\quad \text{empty? } \text{frst } \text{ststa} \Rightarrow \text{concat}(0, \text{addiag } \text{rst } \text{ststa})
\quad \text{concat}(\text{frst}(\text{frst } \text{ststa}), \text{rst}(\text{frst } \text{ststa}) + \text{addiag}(\text{rst } \text{ststa}))

\% Lambert operates on a series \(+[a[i]x**i for i in 1..]\), and produces
\% the series \(+[a[i](x**i/(1-x**i)) for i in 1..]\) i.e. forms the
\% coefficients A[n] which is the sum of a[i] for all divisors i of n
\% (including 1 and n)

\text{rptg1}(I, A) -> ST A
\quad \text{---------}
\quad \text{return the repeating stream } [s,0,...,0]; \text{ (there are } z \text{ zeroes)}
\text{rptg1}(z, s) = \text{repeating } \text{concat}(s, \text{rp}(z, 0))

\text{rptg2}(I, A) -> ST A
\quad \text{---------}
\quad \text{return the repeating stream } [0,...,0,s,0,...,0]
\quad \text{there are } z \text{ leading zeroes and } z-1 \text{ in the period}
\text{rptg2}(z, s) = \text{repeating } \text{concat}(\text{rp}(z, 0), \text{concat}(s, \text{rp}(z-1, 0)))

\text{rptg3}(I, I, I, A) -> ST A
\text{rptg3}(a, d, n, s) =
\quad \text{concat}(\text{rpSt}(n*(a-1), 0), \text{repeating}(\text{concat}(s, \text{rp}(d*n-1, 0))))

\text{lambert } x = \text{delay}
\quad \text{empty? } x \Rightarrow \text{zro()}
\quad \text{zero? } \text{frst } x \Rightarrow
\quad \text{concat}(0, \text{addiag}(\text{map}(\text{rptg1}, \text{integers } 0, \text{rst } x)\text{SP3(I,A,ST A)}))
\quad \text{error } "\text{lambert:constant coefficient should be zero}"

\text{oddlambert } x = \text{delay}
\quad \text{empty? } x \Rightarrow \text{zro()}
\quad \text{zero? } \text{frst } x \Rightarrow
\quad \text{concat}(0, \text{addiag}(\text{map}(\text{rptg1}, \text{oddintegers } 1, \text{rst } x)\text{SP3(I,A,ST A)}))
\quad \text{error } "\text{oddlambert:constant coefficient should be zero}"

\text{evenlambert } x = \text{delay}
\quad \text{empty? } x \Rightarrow \text{zro()}
\quad \text{zero? } \text{frst } x \Rightarrow
\quad \text{concat}(0, \text{addiag}(\text{map}(\text{rptg2}, \text{integers } 1, \text{rst } x)\text{SP3(I,A,ST A)}))
\quad \text{error } "\text{evenlambert:constant coefficient should be zero}"

\text{generalLambert}(\text{st}, a, d) = \text{delay}
\quad a < 1 \text{ or } d < 1 \Rightarrow
error "generalLambert: both integer arguments must be positive"
empty? st => zro()
zero? first st =>
concat(0,adddiag(map((x,y) +-> rptg3(a,d,x,y),
        integers 1,rst st)$SP3(I,A,ST A)))
error "generalLambert: constant coefficient should be zero"

-- misc. functions

ms: (I,I,ST A) -> ST A
ms(m,n,s) == delay
empty? s => zro()
zero? n => concat(first s,ms(m,m-1,rst s))
ms(m,n-1,rst s)

multisect(b,a,x) == ms(a+b,0,rest(x,a :: NNI))

altn: (ST A,ST A) -> ST A
altn(zs,s) == delay
empty? s => zro()
concat(first s,concat(zs,altn(zs,rst s)))
invmultisect(a,b,x) ==
concat(rpSt(b,0),altn(rpSt(a + b - 1,0),x))

-- comps(ststa,y) forms the composition of +/b[i,j]*y**i*x**j
-- where y is a power series in y.

cssa ==> concat$(ST ST A)
mapsa ==> map$SP2(ST A,ST A)
comps: (ST ST A,ST A) -> ST ST A
comps(ststa,x) == delay$(ST ST A)
empty? ststa => empty()$(ST ST A)
empty? x => cssa(first ststa,empty()$(ST ST A))
cssa(first ststa,mapsa(y +-> (rst x)*y,comps(rst ststa,x)))

if A has Algebra RN then
integre: (ST A,1) -> ST A
integre(x,n) == delay
empty? x => zro()
concat((1$I/n) * first(x),integre(rst x,n + 1))

integ: ST A -> ST A
integ x == integre(x,1)

integrate(a,x) == concat(a,integ x)
lazyIntegrate(s,xf) == concat(s,integ(delay xf))

nldere:(ST ST A,ST A) -> ST A
nldere(lslsa,c) == lazyIntegrate(0,adddiag(comps(lslsa,c)))
nlde lslsa == YS(y +-> nldere(lslsa,y))

RATPOWERS : Boolean := A has "**": (A,RN) -> A

smult: (RN,ST A) -> ST A
smult(rn,x) == map(y +-> rn*y, x)

powerrn:(RN,ST A,ST A) -> ST A
powerrn(rn,x,c) == delay
    concat(1,integ(smult(rn + 1,c * deriv x)) - rst x * c)
powern(rn,x) ==
    order : I := 0
    for n in 0.. repeat
        empty? x => return zro()
        not zero? frst x => (order := n; leave x)
        x := rst x
        n = 1000 =>
            error "**: series with many leading zero coefficients"
        (ord := (order exquo denom(rn))) case "failed" =>
            error "**: rational power does not exist"
        co := frst x
        (invCo := recip co) case "failed" =>
            error "** rational power of coefficient undefined"
        order := order + ord
        x := powern(rn,x,c)
    end
    co**rn * YS(y +-> powerrn(rn,(invCo :: A)*x, y))

RATPOWERS => co**rn * YS(y +-> powerrn(rn,(invCo :: A)*x, y))
error "** rational power of coefficient undefined"

if A has Field then
    mapdiv(x,y) == delay
    empty? y => error "stream division by zero"
    empty? x => zro()
    concat(frst x/frst y,mapdiv(rst x,rst y))

    ginteg: (I -> A,ST A) -> ST A
ginteg(f,x) == mapdiv(x,map(f,integers 1)$SP2(I,A))

    lazyGintegrate(fntoa,s,xf) == concat(s,ginteg(fntoa,delay xf))

    finteg: ST A -> ST A
    finteg x == mapdiv(x,int 1)
    powerre: (A,ST A,ST A) -> ST A
    powerre(s,x,c) == delay
empty? x => zro()
frst x'=1 => error "** constant coefficient should be 1"
concat(frst x,finteg((s+1)*(c*deriv x))-rst x * c)
power(s,x) == YS(y += powerre(s,x,y))

package STNSR StreamTensor

)set break resume
)sys rm -f StreamTensor.output
)spool StreamTensor.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show StreamTensor
--R
--R StreamTensor(R: Type) is a package constructor
--R Abbreviation for StreamTensor is STNSR
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STNSR
--R
--R-------------------------------- Operations --------------------------------
--R tensorMap : (Stream(R),(R -> List(R))) -> Stream(R)
--R
--E 1
)

spool
)lisp (bye)
--- StreamTensor.help ---

=====================================================================
StreamTensor examples
=====================================================================

This package has no description

See Also:
o )show StreamTensor

---

StreamTensor (STNSR)

Exports:
tensorMap

--- package STNSR StreamTensor ---

)abbrev package STNSR StreamTensor
++ Description:
++ This package has no description
StreamTensor(R: Type): with

    tensorMap: (Stream R, R -> List R) -> Stream R
    ++ tensorMap([s1, s2, ...], f) returns the stream consisting of all
    ++ elements of f(s1) followed by all elements of f(s2) and so on.

    == add

    tensorMap(s, f) ==
empty? s => empty()
concat([f first s], delay tensorMap(rest s, f))

---

STNSR.dotabb

"STNSR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=STNSR"]
"TYPE" [color=lightblue,href="bookvol10.2.pdf#nameddest=TYPE"];
"STNSR" -> "TYPE"

---

package STTF StreamTranscendentalFunctions

---

StreamTranscendentalFunctions.input

)set break resume
)sys rm -f StreamTranscendentalFunctions.output
)spool StreamTranscendentalFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show StreamTranscendentalFunctions
--R
--R StreamTranscendentalFunctions(Coef: Algebra(Fraction(Integer))) is a package constructor
--R Abbreviation for StreamTranscendentalFunctions is STTF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for STTF
--R
--R------------------------------- Operations --------------------------------
--R acos : Stream(Coef) -> Stream(Coef) acosh : Stream(Coef) -> Stream(Coef)
--R acot : Stream(Coef) -> Stream(Coef) acoth : Stream(Coef) -> Stream(Coef)
--R acsc : Stream(Coef) -> Stream(Coef) acsch : Stream(Coef) -> Stream(Coef)
--R asec : Stream(Coef) -> Stream(Coef) asech : Stream(Coef) -> Stream(Coef)
--R asin : Stream(Coef) -> Stream(Coef) asinh : Stream(Coef) -> Stream(Coef)
--R atan : Stream(Coef) -> Stream(Coef) atanh : Stream(Coef) -> Stream(Coef)
--R cos : Stream(Coef) -> Stream(Coef) cosh : Stream(Coef) -> Stream(Coef)
--R cot : Stream(Coef) -> Stream(Coef) coth : Stream(Coef) -> Stream(Coef)
--R csc : Stream(Coef) -> Stream(Coef) csch : Stream(Coef) -> Stream(Coef)
--R exp : Stream(Coef) -> Stream(Coef) log : Stream(Coef) -> Stream(Coef)
--R sec : Stream(Coef) -> Stream(Coef) sech : Stream(Coef) -> Stream(Coef)
StreamTranscendentalFunctions (STTF)

STTF

STRING

---

StreamTranscendentalFunctions examples

StreamTranscendentalFunctions implements transcendental functions on Taylor series, where a Taylor series is represented by a stream of its coefficients.

See Also:
- )show StreamTranscendentalFunctions

---
Exports:
acos acot acsc asec asin atan cos cot csc exp sec sin tan sincos sinhcosh

-- package STTF StreamTranscendentalFunctions --

)abbrev package STTF StreamTranscendentalFunctions
++ Author: William Burge, Clifton J. Williamson
++ Date Created: 1986
++ Date Last Updated: 6 March 1995
++ Description:
++ StreamTranscendentalFunctions implements transcendental functions on
++ Taylor series, where a Taylor series is represented by a stream of
++ its coefficients.

StreamTranscendentalFunctions(Coef): Exports == Implementation where
  Coef : Algebra Fraction Integer
  L ==> List
  I ==> Integer
  RN ==> Fraction Integer
  SG ==> String
  ST ==> Stream Coef
  STT ==> StreamTaylorSeriesOperations Coef
  YS ==> Y$ParadoxicalCombinatorsForStreams(Coef)

Exports ==> with
--% Exponentials and Logarithms
  exp : ST -> ST  
    ++ exp(st) computes the exponential of a power series st.
  log : ST -> ST  
    ++ log(st) computes the log of a power series.
  "**" : (ST,ST) -> ST
    ++ st1 ** st2 computes the power of a power series st1 by another
    ++ power series st2.

--% TrigonometricFunctionCategory
  sincos : ST -> Record(sin:ST, cos:ST)  
    ++ sincos(st) returns a record containing the sine and cosine
    ++ of a power series st.
  sin : ST -> ST
    ++ sin(st) computes sine of a power series st.
  cos : ST -> ST
    ++ cos(st) computes cosine of a power series st.
  tan : ST -> ST
    ++ tan(st) computes tangent of a power series st.
  cot : ST -> ST
    ++ cot(st) computes cotangent of a power series st.
  sec : ST -> ST
++ sec(st) computes secant of a power series st.
csc : ST -> ST
  ++ csc(st) computes cosecant of a power series st.
asin : ST -> ST
  ++ asin(st) computes arcsine of a power series st.
acos : ST -> ST
  ++ acos(st) computes arccosine of a power series st.
atan : ST -> ST
  ++ atan(st) computes arctangent of a power series st.
acot : ST -> ST
  ++ acot(st) computes arccotangent of a power series st.
asec : ST -> ST
  ++ asec(st) computes arcsecant of a power series st.
acsc : ST -> ST
  ++ acsc(st) computes arccosecant of a power series st.

--% HyperbolicTrigonometricFunctionCategory

sinhcosh: ST -> Record(sinh:ST, cosh:ST)
  ++ sinh(sinh(st) returns a record containing
  ++ the hyperbolic sine and cosine
  ++ of a power series st.
sinh : ST -> ST
  ++ sinh(st) computes the hyperbolic sine of a power series st.
cosh : ST -> ST
  ++ cosh(st) computes the hyperbolic cosine of a power series st.
tanh : ST -> ST
  ++ tanh(st) computes the hyperbolic tangent of a power series st.
coth : ST -> ST
  ++ coth(st) computes the hyperbolic cotangent of a power series st.
sech : ST -> ST
  ++ sech(st) computes the hyperbolic secant of a power series st.
csch : ST -> ST
  ++ csch(st) computes the hyperbolic cosecant of a power series st.
asinh : ST -> ST
  ++ asinh(st) computes the inverse hyperbolic sine of a power series st.
acosh : ST -> ST
  ++ acosh(st) computes the inverse hyperbolic cosine
  ++ of a power series st.
atanh : ST -> ST
  ++ atanh(st) computes the inverse hyperbolic tangent
  ++ of a power series st.
acoth : ST -> ST
  ++ acoth(st) computes the inverse hyperbolic cotangent of a power series st.
asech : ST -> ST
  ++ asech(st) computes the inverse hyperbolic secant of a power series st.
acsch : ST -> ST
  ++ acsch(st) computes the inverse hyperbolic cosecant of a power series st.
Implementation ==> add
import StreamTaylorSeriesOperations Coef

TRANSFCN : Boolean := Coef has TranscendentalFunctionCategory

--% Error Reporting
TRCONST : SG := "series expansion involves transcendental constants"
NPOWERS : SG := "series expansion has terms of negative degree"
FPowers : SG := "series expansion has terms of fractional degree"
MAYFPOW : SG := "series expansion may have terms of fractional degree"
LOGS : SG := "series expansion has logarithmic term"
NPOWLOG : SG :=
    "series expansion has terms of negative degree or logarithmic term"
FPOLWLOG : SG :=
    "series expansion has terms of fractional degree or logarithmic term"
NOTINV : SG := "leading coefficient not invertible"

--% Exponentials and Logarithms
expre:(Coef,ST,ST) -> ST
expre(r,e,dx) == lazyIntegrate(r,e*dx)
expre z ==
empty? z => 1 :: ST
(coef := frst z) = 0 => YS(y +-> expre(1,y,deriv z))
TRANSFCN => YS(y +-> expre(exp coef,y,deriv z))
error concat("exp: ",TRCONST)

log z ==
empty? z => error "log: constant coefficient should not be 0"
(coef := frst z) = 0 => error "log: constant coefficient should not be 0"
coef = 1 => lazyIntegrate(0,deriv z/z)
TRANSFCN => lazyIntegrate(log coef,deriv z/z)
error concat("log: ",TRCONST)

z1:ST ** z2:ST == exp(z2 * log z1)

--% Trigonometric Functions
sincosre:(Coef,Coef,L ST,ST,Coef) -> L ST
sincosre(rs,rc,sc,dx,sign) ==
[lazyIntegrate(rs,(second sc)*dx),lazyIntegrate(rc,sign*(first sc)*dx)]

-- When the compiler had difficulties with the above definition,
-- I did the following to help it:
-- sincosre:(Coef,Coef,L ST,ST,Coef) -> L ST
-- sincosre(rs,rc,sc,dx,sign) ==
sincos z ==
empty? z => [0 :: ST,1 :: ST]
l :=
(coef := frst z) = 0 => YS(y +-> sincosre(0,1,y,deriv z,-1),2)
TRANSFCN => YS(y +-> sincosre(sin coef,cos coef,y,deriv z,-1),2)
error concat("sincos: ",TRCONST)[first l,second l]

sin z == sincos(z).sin
cos z == sincos(z).cos

tanre:(Coef,ST,ST,Coef) -> ST
tanre(r,t,dx,sign) == lazyIntegrate(r,((1 :: ST) + sign*t*t)*dx)

-- When the compiler had difficulties with the above definition,
-- I did the following to help it:

-- tanre:(Coef,ST,ST,Coef) -> ST
-- tanre(r,t,dx,sign) ==
-- st1 : ST := t * t
-- st1 := sign * st1
-- st2 : ST := 1 :: ST
-- st1 := st2 + st1
-- st1 := st1 * dx
-- lazyIntegrate(r,st1)

tan z ==
empty? z => 0 :: ST
(coef := frst z) = 0 => YS(y +-> tanre(0,y,deriv z,1))
TRANSFCN => YS(y +-> tanre(tan coef,y,deriv z,1))
error concat("tan: ",TRCONST)

cotre:(Coef,ST,ST) -> ST
cotre(r,t,dx) == lazyIntegrate(r,-((1 :: ST) + t*t)*dx)

-- When the compiler had difficulties with the above definition,
-- I did the following to help it:

-- cotre:(Coef,ST,ST) -> ST
cotre(r,t,dx) ==
  -- st1 : ST := t * t
  -- st2 : ST := 1 :: ST
  -- st1 := st2 + st1
  -- st1 := st1 * dx
  -- st1 := -st1
-- lazyIntegrate(r,st1)

cot z ==
  empty? z => error "cot: cot(0) is undefined"
  (coef := frst z) = 0 => error concat("cot: ",NPOWERS)
  TRANSFCN => YS(y ++> cotre(cot coef,y,deriv z))
  error concat("cot: ",TRCONST)

sec z ==
  empty? z => 1 :: ST
  frst z = 0 => recip(cos z) :: ST
  TRANSFCN =>
    cosz := cos z
    first cosz = 0 => error concat("sec: ",NPOWERS)
    recip(cosz) :: ST
    error concat("sec: ",TRCONST)

csc z ==
  empty? z => error "csc: csc(0) is undefined"
  TRANSFCN =>
    sinz := sin z
    first sinz = 0 => error concat("csc: ",NPOWERS)
    recip(sinz) :: ST
    error concat("csc: ",TRCONST)

orderOrFailed : ST -> Union(I,"failed")
orderOrFailed x ==
  -- returns the order of x or "failed"
  -- if -1 is returned, the series is identically zero
  for n in 0..1000 repeat
    empty? x => return -1
    not zero? frst x => return n :: I
    x := rst x
  "failed"

asin z ==
  empty? z => 0 :: ST
  (coef := frst z) = 0 =>
    integrate(0,powern(-1/2,(1 :: ST) - z*z) * (deriv z))
  TRANSFCN =>
    coef = 1 or coef = -1 =>
    x := (1 :: ST) - z*z
    -- compute order of 'x'
    (ord := orderOrFailed x) case "failed" =>
      error concat("asin: ",MAYFPow)
    (order := ord :: I) = -1 => return asin(coef) :: ST
    odd? order => error concat("asin: ",FPowers)
    squirt := powern(1/2,x)
    (quot := (deriv z) exquo squirt) case "failed" =>
      error concat("asin: ",NOTinv)
integrate(asin coef,quot :: ST)
integrate(asin coef,powern(-1/2,(1 :: ST) - z*z) * (deriv z))
error concat("asin: ",TRCONST)

acos z ==
empty? z =>
TRANSFCN => acos(0)$Coef :: ST
error concat("acos: ",TRCONST)
TRANSFCN =>
coef := frst z
coef = 1 or coef = -1 =>
x := (1 :: ST) - z*z
-- compute order of 'x'
(order := orderOrFailed x) case "failed" =>
error concat("acos: ",MAYFPW)
(order := ord :: I) = -1 => return acos(coef) :: ST
odd? order => error concat("acos: ",FPOWERS)
squirt := powern(1/2,x)
(quot := (-deriv z) exquo squirt) case "failed" =>
error concat("acos: ",NOTINV)
integrate(acos coef,quot :: ST)
integrate(acos coef,-powern(-1/2,(1 :: ST) - z*z) * (deriv z))
error concat("acos: ",TRCONST)

atan z ==
empty? z => 0 :: ST
(coef := frst z) = 0 =>
integrate(0,(recip((1 :: ST) + z*z) :: ST) * (deriv z))
TRANSFCN =>
(y := recip((1 :: ST) + z*z)) case "failed" =>
error concat("atan: ",LOGS)
integrate(atan coef,(y :: ST) * (deriv z))
error concat("atan: ",TRCONST)

acot z ==
empty? z =>
TRANSFCN => acot(0)$Coef :: ST
error concat("acot: ",TRCONST)
TRANSFCN =>
(y := recip((1 :: ST) + z*z)) case "failed" =>
error concat("acot: ",LOGS)
integrate(acot frst z,-(y :: ST) * (deriv z))
error concat("acot: ",TRCONST)

asec z ==
empty? z => error "asec: constant coefficient should not be 0"
TRANSFCN =>
(coef := frst z) = 0 =>
error "asec: constant coefficient should not be 0"
coef = 1 or coef = -1 =>
x := z*z - (1 :: ST)
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("asec: ",MAYFPow)
(order := ord :: I) = -1 => return asec(coef) :: ST
odd? order => error concat("asec: ",FPowers)
squirt := powern(1/2,x)
(quot := (deriv z) exquo squirt) case "failed" =>
  error concat("asec: ",NOTINV)
(quot2 := (quot :: ST) exquo z) case "failed" =>
  error concat("asec: ",NOTINV)
integrate(asec coef,quot2 :: ST)
integrate(asec coef,(powern(-1/2,z*z-(1::ST))*(deriv z)) / z)
error concat("asec: ",TRCONST)

acsc z ==
empty? z => error "acsc: constant coefficient should not be zero"
TRANSFCN =>
(coef := frst z) = 0 =>
  error "acsc: constant coefficient should not be zero"
coef = 1 or coef = -1 =>
x := z*z - (1 :: ST)
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("acsc: ",MAYFPow)
(order := ord :: I) = -1 => return acsc(coef) :: ST
odd? order => error concat("acsc: ",FPowers)
squirt := powern(1/2,x)
(quot := (-deriv z) exquo squirt) case "failed" =>
  error concat("acsc: ",NOTINV)
(quot2 := (quot :: ST) exquo z) case "failed" =>
  error concat("acsc: ",NOTINV)
integrate(acsc coef,quot2 :: ST)
integrate(acsc coef,-(powern(-1/2,z*z-(1::ST))*(deriv z)) / z)
error concat("acsc: ",TRCONST)

--% Hyperbolic Trigonometric Functions

sinhcosh z ==
empty? z => [0 :: ST,1 :: ST]
l :=
(coef := frst z) = 0 => YS(y +-> sincosre(0,1,y,deriv z,1),2)
TRANSFCN => YS(y +-> sincosre(sinh coef,cosh coef,y,deriv z,1),2)
error concat("sinhcosh: ",TRCONST)
[first l,second l]

sinh z == sinhcos(z).sinh
cosh z == sinhcos(z).cosh
tanh z ==
empty? z => 0 :: ST
(coef := frst z) = 0 => YS(y +-> tanre(0,y,deriv z,-1))
TRANSFCN => YS(y +-> tanre(tanh coef,y,deriv z,-1))
error concat("tanh: ",TRCONST)

coth z ==
tanhz := tanh z
empty? tanhz => error "coth: coth(0) is undefined"
(frst tanhz) = 0 => error concat("coth: ",NPOWERS)
recip(tanhz) :: ST

sech z ==
coshz := cosh z
(empty? coshz) or (frst coshz = 0) => error concat("sech: ",NPOWERS)
recip(coshz) :: ST

csch z ==
sinhz := sinh z
(empty? sinhz) or (frst sinhz = 0) => error concat("csch: ",NPOWERS)
recip(sinhz) :: ST

asinh z ==
empty? z => 0 :: ST
(coef := frst z) = 0 => log(z + powern(1/2,(1 :: ST) + z*z))
TRANSFCN =>
x := (1 :: ST) + z*z
-- compute order of 'x', in case coefficient(z,0) = +- %i
(ord := orderOrFailed x) case "failed" =>
  error concat("asinh: ",MAYFPow)
(order := ord :: I) = -1 => return asinh(coef) :: ST
odd? order => error concat("asinh: ",FPOWERS)
-- the argument to 'log' must have a non-zero constant term
log(z + powern(1/2,x))
error concat("asinh: ",TRCONST)

acosh z ==
empty? z =>
TRANSFCN => acosh(0)$Coef :: ST
error concat("acosh: ",TRCONST)
TRANSFCN =>
coef := frst z
coef = 1 or coef = -1 =>
x := z*z - (1 :: ST)
-- compute order of 'x'
(ord := orderOrFailed x) case "failed" =>
  error concat("acosh: ",MAYFPow)
(order := ord :: I) = -1 => return acosh(coef) :: ST
odd? order => error concat("acosh: ",FPOWERS)
-- the argument to 'log' must have a non-zero constant term
log(z + powern(1/2,x))
\[
\log(z + \text{powern}(1/2, z^2 - (1 :: ST)))
\]
error concat("acosh: ", TRCONST)

**atanh z**

empty? z \Rightarrow 0 :: ST
(coef := frst z) = 0 \Rightarrow
  (inv(2 :: RN) :: Coef) \ast \log(((1 :: ST) + z) / ((1 :: ST) - z))
TRANSFCN \Rightarrow
c coef = 1 or coef = -1 \Rightarrow error concat("atanh: ", LOGS)
  (inv(2 :: RN) :: Coef) \ast \log(((1 :: ST) + z) / ((1 :: ST) - z))
error concat("atanh: ", TRCONST)

**acoth z**

empty? z \Rightarrow
TRANSFCN \Rightarrow acoth(0) :: ST
error concat("acoth: ", TRCONST)
TRANSFCN \Rightarrow
  frst z = 1 or frst z = -1 \Rightarrow error concat("acoth: ", LOGS)
  (inv(2 :: RN) :: Coef) \ast \log((z + (1 :: ST)) / (z - (1 :: ST)))
error concat("acoth: ", TRCONST)

**asech z**

empty? z \Rightarrow error "asech: asech(0) is undefined"
TRANSFCN \Rightarrow
  (coef := frst z) = 0 \Rightarrow error concat("asech: ", NPOWLOG)
  coef = 1 or coef = -1 \Rightarrow
    x := (1 :: ST) - z^2
    -- compute order of 'x'
    (ord := orderOrFailed x) case "failed" =>
      error concat("asech: ", MAYFPOW)
    (order := ord :: I) = -1 \Rightarrow return asech(coef) :: ST
    odd? order \Rightarrow error concat("asech: ", FPOWERS)
      log(((1 :: ST) + powern(1/2, x)) / z)
    log(((1 :: ST) + powern(1/2, (1 :: ST) - z^2)) / z)
error concat("asech: ", TRCONST)

**acsch z**

empty? z \Rightarrow error "acsch: acsch(0) is undefined"
TRANSFCN \Rightarrow
  frst z = 0 \Rightarrow error concat("acsch: ", NPOWLOG)
  x := z^2 + (1 :: ST)
  -- compute order of 'x'
  (ord := orderOrFailed x) case "failed" =>
    error concat("acsch: ", MAYFPOW)
  (order := ord :: I) = -1 \Rightarrow return acsch(frst z) :: ST
  odd? order \Rightarrow error concat("acsch: ", FPOWERS)
    log(((1 :: ST) + powern(1/2, x)) / z)
  log(((1 :: ST) + powern(1/2, (1 :: ST) - z^2)) / z)
error concat("acsch: ", TRCONST)
package STTFNC StreamTranscendentalFunctionsNonCommutative

Operations

-R acos : Stream(Coef) -> Stream(Coef)  acosh : Stream(Coef) -> Stream(Coef)
-R acot : Stream(Coef) -> Stream(Coef)  acoth : Stream(Coef) -> Stream(Coef)
-R acsc : Stream(Coef) -> Stream(Coef)  acsch : Stream(Coef) -> Stream(Coef)
-R asec : Stream(Coef) -> Stream(Coef)  asech : Stream(Coef) -> Stream(Coef)
-R asin : Stream(Coef) -> Stream(Coef)  asinh : Stream(Coef) -> Stream(Coef)
-R atan : Stream(Coef) -> Stream(Coef)  atanh : Stream(Coef) -> Stream(Coef)
-R cos : Stream(Coef) -> Stream(Coef)  cosh : Stream(Coef) -> Stream(Coef)
-R cot : Stream(Coef) -> Stream(Coef)  coth : Stream(Coef) -> Stream(Coef)
-R csc : Stream(Coef) -> Stream(Coef)  csch : Stream(Coef) -> Stream(Coef)
-R exp : Stream(Coef) -> Stream(Coef)  log : Stream(Coef) -> Stream(Coef)
-R sec : Stream(Coef) -> Stream(Coef)  sech : Stream(Coef) -> Stream(Coef)
-R sin : Stream(Coef) -> Stream(Coef)  sinh : Stream(Coef) -> Stream(Coef)
-R tan : Stream(Coef) -> Stream(Coef)  tanh : Stream(Coef) -> Stream(Coef)
-R ** : (Stream(Coef),Stream(Coef)) -> Stream(Coef)
StreamTranscendentalFunctionsNonCommutative (STTFNC)

Exports:
acos acot acsc asec asin
atan cos cot csc exp
sec sin tan ???

— package STTFNC StreamTranscendentalFunctionsNonCommutative —
(abbrev package STTFNC StreamTranscendentalFunctionsNonCommutative
   ++ Author: Clifton J. Williamson
   ++ Date Created: 26 May 1994
   ++ Date Last Updated: 26 May 1994
   ++ Description: StreamTranscendentalFunctionsNonCommutative implements transcendental functions on Taylor series over a non-commutative ring, where a Taylor series is represented by a stream of its coefficients.

StreamTranscendentalFunctionsNonCommutative(Coef): _
   Exports == Implementation where
     Coef : Algebra Fraction Integer
     I    ==> Integer
     SG   ==> String
     ST   ==> Stream Coef
     STTF ==> StreamTranscendentalFunctions Coef

   Exports ==> with
   --% Exponentials and Logarithms
     exp : ST -> ST
     log : ST -> ST
     "**" : (ST,ST) -> ST
     ++ exp(st) computes the exponential of a power series st.
     ++ log(st) computes the log of a power series.
     ++ st1 ** st2 computes the power of a power series st1 by another power series st2.

   --% TrigonometricFunctionCategory
     sin : ST -> ST
     cos : ST -> ST
     tan : ST -> ST
     cot : ST -> ST
     sec : ST -> ST
     csc : ST -> ST
     asin : ST -> ST
     acos : ST -> ST
     atan : ST -> ST
     acot : ST -> ST
     asec : ST -> ST
     ++ sin(st) computes sine of a power series st.
     ++ cos(st) computes cosine of a power series st.
     ++ tan(st) computes tangent of a power series st.
     ++ cot(st) computes cotangent of a power series st.
     ++ sec(st) computes secant of a power series st.
     ++ csc(st) computes cosecant of a power series st.
     ++ asin(st) computes arcsine of a power series st.
     ++ acos(st) computes arccosine of a power series st.
     ++ atan(st) computes arctangent of a power series st.
     ++ acot(st) computes arccotangent of a power series st.
     ++ asec(st) computes arccosecant of a power series st.

   Routines: 
     ++ exp(st) computes the exponential of a power series st.
     ++ log(st) computes the log of a power series.
     ++ st1 ** st2 computes the power of a power series st1 by another power series st2.
     ++ sin(st) computes sine of a power series st.
     ++ cos(st) computes cosine of a power series st.
     ++ tan(st) computes tangent of a power series st.
     ++ cot(st) computes cotangent of a power series st.
     ++ sec(st) computes secant of a power series st.
     ++ csc(st) computes cosecant of a power series st.
     ++ asin(st) computes arcsine of a power series st.
     ++ acos(st) computes arccosine of a power series st.
     ++ atan(st) computes arctangent of a power series st.
     ++ acot(st) computes arccotangent of a power series st.
     ++ asec(st) computes arccosecant of a power series st.)
acsc : ST -> ST
  ++ acsc(st) computes arccosecant of a power series st.

--% HyperbolicTrigonometricFunctionCategory
sinh : ST -> ST
  ++ sinh(st) computes the hyperbolic sine of a power series st.
cosh : ST -> ST
  ++ cosh(st) computes the hyperbolic cosine of a power series st.
tanh : ST -> ST
  ++ tanh(st) computes the hyperbolic tangent of a power series st.
coth : ST -> ST
  ++ coth(st) computes the hyperbolic cotangent of a power series st.
sech : ST -> ST
  ++ sech(st) computes the hyperbolic secant of a power series st.
csch : ST -> ST
  ++ csch(st) computes the hyperbolic cosecant of a power series st.
asinh : ST -> ST
  ++ asinh(st) computes the inverse hyperbolic sine of a power series st.
acosh : ST -> ST
  ++ acosh(st) computes the inverse hyperbolic cosine
  ++ of a power series st.
atanh : ST -> ST
  ++ atanh(st) computes the inverse hyperbolic tangent
  ++ of a power series st.
acoth : ST -> ST
  ++ acoth(st) computes the inverse hyperbolic
  ++ cotangent of a power series st.
asech : ST -> ST
  ++ asech(st) computes the inverse hyperbolic secant of a
  ++ power series st.
acsch : ST -> ST
  ++ acsch(st) computes the inverse hyperbolic cosecant of a
  ++ power series st.

Implementation ==> add
  import StreamTaylorSeriesOperations(Coef)

--% Error Reporting
ZERO : SG := "series must have constant coefficient zero"
ONE : SG := "series must have constant coefficient one"
NPOWERS : SG := "series expansion has terms of negative degree"

--% Exponentials and Logarithms
exp z ==
  empty? z => 1 :: ST
  (frst z) = 0 =>
    expx := exp(monom(1,1))$STTF
    compose(expx,z)
error concat("exp: ",ZERO)

log z ==
    empty? z => error concat("log: ",ONE)
    (frst z) = 1 =>
        log1PlusX := log(monom(1,0) + monom(1,1))$STTF
        compose(log1PlusX,z - monom(1,0))
        error concat("log: ",ONE)

    (z1:ST) ** (z2:ST) == exp(log(z1) * z2)

--% Trigonometric Functions

sin z ==
    empty? z => 0 :: ST
    (frst z) = 0 =>
        sinx := sin(monom(1,1))$STTF
        compose(sinx,z)
        error concat("sin: ",ZERO)

cos z ==
    empty? z => 1 :: ST
    (frst z) = 0 =>
        cosx := cos(monom(1,1))$STTF
        compose(cosx,z)
        error concat("cos: ",ZERO)

tan z ==
    empty? z => 0 :: ST
    (frst z) = 0 =>
        tanx := tan(monom(1,1))$STTF
        compose(tanx,z)
        error concat("tan: ",ZERO)

cot z ==
    empty? z => error "cot: cot(0) is undefined"
    (frst z) = 0 => error concat("cot: ",NPOWERS)
    error concat("cot: ",ZERO)

sec z ==
    empty? z => 1 :: ST
    (frst z) = 0 =>
        secx := sec(monom(1,1))$STTF
        compose(secx,z)
        error concat("sec: ",ZERO)

csc z ==
    empty? z => error "csc: csc(0) is undefined"
    (frst z) = 0 => error concat("csc: ",NPOWERS)
    error concat("csc: ",ZERO)
asinx := asin(monom(1,1))$STTF
compose(asinx,z)
error concat("asin: ",ZERO)

atan x := atan(monom(1,1))$STTF
compose(atanx,z)
error concat("atan: ",ZERO)

acos z == error "acos: acos undefined on this coefficient domain"
acot z == error "acot: acot undefined on this coefficient domain"
asec z == error "asec: asec undefined on this coefficient domain"
acsc z == error "acsc: acsc undefined on this coefficient domain"

--% Hyperbolic Trigonometric Functions

sinhx := sinh(monom(1,1))$STTF
compose(sinhx,z)
error concat("sinh: ",ZERO)

cosh x := cosh(monom(1,1))$STTF
compose(coshx,z)
error concat("cosh: ",ZERO)

tanh x := tanh(monom(1,1))$STTF
compose(tanhx,z)
error concat("tanh: ",ZERO)

coth x := error "coth: coth(0) is undefined"
(frst z) = 0 => error concat("coth: ",NPOWERS)
error concat("coth: ",ZERO)

sech z ==
empty? z => 1 :: ST
\begin{verbatim}
(frst z) = 0 =>
    sechx := sech(monom(1,1))$STTF
    compose(sechx,z)
    error concat("sech: ",ZERO)

csch z ==
    empty? z => error "csch: csch(0) is undefined"
    (frst z) = 0 => error concat("csch: ",NPOWERS)
    error concat("csch: ",ZERO)

asinh z ==
    empty? z => 0 :: ST
    (frst z) = 0 =>
        asinhx := asinh(monom(1,1))$STTF
        compose(asinhx,z)
        error concat("asinh: ",ZERO)

atanh z ==
    empty? z => 0 :: ST
    (frst z) = 0 =>
        atanhx := atanh(monom(1,1))$STTF
        compose(atanhx,z)
        error concat("atanh: ",ZERO)

acosh z == error "acosh: acosh undefined on this coefficient domain"
acoth z == error "acoth: acoth undefined on this coefficient domain"
asech z == error "asech: asech undefined on this coefficient domain"
acsch z == error "acsch: acsch undefined on this coefficient domain"
\end{verbatim}
(sys rm -f StructuralConstantsPackage.output
spool StructuralConstantsPackage.output
set message test on
set message auto off
clear all

--S 1 of 1
show StructuralConstantsPackage
--R
--R StructuralConstantsPackage(R: Field) is a package constructor
--R Abbreviation for StructuralConstantsPackage is SCPKG
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SCPKG
--R
--R----------------------------------- Operations -----------------------------------
--R    coordinates : (Matrix(R),List(Matrix(R))) -> Vector(R)
--R    structuralConstants : (List(Symbol),Matrix(Fraction(Polynomial(R)))) -> Vector(Matrix(Fraction(Polynomial(R))))
--R    structuralConstants : (List(Symbol),Matrix(Polynomial(R))) -> Vector(Matrix(Polynomial(R)))
--R    structuralConstants : List(Matrix(R)) -> Vector(Matrix(R))
--R
--E 1

)spool
)lisp (bye)

--- StructuralConstantsPackage.help ---

====================================================================
StructuralConstantsPackage examples
====================================================================

StructuralConstantsPackage provides functions creating structural constants from a multiplication tables or a basis of a matrix algebra and other useful functions in this context.

See Also:
  o )show StructuralConstantsPackage

---
StructuralConstantsPackage (SCPKG)

Exports:
coordinates structuralConstants

— package SCPKG StructuralConstantsPackage —

)abbrev package SCPKG StructuralConstantsPackage
++ Authors: J. Grabmeier
++ Date Created: 02 April 1992
++ Date Last Updated: 14 April 1992
++ Description:
++ StructuralConstantsPackage provides functions creating
++ structural constants from a multiplication tables or a basis
++ of a matrix algebra and other useful functions in this context.

StructuralConstantsPackage(R:Field): public == private where

L ==> List
S ==> Symbol
FRAC ==> Fraction
POLY ==> Polynomial
V ==> Vector
M ==> Matrix
REC ==> Record(particular: Union(V R,"failed"),basis: List V R)
LSMP ==> LinearSystemMatrixPackage(R,V R,V R, M R)

public ==> with
   -- what we really want to have here is a matrix over
   -- linear polynomials in the list of symbols, having arbitrary
   -- coefficients from a ring extension of R, e.g. FRAC POLY R.
   structuralConstants : (L S, M FRAC POLY R) -> V M FRAC POLY R
   ++ structuralConstants(ls,mt) determines the structural constants
   ++ of an algebra with generators ls and multiplication table mt, the
   ++ entries of which must be given as linear polynomials in the
   ++ indeterminates given by ls. The result is in particular useful
   ++ as fourth argument for \spadtype{AlgebraGivenByStructuralConstants}
++ and \spadtype{GenericNonAssociativeAlgebra}.
structuralConstants : (L S, M POLY R) -> V M POLY R
++ structuralConstants(ls,mt) determines the structural constants
++ of an algebra with generators ls and multiplication table mt, the
++ entries of which must be given as linear polynomials in the
++ indeterminates given by ls. The result is in particular useful
++ as fourth argument for \spadtype{AlgebraGivenByStructuralConstants}
++ and \spadtype{GenericNonAssociativeAlgebra}.
structuralConstants: L M R -> V M R
++ structuralConstants(basis) takes the basis of a matrix
++ algebra, e.g. the result of \spadfun{basisOfCentroid} and calculates
++ the structural constants.
++ Note, that the it is not checked, whether basis really is a
++ basis of a matrix algebra.
coordinates: (M R, L M R) -> V R
++ coordinates(a,[v1,...,vn]) returns the coordinates of \spad{a}
++ with respect to the \spad{R}-module basis \spad{v1},...,\spad{vn}.

private ==> add

matrix2Vector: M R -> V R
matrix2Vector m ==
lili : L L R := listOfLists m
li : L R := reduce(concat, lili)
construct(li)$(V R)

coordinates(x,b) ==
m : NonNegativeInteger := (maxIndex b) :: NonNegativeInteger
n : NonNegativeInteger := nrows(b.1) * ncols(b.1)
transitionMatrix : Matrix R := new(n,m,0$R)$Matrix(R)
for i in 1..m repeat
  setColumn_!(transitionMatrix,i,matrix2Vector(b.i))
res : REC := solve(transitionMatrix,matrix2Vector(x))$LSMP
if (not every?(zero?$R,first res.basis)) then
  error("coordinates: the second argument is linearly dependent")
(res.particular case "failed") =>$
  error("coordinates: first argument is not in linear span of second argument")
(res.particular) :: (Vector R)

structuralConstants b ==
-- be careful with the possibility that b is not a basis
-- m : NonNegativeInteger := (maxIndex b) :: NonNegativeInteger
m : NonNegativeInteger := rank()
sC : Vector Matrix R := [new(m,m,0$R) for k in 1..m]
for i in 1..m repeat
  for j in 1..m repeat
    covec : Vector R := coordinates(b.i * b.j, b)%%
    for k in 1..m repeat
setelt( sC.k, i, j, covec.k )

structuralConstants(ls:L S, mt: M POLY R) ==
nn := #(ls)
nrows(mt) ~= nn or ncols(mt) ~= nn =>
   error "structuralConstants: size of second argument does not agree with number of generators"
gamma : L M POLY R := []
lscopy : L S := copy ls
while not null lscopy repeat
   mat : M POLY R := new(nn,nn,0)
s : S := first lscopy
   for i in 1..nn repeat
      for j in 1..nn repeat
         p := qelt(mt,i,j)
         totalDegree(p,ls) > 1 =>
            error "structuralConstants: entries of second argument must be linear polynomials in the generators"
         if (c := coefficient(p, s, 1) ) ~= 0 then qsetelt_!(mat,i,j,c)
         gamma := cons(mat, gamma)
lscopy := rest lscopy
vector reverse gamma

structuralConstants(ls:L S, mt: M FRAC POLY R) ==
nn := #(ls)
nrows(mt) ~= nn or ncols(mt) ~= nn =>
   error "structuralConstants: size of second argument does not agree with number of generators"
gamma : L M FRAC(POLY R) := []
lscopy : L S := copy ls
while not null lscopy repeat
   mat : M FRAC(POLY R) := new(nn,nn,0)
s : S := first lscopy
   for i in 1..nn repeat
      for j in 1..nn repeat
         r := qelt(mt,i,j)
         q := denom(r)
         totalDegree(q,ls) ~= 0 =>
            error "structuralConstants: entries of second argument must be (linear) polynomials in the generators"
         p := numer(r)
         totalDegree(p,ls) > 1 =>
            error "structuralConstants: entries of second argument must be linear polynomials in the generators"
         if (c := coefficient(p, s, 1) ) ~= 0 then qsetelt_!(mat,i,j,c/q)
         gamma := cons(mat, gamma)
lscopy := rest lscopy
vector reverse gamma
package SHP SturmHabichtPackage

-- SturmHabichtPackage.input --

)set break resume
)sys rm -f SturmHabichtPackage.output
)spool SturmHabichtPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SturmHabichtPackage
--R
--R SturmHabichtPackage(R: OrderedIntegralDomain,x: Symbol) is a package constructor
--R Abbreviation for SturmHabichtPackage is SHP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SHP
--R
--R---------------------------------------- Operations ----------------------------------------
--R SturmHabicht : (UnivariatePolynomial(x,R),UnivariatePolynomial(x,R)) -> Integer
--R SturmHabichtCoefficients : (UnivariatePolynomial(x,R),UnivariatePolynomial(x,R)) -> List(R)
--R SturmHabichtMultiple : (UnivariatePolynomial(x,R),UnivariatePolynomial(x,R)) -> Integer if R has GCDDOM
--R SturmHabichtSequence : (UnivariatePolynomial(x,R),UnivariatePolynomial(x,R)) -> List(UnivariatePolynomial(x,R))
--R countRealRoots : UnivariatePolynomial(x,R) -> Integer
--R countRealRootsMultiple : UnivariatePolynomial(x,R) -> Integer if R has GCDDOM
--R subresultantSequence : (UnivariatePolynomial(x,R),UnivariatePolynomial(x,R)) -> List(UnivariatePolynomial(x,R))
--R
--E 1

)spool
)lisp (bye)
SturmHabichtPackage (SHP)

Exports:

- SturmHabicht
- SturmHabichtCoefficients
- SturmHabichtMultiple
- SturmHabichtSequence
- countRealRoots
- countRealRootsMultiple
- subresultantSequence

---

---

---

---
CHAPTER 20. CHAPTER S

SturmHabichtPackage(R,x): T == C where
R: OrderedIntegralDomain
x: Symbol

UP ==> UnivariatePolynomial
L ==> List
INT ==> Integer
NNI ==> NonNegativeInteger

T == with
-- subresultantSequenceBegin:(UP(x,R),UP(x,R)) -> L UP(x,R)
-- ++ \spad{subresultantSequenceBegin(p1,p2)} computes the initial terms
-- ++ of the Subresultant sequence Sres(j)(P,deg(P),Q,deg(P)-1)
-- ++ when deg(Q)<deg(P)
-- subresultantSequenceNext:L UP(x,R) -> L UP(x,R)
-- subresultantSequenceInner:(UP(x,R),UP(x,R)) -> L UP(x,R)
subresultantSequence:(UP(x,R),UP(x,R)) -> L UP(x,R)
++ subresultantSequence(p1,p2) computes the (standard)
++ subresultant sequence of p1 and p2
-- sign:R -> R
-- delta:NNI -> R

-- polsth1:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
-- polsth2:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
-- polsth3:(UP(x,R),NNI,UP(x,R),NNI,R) -> L UP(x,R)
SturmHabichtSequence:(UP(x,R),UP(x,R)) -> L UP(x,R)
++ SturmHabichtSequence(p1,p2) computes the Sturm-Habicht
++ sequence of p1 and p2
SturmHabichtCoefficients:(UP(x,R),UP(x,R)) -> L R
++ SturmHabichtCoefficients(p1,p2) computes the principal
++ Sturm-Habicht coefficients of p1 and p2

-- variation:L R -> INT
-- permanence:L R -> INT
-- qzeros:L R -> L R
-- epsil:(NNI,R,R) -> INT
-- numbce:L R -> NNI
-- numb:R -> NNI
-- wfunctaux:L R -> INT
-- wfunct:L R -> INT

SturmHabicht:(UP(x,R),UP(x,R)) -> INT
++ SturmHabicht(p1,p2) computes c_{+}-c_{-} where
++ c_{+} is the number of real roots of p1 with p2>0 and c_{-} is the number of real roots of p1 with p2<0. If p2=1 what
++ you get is the number of real roots of p1.
countRealRoots:(UP(x,R)) -> INT
++ countRealRoots(p) says how many real roots p has
if R has GcdDomain then
SturmHabichtMultiple:(UP(x,R),UP(x,R)) -> INT
\[ \text{countRealRootsMultiple}(p) \text{ says how many real roots } p \text{ has,} \]
\[ \text{countRealRootsMultiple}: (\text{UP}(x, \mathbb{R})) \rightarrow \text{INT} \]
\[ \text{countRealRootsMultiple}(p) \text{ computes } c_{+} - c_{-} \]
\[ c_{+} \text{ is the number of real roots of } p_1 \text{ with } p_2 > 0 \]
\[ c_{-} \text{ is the number of real roots of } p_1 \text{ with } p_2 < 0. \]
\[ \text{countRealRootsMultiple}(p) \text{ you get is the number of real roots of } p_1. \]
c1**((in1-d2)::NNI))::UP(x,R)
pr2:UP(x,R):=
(pseudoRemainder(p1,p2) exquo (-c1)**((in1-d2+2)::NNI))::UP(x,R)
LSub:L UP(x,R):=[pr1,pr2]
for k in ((d2+1)::INT)..((in1-1)::INT) repeat
  LSub:L UP(x,R):=append([0]:L UP(x,R),LSub:L UP(x,R))
append(LcsI:L UP(x,R),LSub:L UP(x,R))

subresultantSequenceInner(p1,p2):L UP(x,R) ==
  Lin:L UP(x,R):=subresultantSequenceBegin(p1:UP(x,R),p2:UP(x,R))
  indf:NNI:= if not(Lin.last::UP(x,R) = 0) then degree(Lin.last::UP(x,R))
              else 0
  while not(indf = 0) repeat
    Lin:L UP(x,R):=subresultantSequenceNext(Lin:L UP(x,R))
    indf:NNI:= if not(Lin.last::UP(x,R)=0) then degree(Lin.last::UP(x,R))
                else 0
    for j in #(Lin:L UP(x,R))..degree(p1) repeat
      Lin:L UP(x,R):=append(Lin:L UP(x,R),[0]:L UP(x,R))
  Lin

-- Computation of the subresultant sequence Sres(j)(P,p,Q,q) when:
-- deg(P) = p and deg(Q) = q and p > q
subresultantSequence(p1,p2):L UP(x,R) ==
  p:NNI:=degree(p1)
  q:NNI:=degree(p2)
  List1:L UP(x,R):=subresultantSequenceInner(p1,p2)
  List2:L UP(x,R):=[p1,p2]
  c1:R:=leadingCoefficientInner(p1)
  for j in 3..#(List1) repeat
    Pr0:UP(x,R):=List1.j
    Pr1:UP(x,R):=(Pr0 exquo c1**((p-q-1)::NNI))::UP(x,R)
    List2:L UP(x,R):=append(List2:L UP(x,R),[Pr1]:L UP(x,R))
  List2

-- Computation of the sign (+1,0,-1) of an element in an ordered integral
-- domain
--
-- sign(r:R):R ==
--  r =$R 0 => 0
--  r >$R 0 => 1
--  -1

-- Computation of the delta function:

delta(int1:NNI):R ==
  (-1)**((int1*(int1+1) exquo 2)::NNI)
-- Computation of the Sturm-Habicht sequence of two polynomials $P$ and $Q$
-- in $R[x]$ where $R$ is an ordered integral domain

polsth1(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
  sc1:R:=(sign(c1))::R
  Pr1:UP(x,R):=pseudoRemainder(differentiate(p1)*p2,p1)
  Pr2:UP(x,R):=(Pr1 exquo c1**(q::NNI))::UP(x,R)
  c2:R:=leadingCoefficient(Pr2)
  r:NNI:=degree(Pr2)
  Pr3:UP(x,R):=monomial(sc1**((p-r-1)::NNI),0)*p1
  Pr4:UP(x,R):=monomial(sc1**((p-r-1)::NNI),0)*Pr2
  Listf:L UP(x,R):=[Pr3,Pr4]
  if r < p-1 then
    Pr5:UP(x,R):=monomial(delta((p-r-1)::NNI)*c2**((p-r-1)::NNI),0)*Pr2
    for j in ((r+1)::INT)..((p-2)::INT) repeat
      Listf:L UP(x,R):=append(Listf:L UP(x,R),[0]:L UP(x,R))
      Listf:L UP(x,R):=append(Listf:L UP(x,R),[Pr5]:L UP(x,R))
  if Pr1=0 then List1:L UP(x,R):=Listf
  else List1:L UP(x,R):=subresultantSequence(p1,Pr2)
  List2:L UP(x,R):=[]
  for j in 0..((r-1)::INT) repeat
    Pr6:UP(x,R):=monomial(delta((p-j-1)::NNI),0)*List1.((p-j+1)::NNI)
    List2:L UP(x,R):=append([Pr6]:L UP(x,R),List2:L UP(x,R))
  append(Listf:L UP(x,R),List2:L UP(x,R))

polsth2(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
  sc1:R:=(sign(c1))::R
  Pr1:UP(x,R):=monomial(sc1,0)*p1
  Pr2:UP(x,R):=differentiate(p1)*p2
  Pr3:UP(x,R):=monomial(sc1,0)*Pr2
  Listf:L UP(x,R):=[Pr1,Pr3]
  List1:L UP(x,R):=subresultantSequence(p1,Pr2)
  List2:L UP(x,R):=[]
  for j in 0..((p-2)::INT) repeat
    Pr4:UP(x,R):=monomial(delta((p-j-1)::NNI),0)*List1.((p-j+1)::NNI)
    Pr5:UP(x,R):=(Pr4 exquo c1)::UP(x,R)
    List2:L UP(x,R):=append([Pr5]:L UP(x,R),List2:L UP(x,R))
  append(Listf:L UP(x,R),List2:L UP(x,R))

polsth3(p1,p:NNI,p2,q:NNI,c1:R):L UP(x,R) ==
  sc1:R:=(sign(c1))::R
  q1:NNI:=(q-1)::NNI
  v:NNI:=(p+q1)::NNI
  Pr1:UP(x,R):=monomial(delta(q1::NNI)*sc1**((q+1)::NNI),0)*p1
  Listf:L UP(x,R):=[Pr1]
  List1:L UP(x,R):=subresultantSequence(differentiate(p1)*p2,p1)
  List2:L UP(x,R):=[]
  for j in 0..((p-1)::NNI) repeat
Pr2:UP(x,R):=monomial(delta((v-j)::NNI),0)*List1.((v-j+1)::NNI)
Pr3:UP(x,R):=(Pr2 exquo c1)::UP(x,R)
List2:L UP(x,R):=append([Pr3]:L UP(x,R),List2:L UP(x,R))
append(Listf:L UP(x,R),List2:L UP(x,R))

SturmHabichtSequence(p1,p2):L UP(x,R) ==
p:NNI:=degree(p1)
q:NNI:=degree(p2)
c1:R:=leadingCoefficient(p1)
c1 = 1 or q = 1 => polsth1(p1,p,p2,q,c1)
q = 0 => polsth2(p1,p,p2,q,c1)
polsth3(p1,p,p2,q,c1)

-- Computation of the Sturm-Habicht principal coefficients of two
-- polynomials P and Q in R[x] where R is an ordered integral domain
SturmHabichtCoefficients(p1,p2):L R ==
List1:L UP(x,R):=SturmHabichtSequence(p1,p2)
List2:L R:=[]
qp:NNI:=#(List1)::NNI
[coefficient(p,(qp-j)::NNI) for p in List1 for j in 1..qp]
for j in 1..qp repeat
  Ply:R:=coefficient(List1.j,(qp-j)::NNI)
  List2:L R:=append(List2,[Ply])
-- List2

-- Computation of the number of sign variations of a list of non zero
-- elements in an ordered integral domain
variation(Lsig:L R):INT ==
  size?(Lsig,1) => 0
  elt1:R:=first Lsig
  elt2:R:=Lsig.2
  sig1:R:=(sign(elt1*elt2))::R
  List1:L R:=rest Lsig
  sig1 = 1 => variation List1
  1+variation List1

-- Computation of the number of sign permanences of a list of non zero
-- elements in an ordered integral domain
permanence(Lsig:L R):INT ==
  size?(Lsig,1) => 0
  elt1:R:=first Lsig
  elt2:R:=Lsig.2
  sig1:R:=(sign(elt1*elt2))::R
  List1:L R:=rest Lsig
sig1 = -1 => permanence List1
1+permanence List1

-- Computation of the functional \( W \) which works over a list of elements
-- in an ordered integral domain, with non zero first element

qzeros(Lsig:L R):L R ==
while last Lsig = 0 repeat
   Lsig:L R:=reverse rest reverse Lsig
   Lsig

epsil(int1:NNI,elt1:R,elt2:R):INT ==
   int1 = 0 => 0
   odd? int1 => 0
   ct1:INT:=if elt1 > 0 then 1 else -1
   ct2:INT:=if elt2 > 0 then 1 else -1
   ct3:NNI:=(int1 exquo 2)::NNI
   ct4:INT:=(ct1*ct2)::INT
   (-1)**(ct3::NNI))*ct4

numbnc(Lsig:L R):NNI ==
   null Lsig => 0
   eltp:=Lsig.1
   eltp = 0 => 0
   1 + numbnc(rest Lsig)

numbce(Lsig:L R):NNI ==
   null Lsig => 0
   eltp:=Lsig.1
   not(eltp = 0) => 0
   1 + numbce(rest Lsig)

wfunctaux(Lsig:L R):INT ==
   null Lsig => 0
   List2:L R:=[]
   List1:L R:=Lsig:L R
   cont1:NNI:=numbce(List1:L R)
   for j in 1..cont1 repeat
      List2:L R:=append(List2:L R,[first List1]:L R)
      List1:L R:=rest List1
      ind2:INT:=0
      cont2:NNI:=numbce(List1:L R)
      for j in 1..cont2 repeat
         List1:L R:=rest List1
         ind2:=epsil(cont2:NNI,last List2,first List1)
      ind3:INT:=permanence(List2:L R)-variation(List2:L R)
      ind4:INT:=ind2+ind3
      ind4+wfunctaux(List1:L R)
\texttt{wfunct(Lsig:L R):INT ==}
\texttt{List1:L R:=qzeros(Lsig:L R)}
\texttt{wfunctaux(List1:L R)}

\textbf{-- Computation of the integer number:}
\textbf{--} \#\{a \in R/\text{P}(a)=0 \bar{Q}(a)>0\} - \#\{a \in R/\text{P}(a)=0 \bar{Q}(a)<0\}
\textbf{-- where:}
\textbf{--} - R is an ordered integral domain,
\textbf{--} - R is the real closure of R,
\textbf{--} - P and Q are polynomials in R[x],
\textbf{--} - by \#[A] we note the cardinal of the set A

\textbf{-- In particular:}
\textbf{--} - SturmHabicht(P,1) is the number of "real" roots of P,
\textbf{--} - SturmHabicht(P,Q*2) is the number of "real" roots of P making Q neq 0

\texttt{SturmHabicht(p1,p2):INT ==}
\texttt{-- print("+" :: Ex)
  p2 = 0 => 0}
\texttt{degree(p1:UP(x,R)) = 0 => 0}
\texttt{List1:L UP(x,R):=SturmHabichtSequence(p1,p2)}
\texttt{qp:NNI:=\#(List1)::NNI}
\texttt{wfunct [coefficient(p,(qp-j)::NNI) for p in List1 for j in 1..qp]}

\texttt{countRealRoots(p1):INT == SturmHabicht(p1,1)}

if R has GcdDomain then
\texttt{SturmHabichtMultiple(p1,p2):INT ==}
\texttt{-- print("+" :: Ex)
  p2 = 0 => 0}
\texttt{degree(p1:UP(x,R)) = 0 => 0}
\texttt{SH:L UP(x,R):=SturmHabichtSequence(p1,p2)}
\texttt{qp:NNI:=\#(SH)::NNI}
\texttt{ans:= wfunct [coefficient(p,(qp-j)::NNI) for p in SH for j in 1..qp]}
\texttt{SH:=reverse SH}
\texttt{while first SH = 0 repeat SH:=rest SH}
\texttt{degree first SH = 0 => ans}
\texttt{-- OK: it probably wasn't square free, so this item is probably the}
\texttt{-- gcd of p1 and p1'}
\texttt{-- unless p1 and p2 have a factor in common (naughty!)}
\texttt{ differentiate(p1) exquo first SH case UP(x,R) =>}
\texttt{ -- it was the gcd of p1 and p1'}
\texttt{ ans+SturmHabichtMultiple(first SH,p2)}
\texttt{sqfr:=factorList squareFree p1}
\texttt{#sqfr = 1 and sqfr.first.xpnt=1 => ans}
\texttt{reduce("+",[f.xpnt*SturmHabicht(f.fctr,p2) for f in sqfr])}

\texttt{countRealRootsMultiple(p1):INT == SturmHabichtMultiple(p1,1)}
package SUBRESP SubResultantPackage

-- SubResultantPackage.input --

)set break resume
)sys rm -f SubResultantPackage.output
)spool SubResultantPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SubResultantPackage
--R
--R SubResultantPackage(R: IntegralDomain,UP: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for SubResultantPackage is SUBRESP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SUBRESP
--R
--R---------------------------------------- Operations ----------------------------------------
--R primitivePart : (UP,R) -> UP if R has EUCDOM
--R subresultantVector : (UP,UP) -> PrimitiveArray(UP)
--R
--E 1

)spool
)lisp (bye)

SubResultantPackage examples
This package computes the subresultants of two polynomials which is needed for the ‘Lazard Rioboo’ enhancement to Trager’s integrations formula. For efficiency reasons this has been rewritten to call Lionel Ducos package which is currently the best one.

See Also:
- )show SubResultantPackage

---

SubResultantPackage (SUBRESP)

Exports:
- primitivePart
- subresultantVector

--- package SUBRESP SubResultantPackage ---

)abbrev package SUBRESP SubResultantPackage
++ Author: Barry Trager, Renaud Rioboo
++ Date Created: 1987
++ Date Last Updated: August 2000
++ Description:
++ This package computes the subresultants of two polynomials which is needed + for the ‘Lazard Rioboo’ enhancement to Trager’s integrations formula
++ For efficiency reasons this has been rewritten to call Lionel Ducos
++ package which is currently the best one.

SubResultantPackage(R, UP): Exports == Implementation where
  R : IntegralDomain
  UP: UnivariatePolynomialCategory R
Z ==> Integer
N ==> NonNegativeInteger

Exports ==> with
subresultantVector: (UP, UP) -> PrimitiveArray UP
  ++ subresultantVector(p, q) returns \spad{\{p_0, \ldots, p_n\}}
  ++ where \(p_i\) is the \(i\)-th subresultant of \(p\) and \(q\).
  ++ In particular, \spad{p_0 = \text{resultant}(p, q)}.
if R has EuclideanDomain then
  primitivePart : (UP, R) -> UP
    ++ primitivePart(p, q) reduces the coefficient of \(p\)
    ++ modulo \(q\), takes the primitive part of the result,
    ++ and ensures that the leading coefficient of that
    ++ result is monic.

Implementation ==> add

Lionel ==> PseudoRemainderSequence(R,UP)

if R has EuclideanDomain then
  primitivePart(p, q) ==
    rec := extendedEuclidean(leadingCoefficient p, q, 1)::Record(coef1:R, coef2:R)
    unitCanonical primitivePart map(x1 +-> (rec.coef1 * x1) rem q, p)

subresultantVector(p1, p2) ==
  F : UP -- auxiliary stuff !
  res : PrimitiveArray(UP) := new(2+max(degree(p1),degree(p2)), 0)
  --
  -- kind of stupid interface to Lionel's Package !!!!!!!!!!!!!
  -- might have been wiser to rewrite the loop ...
  -- But I'm too lazy. [rr]
  --
  l := chainSubResultants(p1,p2)$Lionel
  --
  -- this returns the chain of non null subresultants !
  -- we must rebuild subresultants from this.
  -- we really hope Lionel Ducos minded what he wrote
  -- since we are fully blind !
  --
  null l =>
    -- Hum it seems that Lionel returns [] when \min(|p1|,|p2|) = 0
    zero?(degree(p1)) =>
      res.degree(p2) := p2
      if degree(p2) > 0 then
        res.((degree(p2)-1)::NonNegativeInteger) := p1
        res.0 := (leadingCoefficient(p1)**(degree p2)) :: UP
      else
        -- both are of degree 0 the resultant is 1 according to Loos
res.0 := 1
res
zero?(degree(p2)) =>
  if degree(p1) > 0
  then
    res.((degree(p1)-1)::NonNegativeInteger) := p2
    res.0 := (leadingCoefficient(p2)**(degree p1)) :: UP
  else
    -- both are of degree 0 the resultant is 1 according to Loos
    res.0 := 1
res
error "SUBRESP: strange Subresultant chain from PRS"
Sn := first(l)
--
-- as of Loos definitions last subresultant should not be defective
--
l := rest(l)
n := degree(Sn)
F := Sn
null l => error "SUBRESP: strange Subresultant chain from PRS"
zero? Sn => error "SUBRESP: strange Subresultant chain from PRS"
while (l ^= []) repeat
  res.(n) := Sn
  F := first(l)
l := rest(l)
  -- F is potentially defective
  if degree(F) = n
  then
    -- -- F is defective
    --
    null l => error "SUBRESP: strange Subresultant chain from PRS"
    Sn := first(l)
l := rest(l)
n := degree(Sn)
    res.((n-1)::NonNegativeInteger) := F
  else
    -- -- F is non defective
    --
    degree(F) < n => error "strange result !"
    Sn := F
    n := degree(Sn)
    --
    -- Lionel forgets about p1 if |p1| > |p2|
    -- forgets about p2 if |p2| > |p1|
    -- but he reminds p2 if |p1| = |p2|
    -- a glance at Loos should correct this !
    --
    res.n := Sn
-- Loos definition
if degree(p1) = degree(p2)
then
  res.((degree p1)+1) := p1
else
  if degree(p1) > degree(p2)
    then
      res.((degree p1)) := p1
    else
      res.((degree p2)) := p2
res

---

-- SUBRESP.dotabb --
"SUBRESP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUBRESP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SUBRESP" -> "PFECAT"

---

package SUPFRACF SupFractionFactorizer

--- SupFractionFactorizer.input ---

)set break resume
)sys rm -f SupFractionFactorizer.output
)spool SupFractionFactorizer.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SupFractionFactorizer
--R
--R SupFractionFactorizer(E: OrderedAbelianMonoidSup,OV: OrderedSet,R: GcdDomain,P: PolynomialCategory(R,E,OV)) is a package constructor
--R Abbreviation for SupFractionFactorizer is SUPFRACF
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SUPFRACF
--R
--R-------------------------- Operations --------------------------
SUPFRACF  

Exports:  
  factor    squareFree

--- package SUPFRACF SupFractionFactorizer ---

)abbrev package SUPFRACF SupFractionFactorizer
+ Author: P. Gianni
+ Date Created: October 1993
+ Date Last Updated: March 1995
+ Description:
+ SupFractionFactorize contains the factor function for univariate
+ polynomials over the quotient field of a ring S such that the package
+ MultivariateFactorize works for S.

SupFractionFactorizer(E,OV,R,P) : C == T
where
  E : OrderedAbelianMonoidSup
  OV : OrderedSet
  R : GcdDomain
  P : PolynomialCategory(R,E,OV)
  FP ==> Fraction P
  SUP ==> SparseUnivariatePolynomial

C == with
  factor : SUP FP -> Factored SUP FP
    ++ factor(p) factors the univariate polynomial p with coefficients
    ++ which are fractions of polynomials over R.
  squareFree : SUP FP -> Factored SUP FP
    ++ squareFree(p) returns the square-free factorization of the univariate polynomial p with coefficients
    ++ which are fractions of polynomials over R. Each factor has no repeated roots and the factors are
    ++ pairwise relatively prime.

T == add
  MFACT ==> MultivariateFactorize(OV,E,R,P)
  MSQFR ==> MultivariateSquareFree(E,OV,R,P)
  UPCF2 ==> UnivariatePolynomialCategoryFunctions2

factor(p:SUP FP) : Factored SUP FP ==
  p=0 => 0
  R has CharacteristicZero and R has EuclideanDomain =>
    pden : P := lcm [denom c for c in coefficients p]
    pol : SUP FP := (pdens::FP)*p
    ipol: SUP P := map(numer,pol)$UPCF2(FP,SUP FP,P,SUP P)
    ffact: Factored SUP P := 0
    ffact := factor(ipol)$MFACT
    makeFR((1/pden * map(coerce,unit ffact)$UPCF2(P,SUP P,FP,SUP FP)),
      [["prime",map(coerce,u.factor)$UPCF2(P,SUP P,FP,SUP FP),
        u.exponent] for u in factors ffact])
  squareFree p

squareFree(p:SUP FP) : Factored SUP FP ==
  p=0 => 0
  pden : P := lcm [denom c for c in coefficients p]
  pol : SUP FP := (pdens::FP)*p
  ipol: SUP P := map(numer,pol)$UPCF2(FP,SUP FP,P,SUP P)
  ffact: Factored SUP P := 0
if R has CharacteristicZero and R has EuclideanDomain then
  ffact := squareFree(ipol)$MSQFR
else ffact := squareFree(ipol)
makeFR((1/pden * map(coerce,unit ffact)$UPCF2(P,SUP P,FP,SUP FP)),
  [["sqfr",map(coerce,u.factor)$UPCF2(P,SUP P,FP,SUP FP),
    u.exponent] for u in factors ffact])

— SUPFRACF.dotabb —

"SUPFRACF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SUPFRACF"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"SUPFRACF" -> "PFECAT"

package ODESYS SystemODESolver

— SystemODESolver.input —

)set break resume
)sys rm -f SystemODESolver.output
)spool SystemODESolver.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SystemODESolver
--R
--R SystemODESolver(F: Field,LO: LinearOrdinaryDifferentialOperatorCategory(F)) is a package constructor
--R Abbreviation for SystemODESolver is ODESYS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ODESYS
--R
--R------------------------------- Operations --------------------------------
--R solve : (Matrix(F),Vector(F),((LO,F) -> Union(Record(particular: F,basis: List(F)),"failed"))
--R solveInField : (Matrix(LO),Vector(F),((LO,F) -> Record(particular: Union(F,"failed"),basis:
--R triangulate : (Matrix(F),Vector(F)) -> Record(A: Matrix(F),eqs: List(Record(C: Matrix(F),g:
--R triangulate : (Matrix(LO),Vector(F)) -> Record(mat: Matrix(LO),vec: Vector(F))
--R
--E 1
SystemODESolver (ODESYS)

Exports:
- solve
- solveInField
- triangulate

---

package ODESYS SystemODESolver

++) abbrev package ODESYS SystemODESolver
++) Author: Manuel Bronstein
++) Date Created: 11 June 1991
++) Date Last Updated: 13 April 1994
++) Description:
++) SystemODESolver provides tools for triangulating
and solving some systems of linear ordinary differential equations.

SystemODESolver(F, LO): Exports == Implementation where
  F : Field
  LO: LinearOrdinaryDifferentialOperatorCategory F

N  ==> NonNegativeInteger
Z  ==> Integer
MF ==> Matrix F
M  ==> Matrix LO
V  ==> Vector F
UF ==> Union(F, "failed")
UV ==> Union(V, "failed")
REC ==> Record(mat: M, vec: V)
FSL ==> Record(particular: UF, basis: List F)
VSL ==> Record(particular: UV, basis: List V)
SOL ==> Record(particular: F, basis: List F)
USL ==> Union(SOL, "failed")
ER ==> Record(C: MF, g: V, eq: LO, rh: F)

Exports ==> with
  triangulate: (MF, V) -> Record(A:MF, eqs: List ER)
  triangulate: (M, V) -> REC
  solve: (MF,V,(LO,F)->USL) -> Union(Record(particular:V, basis:MF),"failed")
  solveInField: (M, V, (LO,F) -> FSL) -> VSL

Implementation ==> add
import PseudoLinearNormalForm F

applyLodo : (M, Z, V, N) -> F
applyLodo0 : (M, Z, Matrix F, Z, N) -> F
backsolve : (M, V, (LO, F) -> FSL) -> VSL
firstnonzero: (M, Z) -> Z
FSL2USL : FSL -> USL
M2F : M -> Union(MF, "failed")

diff := D()$LO

solve(mm, v, solve) ==
  rec := triangulate(mm, v)
  sols:List(SOL) := empty()
  for e in rec.eqs repeat
    (u := solve(e.eq, e.rh)) case "failed" => return "failed"
    sols := concat(u::SOL, sols)
  n := nrows(rec.A) -- dimension of original vectorspace
  k:N := 0 -- sum of sizes of visited companionblocks
  i:N := 0 -- number of companionblocks
  m:N := 0 -- number of Solutions
  part:V := new(n, 0)
  -- count first the different solutions
  for sol in sols repeat
    m := m + count((f1:F):Boolean +-> f1 ^= 0, sol.basis)$List(F)
  SolMatrix:MF := new(n, m, 0)
  m := 0
  for sol in reverse_! sols repeat
    i := i+1
    er := rec.eqs.i
    nn := #(er.g) -- size of active companionblock
    for s in sol.basis repeat
      solVec:V := new(n, 0)
      -- compute corresponding solution base with recursion (24)
      solVec(k+1) := s
      for l in 2..nn repeat solVec(k+l) := diff solVec(k+l-1)
      m := m+1
      setColumn!(SolMatrix, m, solVec)
    -- compute with (24) the corresponding components of the part. sol.
    part(k+1) := sol.particular
    for l in 2..nn repeat part(k+l) := diff part(k+l-1) - (er.g)(l-1)
    k := k+nn
    -- transform these values back to the original system

triangulate(m:MF, v:V) ==
  k:N := 0 -- sum of companion-dimensions
  rat := normalForm(m, 1, (f1:F):F ++-> - diff f1)
  l := companionBlocks(rat.R, rat.Ainv * v)
  ler:List(ER) := empty()
for er in l repeat
  n := nrow(er.C) -- dimension of this companion vectorspace
  op:LO := 0 -- compute homogeneous equation
  for j in 0..n-1 repeat 
    op := op + monomial((er.C)(n, j + 1), j)
  end for
  op := monomial(1, n) - op

  sum:V := new(n::N, 0) -- compute inhomogen Vector (25)
  for j in 1..n-1 repeat
    sum(j+1) := diff(sum j) + (er.g) j
  end for
  h0:F := 0 -- compute inhomogenity (26)
  for j in 1..n repeat
    h0 := h0 - (er.C)(n, j) * sum j
  end for
  h0 := h0 + diff(sum n) + (er.g) n
  ler := concat([er.C, er.g, op, h0], ler)
  k := k + n
end for er

-- like solveInField, but expects a system already triangularized
backsolve(m, v, solve) ==
  part:V
  r := maxRowIndex m
  offset := minIndex v - (mr := minRowIndex m)
  while r >= mr and every?(zero?, row(m, r))$Vector(LO) repeat r := r - 1

  if r < mr => error "backsolve: system has a 0 matrix"
  c := firstnonzero(m, r) ^= maxColIndex m =>
    error "backsolve: undetermined system"
  rec := solve(m(r, c), v(r + offset))

  dim := (r - mr + 1)::N
  if part? := ((u := rec.particular) case F)) then
    part := new(dim, 0) -- particular solution
    part(r + offset) := u::F
  end if

-- hom is the basis for the homogeneous solutions, each column is a solution
  hom:Matrix(F) := new(dim, #(rec.basis), 0)
  for i in minColIndex hom .. maxColIndex hom for b in rec.basis repeat
    hom(r, i) := b
  end for
  n:N := 1 -- number of equations already solved
  while r > mr repeat
    r := r - 1
    c := c - 1
    if firstnonzero(m, r) ^= c => error "backsolve: undetermined system"
    degree(eq := m(r, c)) > 0 => error "backsolve: pivot of order > 0"
    a := leadingCoefficient(eq):F
    if part? then
      part(r + offset) := (v(r + offset) - applyLodo(m, r, part, n)) / a
    end if
    for i in minColIndex hom .. maxColIndex hom repeat
      hom(r, i) := - applyLodo0(m, r, hom, i, n)
    end for
    n := n + 1
  end while

  bas:List(V) := [column(hom,i) for i in minColIndex hom..maxColIndex hom]
  if part? then
    bas := [part, bas]
  end if

  solveInField(m, v, solve) ==
    (((n := nrows m) = ncols m) and

((u := M2F(diagonalMatrix [diff for i in 1..n] - m)) case MF) =>
  (uu := solve(u::MF, v, (l1:LO,f2:F):USL +-> FSL2USL solve(l1, f2))) case "failed" =>
    ["failed", empty()]
  rc := uu::Record(particular:V, basis:MF)
  [rc.particular, [column(rc.basis, i) for i in 1..ncols(rc.basis)]]
rec := triangulate(m, v)
backsolve(rec.mat, rec.vec, solve)

M2F m ==
mf:MF := new(nrows m, ncols m, 0)
for i in minRowIndex m .. maxRowIndex m repeat
  for j in minColIndex m .. maxColIndex m repeat
    (u := retractIfCan(m(i, j))@Union(F, "failed")) case "failed" =>
      return "failed"
    mf(i, j) := u::F
mf

FSL2USL rec ==
  rec.particular case "failed" => "failed"
  [rec.particular::F, rec.basis]

-- returns the index of the first nonzero entry in row r of m
firstnonzero(m, r) ==
  for c in minColIndex m .. maxColIndex m repeat
    m(r, c) ^= 0 => return c
  error "firstnonzero: zero row"

-- computes +/[m(r, i) v(i) for i ranging over the last n columns of m]
applyLodo(m, r, v, n) ==
  ans:F := 0
c := maxColIndex m
cv := maxIndex v
for i in 1..n repeat
  ans := ans + m(r, c) (v cv)
c := c - 1
cv := cv - 1
ans

-- computes +/[m(r, i) mm(i, c) for i ranging over the last n columns of m]
applyLodo0(m, r, mm, c, n) ==
  ans := 0
rr := maxRowIndex mm
cc := maxColIndex m
for i in 1..n repeat
  ans := ans + m(r, cc) mm(rr, c)
c := cc - 1
rr := rr - 1
ans
triangulate(m:M, v:V) ==
  x := copy m
  w := copy v
  nrows := maxRowIndex x
  ncols := maxColIndex x
  minr := i := minRowIndex x
  offset := minIndex w - minr
  for j in minColIndex x .. ncols repeat
    if i > nrows then leave x
    rown := minr - 1
    for k in i .. nrows repeat
      if (x(k, j) ^= 0) and ((rown = minr - 1) or
        degree x(k,j) < degree x(rown,j)) then rown := k
    rown = minr - 1 => "enuf"
    x := swapRows_!(x, i, rown)
    swap_!(w, i + offset, rown + offset)
  for k in i+1 .. nrows | x(k, j) ^= 0 repeat
    l := rightLcm(x(i,j), x(k,j))
    a := rightQuotient(l, x(i, j))
    b := rightQuotient(l, x(k, j))
    -- l = a x(i,j) = b x(k,j)
    for k1 in j+1 .. ncols repeat
      x(k, k1) := a * x(i, k1) - b * x(k, k1)
      x(k, j) := 0
    w(k + offset) := a(w(i + offset)) - b(w(k + offset))
  i := i+1
  [x, w]

——

— ODESYS.dotabb —

"ODESYS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ODESYS"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"ODESYS" -> "IVECTOR"

——

package SYSSOLP SystemSolvePackage

— SystemSolvePackage.input —

)set break resume
)sys rm -f SystemSolvePackage.output
--S 1 of 1
)show SystemSolvePackage

--)SystemSolvePackage(R: IntegralDomain) is a package constructor
--)Abbreviation for SystemSolvePackage is SYSSOLP
--)This constructor is exposed in this frame.
--)Issue )edit bookvol10.4.pamphlet to see algebra source code for SYSSOLP

--)Operations ---------------------------------.

--)solve : (List(Fraction(Polynomial(R))),List(Symbol)) -> List(List(Equation(Fraction(Polynomial(R))))))
--)solve : (List(Equation(Fraction(Polynomial(R))))) List(Symbol)) -> List(List(Equation(Fraction(Polynomial(R))))))
--)solve : List(Fraction(Polynomial(R))) -> List(List(Equation(Fraction(Polynomial(R))))))
--)solve : List(Equation(Fraction(Polynomial(R))))) List(Symbol)) -> List(List(Equation(Fraction(Polynomial(R))))))
--)solve : (Equation(Fraction(Polynomial(R))),Symbol) -> List(Equation(Fraction(Polynomial(R))))))
--)solve : Equation(Fraction(Polynomial(R))) Equteation(Polynomial(R))))
--)solve : Equation(Fraction(Polynomial(R))) Equteation(Polynomial(R))))

--)triangularSystems : (List(Fraction(Polynomial(R))),List(Symbol)) -> List(List(Polynomial(R))))

--)E 1

)spool
)lisp (bye)

SystemSolvePackage.help —

SystemSolvePackage examples

SystemSolvePackage examples

Symbolic solver for systems of rational functions with coefficients
in an integral domain R.

The systems are solved in the field of rational functions over R.
Solutions are exact of the form variable = value when the value is a
member of the coefficient domain R. Otherwise the solutions are
implicitly expressed as roots of univariate polynomial equations over R.

Care is taken to guarantee that the denominators of the input
equations do not vanish on the solution sets.

The arguments to solve can either be given as equations or as rational
functions interpreted as equal to zero. The user can specify an explicit list of symbols to be solved for, treating all other symbols appearing as parameters or omit the list of symbols in which case the system tries to solve with respect to all symbols appearing in the input.

See Also:
- )show SystemSolvePackage

---

SystemSolvePackage (SYSSOLP)

Exports:
solve triangularSystems

--- package SYSSOLP SystemSolvePackage ---

)abbrev package SYSSOLP SystemSolvePackage
++ Author: P. Gianni
++ Date Created: summer 1988
++ Date Last Updated: summer 1990
++ Description:
++ Symbolic solver for systems of rational functions with coefficients in an integral domain R.
++ The systems are solved in the field of rational functions over R.
++ Solutions are exact of the form variable = value when the value is a member of the coefficient domain R. Otherwise the solutions are implicitly expressed as roots of univariate polynomial equations over R.
++ Care is taken to guarantee that the denominators of the input equations do not vanish on the solution sets.
++ The arguments to solve can either be given as equations or as rational functions interpreted as equal
++ to zero. The user can specify an explicit list of symbols to
++ be solved for, treating all other symbols appearing as parameters
++ or omit the list of symbols in which case the system tries to
++ solve with respect to all symbols appearing in the input.

SystemSolvePackage(R): Cat == Cap where

  NNI ==> NonNegativeInteger
  P ==> Polynomial
  EQ ==> Equation
  L ==> List
  V ==> Vector
  M ==> Matrix
  UP ==> SparseUnivariatePolynomial
  SE ==> Symbol
  IE ==> IndexedExponents Symbol
  SUP ==> SparseUnivariatePolynomial

  R : IntegralDomain
  F ==> Fraction Polynomial R
  PP2 ==> PolynomialFunctions2(R,F)
  PPR ==> Polynomial Polynomial R

Cat == with

  solve: (L F, L SE) -> L L EQ F
  ++ solve(lp,lv) finds the solutions of the list lp of
  ++ rational functions with respect to the list of symbols lv.

  solve: (L EQ F, L SE) -> L L EQ F
  ++ solve(le,lv) finds the solutions of the
  ++ list le of equations of rational functions
  ++ with respect to the list of symbols lv.

  solve: L F -> L L EQ F
  ++ solve(lp) finds the solutions of the list lp of rational
  ++ functions with respect to all symbols appearing in lp.

  solve: L EQ F -> L L EQ F
  ++ solve(le) finds the solutions of the list le of equations of
  ++ rational functions with respect to all symbols appearing in le.

  solve: (F, SE) -> L EQ F
  ++ solve(p,v) solves the equation p=0, where p is a rational function
  ++ with respect to the variable v.

  solve: (EQ F,SE) -> L EQ F
  ++ solve(eq,v) finds the solutions of the equation
  ++ eq with respect to the variable v.

  solve: F -> L EQ F
  ++ solve(p) finds the solution of a rational function p = 0
++ with respect to the unique variable appearing in p.

solve: EQ F -> L EQ F
++ solve(eq) finds the solutions of the equation eq
++ with respect to the unique variable appearing in eq.

triangularSystems: (L F, L SE) -> L L P R
++ triangularSystems(lf,lv) solves the system of equations
++ defined by lf with respect to the list of symbols lv;
++ the system of equations is obtaining
++ by equating to zero the list of rational functions lf.
++ The output is a list of solutions where
++ each solution is expressed as a "reduced" triangular system of
++ polynomials.

Cap == add

import MPolyCatRationalFunctionFactorizer(IE,SE,R,P F)

---- Local Functions ----
linSolve: (L F, L SE) -> Union(L EQ F, "failed")
makePolys : L EQ F -> L F
makeR2F(r : R) : F == r :: (P R) :: F
makeP2F(p:P F):F ==
  lv:=variables p
  lv = [] => retract p
  for v in lv repeat p:=pushdown(p,v)
  retract p

---- Local Functions ----
makeEq(p:P F,lv:L SE): EQ F ==
  z:=last lv
  np:=numer makeP2F p
  lx:=variables np
  for x in lv repeat if member?(x,lx) then leave x
  up:=univariate(np,x)
  (degree up)=1 =>
    equation(x::P(R)::F,-coefficient(up,0)/leadingCoefficient up)
  equation(np::F,0$F)

varInF(v: SE): F == v::P(R) :: F

newInF(n: Integer):F==varInF new()$SE

testDegree(f :P R , lv :L SE) : Boolean ==
  "or"/[degree(f,vv)>0 for vv in lv]

---- Exported Functions ----

-- solve a system of rational functions
triangularSystems(lf: L F, lv: L SE) : L L P R ==
  empty? lv => empty()
  empty? lf => empty()
  #lf = 1 =>
    p := numer(first lf)
    fp:=(factor p)$GeneralizedMultivariateFactorize(SE,IE,R,R,P R)
    [[ff.factor] for ff in factors fp | testDegree(ff.factor, lv)]
  dmp:=DistributedMultivariatePolynomial(lv,P R)
  OV:=OrderedVariableList(lv)
  DP:=DirectProduct(#lv, NonNegativeInteger)
  push:=PushVariables(R,DP,OV,dmp)
  lq : L dmp
  lvv:L OV:=[variable(vv)::OV for vv in lv]
  lq:=[pushup(df::dmp,lvv)$push for f in lf | (df:=denom f)^=1]
  lp:=[pushup(numer(f)::dmp,lvv)$push for f in lf]
  parRes:=groebSolve(lp,lvv)$GroebnerSolve(lv,P R,R)
  if lq=[] then
    gb:=GroebnerInternalPackage(P R,DirectProduct(#lv, NNI),OV,dmp)
    parRes:=[pr for pr in parRes | and/[(redPol(fq, pr pretend List(dmp))$gb) ^=0 for fq in lq]]
    [[retract pushdown(pf,lvv)$push for pf in pr] for pr in parRes]

-- One polynomial. Implicit variable --
solve(pol : F) ==
  zero? pol =>
    error "equation is always satisfied"
  lv:=removeDuplicates
    concat(variables numer pol, variables denom pol)
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  solve(pol,first lv)

-- general solver. Input in equation style. Implicit variables --
solve(eq : EQ F) ==
  pol:= lhs eq - rhs eq
  zero? pol =>
    error "equation is always satisfied"
  lv:=removeDuplicates
    concat(variables numer pol, variables denom pol)
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  solve(pol,first lv)

-- general solver. Input in equation style --
solve(eq:EQ F, var:SE) == solve(lhs eq - rhs eq, var)

-- general solver. Input in polynomial style --
solve(pol:F, var:SE) ==
  if R has GcdDomain then
\[
\text{p} := \text{primitivePart}(\text{numer pol}, \text{var})
\]
\[
\text{fp} := (\text{factor p}) \cdot \text{GeneralizedMultivariateFactorize}(\text{SE}, \text{IE}, \text{R}, \text{R}, \text{PR})
\]
\[
[\text{makeEq}(\text{map}(\text{makeR2F}, \text{ff.factor}) \cdot \text{PP2}, [\text{var}]) \text{ for } \text{ff in factors fp}]
\]
\[
\text{else empty()}
\]

\[
\text{-- Convert a list of Equations in a list of Polynomials}
\]
\[
\text{makePolys}(1 : \text{LEQF}) : \text{L F} \Rightarrow [\text{lhs e - rhs e for e in l}]
\]

\[
\text{-- linear systems solver. Input as list of polynomials --}
\]
\[
\text{linSolve}(1 : \text{L F}, 1 : \text{L SE}) :=
\]
\[
\text{rec: Record(\text{particular: Union(V F, "failed"), basis: L V F})}
\]
\[
\text{lr} : \text{L P R} := [\text{numer f for f in lp}]
\]
\[
\text{rec} := \text{linSolve}(\text{lr}, \text{lv}) \cdot \text{LinearSystemPolynomialPackage} (\text{R}, \text{IE}, \text{SE}, \text{PR})
\]
\[
\text{rec.particular case "failed" \Rightarrow "failed"}
\]
\[
\text{rhs} := \text{rec.particular :: V F}
\]
\[
\text{zeron: V F} := \text{zero(#lv)}
\]
\[
\text{for p in rec.basis | p \Rightarrow zeron repeat}
\]
\[
\text{sym} := \text{newInF(1)}
\]
\[
\text{for i in 1 .. #lv repeat}
\]
\[
\text{rhs.i} := \text{rhs.i + sym*p.i}
\]
\[
\text{eqs: L EQ F} := []
\]
\[
\text{for i in 1 .. #lv repeat}
\]
\[
\text{eqs} := \text{append(eqs, [(lv.i)::(PR)::F = rhs.i])}
\]
\[
\text{eqs}
\]

\[
\text{-- general solver. Input in polynomial style. Implicit variables --}
\]
\[
\text{solve(1r : L F) :=
\]
\[
\text{lv} := "\text{setUnion}"/[\text{setUnion}(\text{variables} \text{numer p}, \text{variables} \text{denom p})
\]
\[
\text{for p in 1r]
\]
\[
\text{solve(1r, lv)}
\]

\[
\text{-- general solver. Input in equation style. Implicit variables --}
\]
\[
\text{solve(1e : L EQ F) :=
\]
\[
\text{lr: = makePolys 1e}
\]
\[
\text{lv} := "\text{setUnion}"/[\text{setUnion}(\text{variables} \text{numer p}, \text{variables} \text{denom p})
\]
\[
\text{for p in 1r]
\]
\[
\text{solve(1r, lv)}
\]

\[
\text{-- general solver. Input in equation style --}
\]
\[
\text{solve(1e: L EQ F, lv: L SE) \Rightarrow solve(makePolys 1e, lv)}
\]

\[
\text{checkLinear(1r: L F, lv: L SE): Boolean} :=
\]
\[
\text{ld:=[denom pol for pol in 1r]}
\]
\[
\text{for f in ld repeat}
\]
\[
\text{if (or\[/member?\{x, lv\} for x in variables f]) then return false}
\]
\[
\text{and/[totalDegree(numer pol, vl) < 2 for pol in 1r]}
\]

\[
\text{-- general solver. Input in polynomial style --}
\]
\[
\text{solve(1r: L F, lv: L SE) :=
\]
\[
\text{empty? lv => empty()}
\]
checkLinear(lr,vl) =>
  -- linear system --
  soln := linSolve(lr, vl)
  soln case "failed" => []
  eqns: L EQ F := []
  for i in 1..#vl repeat
    lhs := (vl.i::(P R))::F
    rhs := rhs soln.i
    eqns := append(eqns, [lhs = rhs])
  [eqns]

  -- polynomial system --
  if R has GcdDomain then
    parRes:=triangularSystems(lr,vl)
    [[makeEq(map(makeR2F,f)$PP2,vl) for f in pr]
     for pr in parRes]
  else []

package SGCF SymmetricGroupCombinatoricFunctions

-- SymmetricGroupCombinatoricFunctions.input --

)set break resume
)sys rm -f SymmetricGroupCombinatoricFunctions.output
)spool SymmetricGroupCombinatoricFunctions.output
)set message test on
)set message auto off
)clear all
--S 1 of 1
)show SymmetricGroupCombinatoricFunctions
--R
--R SymmetricGroupCombinatoricFunctions is a package constructor
--R Abbreviation for SymmetricGroupCombinatoricFunctions is SGCF
---R This constructor is exposed in this frame.
---R Issue )edit bookvol10.4.pamphlet to see algebra source code for SGCF
---R
---R----------------------------------- Operations -----------------------------------
---R coleman : (List(Integer),List(Integer),List(Integer)) -> Matrix(Integer)
---R inverseColeman : (List(Integer),List(Integer),Matrix(Integer)) -> List(Integer)
---R listYoungTableaus : List(Integer) -> List(Matrix(Integer))
---R makeYoungTableau : (List(Integer),List(Integer)) -> Matrix(Integer)
---R nextColeman : (List(Integer),List(Integer),Matrix(Integer)) -> Matrix(Integer)
---R nextLatticePermutation : (List(Integer),List(Integer),Boolean) -> List(Integer)
---R nextPartition : (Vector(Integer),Vector(Integer),Integer) -> List(Integer)
---R numberOfImproperPartitions : (Integer,Integer) -> Integer
---R subSet : (Integer,Integer,Integer) -> List(Integer)
---R unrankImproperPartitions0 : (Integer,Integer,Integer) -> List(Integer)
---R unrankImproperPartitions1 : (Integer,Integer,Integer) -> List(Integer)
---E 1

)spool
)lisp (bye)

---------

--- SymmetricGroupCombinatoricFunctions.help ---

====================================================================
SymmetricGroupCombinatoricFunctions examples
====================================================================

SymmetricGroupCombinatoricFunctions contains combinatoric functions
concerning symmetric groups and representation theory: list young
tableaus, improper partitions, subsets bijection of Coleman.

See Also:
  o )show SymmetricGroupCombinatoricFunctions

---------
SymmetricGroupCombinatoricFunctions (SGCF)

Exports:
coleman  inverseColeman
listYoungTableaus  makeYoungTableau
nextColeman  nextLatticePermutation
nextPartition  numberOfImproperPartitions
subSet  unrankImproperPartitions0
unrankImproperPartitions1

— package SGCF SymmetricGroupCombinatoricFunctions —

)abbrev package SGCF SymmetricGroupCombinatoricFunctions
++ Authors: Johannes Grabmeier, Thorsten Werther
++ Date Created: 03 September 1988
++ Date Last Updated: 07 June 1990
++ References:
++ G. James/ A. Kerber: The Representation Theory of the Symmetric
++ S.G. Williamson: Combinatorics for Computer Science,
++ H. Gollan, J. Grabmeier: Algorithms in Representation Theory and
++ their Realization in the Computer Algebra System Scratchpad,
++ Description:
++ SymmetricGroupCombinatoricFunctions contains combinatoric
++ functions concerning symmetric groups and representation
++ theory: list young tableaus, improper partitions, subsets
++ bijection of Coleman.

SymmetricGroupCombinatoricFunctions(): public == private where

NNI => NonNegativeInteger
I => Integer
L ==> List
M ==> Matrix
V ==> Vector
B ==> Boolean
ICF ==> IntegerCombinatoricFunctions Integer

public ==> with

-- IS THERE A WORKING DOMAIN Tableau ?
-- coerce : M I -> Tableau(I)
-- ++ coerce(ytab) coerces the Young-Tableau ytab to an element of
-- ++ the domain Tableau(I).

coleman : (L I, L I, L I) -> M I
++ coleman(alpha,beta,pi):
++ there is a bijection from the set of matrices having nonnegative
++ entries and row sums alpha, column sums beta
++ to the set of Salpha - Sbeta double cosets of the
++ symmetric group Sn. (Salpha is the Young subgroup
++ corresponding to the improper partition alpha).
++ For a representing element pi of such a double coset,
++ coleman(alpha,beta,pi) generates the Coleman-matrix
++ corresponding to alpha, beta, pi.
++ Note that The permutation pi of \{1,2,\ldots,n\} has to be given
++ in list form.
++ Note that the inverse of this map is inverseColeman
++ (if pi is the lexicographical smallest permutation
++ in the coset). For details see James/Kerber.

inverseColeman : (L I, L I, M I) -> L I
++ inverseColeman(alpha,beta,C):
++ there is a bijection from the set of matrices having nonnegative
++ entries and row sums alpha, column sums beta
++ to the set of Salpha - Sbeta double cosets of the
++ symmetric group Sn. (Salpha is the Young subgroup
++ corresponding to the improper partition alpha).
++ For such a matrix C, inverseColeman(alpha,beta,C)
++ calculates the lexicographical smallest pi in the
++ corresponding double coset.
++ Note that the resulting permutation pi of \{1,2,\ldots,n\}
++ is given in list form.
++ Notes: the inverse of this map is coleman.
++ For details, see James/Kerber.

listYoungTableaus : (L I) -> L M I
++ listYoungTableaus(lambda) where lambda is a proper partition
++ generates the list of all standard tableaux of shape lambda
++ by means of lattice permutations. The numbers of the lattice
++ permutation are interpreted as column labels. Hence the
++ contents of these lattice permutations are the conjugate of
++ lambda.
++ Notes: the functions nextLatticePermutation and
++ makeYoungTableau are used.
++ The entries are from 0,...,n-1.
makeYoungTableau : (L I,L I) -> M I
  ++ makeYoungTableau(lambda,gitter) computes for a given lattice
  ++ permutation gitter and for an improper partition lambda
  ++ the corresponding standard tableau of shape lambda.
  ++ Notes: see listYoungTableaus.
  ++ The entries are from 0,...,n-1.
nextColeman : (L I, L I, M I) -> M I
  ++ nextColeman(alpha,beta,C) generates the next Coleman matrix
  ++ of column sums alpha and row sums beta according
  ++ to the lexicographical order from bottom-to-top.
  ++ The first Coleman matrix is achieved by C=new(1,1,0).
  ++ Also, new(1,1,0) indicates that C is the last Coleman matrix.
nextLatticePermutation : (L I, L I, B) -> L I
  ++ nextLatticePermutation(lambda,lattP,constructNotFirst) generates
  ++ the lattice permutation according to the proper partition
  ++ lambda succeeding the lattice permutation lattP in
  ++ lexicographical order as long as constructNotFirst is true.
  ++ If constructNotFirst is false, the first lattice permutation
  ++ is returned.
  ++ The result nil indicates that lattP has no successor.
nextPartition : (V I, V I, I) -> V I
  ++ nextPartition(gamma,part,number) generates the partition of
  ++ number which follows part according to the right-to-left
  ++ lexicographical order. The partition has the property that
  ++ its components do not exceed the corresponding components of
  ++ gamma. The first partition is achieved by part=[].
  ++ Also, [] indicates that part is the last partition.
nextPartition : (L I, V I, I) -> V I
  ++ nextPartition(gamma,part,number) generates the partition of
  ++ number which follows part according to the right-to-left
  ++ lexicographical order. The partition has the property that
  ++ its components do not exceed the corresponding components of
  ++ gamma. The first partition is achieved by part=[].
  ++ Also, [] indicates that part is the last partition.
numberOfImproperPartitions: (I,I) -> I
  ++ numberOfImproperPartitions(n,m) computes the number of partitions
  ++ of the nonnegative integer n in m nonnegative parts with regarding
  ++ the order (improper partitions).
  ++ Example: numberOfImproperPartitions (3,3) is 10,
  ++ since [0,0,3], [0,1,2], [0,2,1], [0,3,0], [1,0,2], [1,1,1],
  ++ [1,2,0], [2,0,1], [2,1,0], [3,0,0] are the possibilities.
  ++ Note that this operation has a recursive implementation.
subSet : (I,I,I) -> L I
  ++ subSet(n,m,k) calculates the k-th m-subset of the set
  ++ 0,1,...,(n-1) in the lexicographic order considered as
  ++ a decreasing map from 0,...,(m-1) into 0,...,(n-1).
  ++ See S.G. Williamson: Theorem 1.60.
  ++ Error: if not (0 <= m <= n and 0 <= k < (n choose m)).
unrankImproperPartitions0 : (I,I,I) -> L I
++ unrankImproperPartitions0(n,m,k) computes the k-th improper
++ partition of nonnegative n in m nonnegative parts in reverse
++ lexicographical order.
++ Example: [0,0,3] < [0,1,2] < [0,2,1] < [0,3,0] <
++ [1,0,2] < [1,1,1] < [1,2,0] < [2,0,1] < [2,1,0] < [3,0,0].
++ Error: if k is negative or too big.
++ Note that counting of subtrees is done by
++ numberOfImproperPartitions
unrankImproperPartitions1: (I,I,I) -> L I
++ unrankImproperPartitions1(n,m,k) computes the k-th improper
++ partition of nonnegative n in at most m nonnegative parts
++ ordered as follows: first, in reverse
++ lexicographically according to their non-zero parts, then
++ according to their positions (i.e. lexicographical order
++ using subSet: [3,0,0] < [0,3,0] < [0,0,3] < [2,1,0] <
++ [2,0,1] < [0,2,1] < [1,2,0] < [1,0,2] < [0,1,2] < [1,1,1].
++ Note that counting of subtrees is done by
++ numberOfImproperPartitionsInternal.

private == add

import Set I

-- declaration of local functions

numberOfImproperPartitionsInternal: (I,I,I) -> I
-- this is used as subtree counting function in
-- "unrankImproperPartitions1". For (n,m,cm) it counts
-- the following set of m-tuples: The first (from left
-- to right) m-cm non-zero entries are equal, the remaining
-- positions sum up to n. Example: (3,3,2) counts
-- [x,3,0], [x,0,3], [0,x,3], [x,2,1], [x,1,2], x non-zero.

-- definition of local functions

numberOfImproperPartitionsInternal(n,m,cm) ==
  n = 0 => binomial(m,cm)$ICF
  cm = 0 and n > 0 => 0
  s := 0
  for i in 0..n-1 repeat
    s := s + numberOfImproperPartitionsInternal(i,m,cm-1)
  s

-- definition of exported functions
numberOfImproperPartitions(n,m) ==
  if n < 0 or m < 1 then return 0
  if m = 1 or n = 0 then return 1
  s := 0
  for i in 0..n repeat
    s := s + numberOfImproperPartitions(n-i,m-1)
  s

unrankImproperPartitions0(n,m,k) ==
  l : L I := nil$(L I)
  k < 0 => error"counting of partitions is started at 0"
  k >= numberOfImproperPartitions(n,m) =>
    error"there are not so many partitions"
  for t in 0..(m-2) repeat
    s : I := 0
    for y in 0..n repeat
      sOld := s
      s := s + numberOfImproperPartitions(n-y,m-t-1)
      if s > k then leave
      l := append(l,list(y)$(L I))$(L I)
    k := k - sOld
    n := n - y
  l

unrankImproperPartitions1(n,m,k) ==
  -- we use the counting procedure of the leaves in a tree
  -- having the following structure: First of all non-zero
  -- labels for the sons. If addition along a path gives n,
  -- then we go on creating the subtree for (n choose cm)
  -- where cm is the length of the path. These subsets determine
  -- the positions for the non-zero labels for the partition
  -- to be formed. The remaining positions are filled by zeros.
  nonZeros : L I := nil$(L I)
  partition : V I := new(m::NNI,0$I)$(V I)
  k < 0 => nonZeros
  k >= numberOfImproperPartitions(n,m) => nonZeros
  cm : I := m
  --cm gives the depth of the tree
  while n ^= 0 repeat
    s : I := 0
    cm := cm - 1
    for y in n..1 by -1 repeat
      --determination of the next son
      sOld := s
      -- remember old s
      -- this functions counts the number of elements in a subtree
      s := s + numberOfImproperPartitionsInternal(n-y,m,cm)
      if s > k then leave
      -- y is the next son, so put it into the pathlist "nonZero"
      nonZeros := append(nonZeros,list(y)$(L I))$(L I)
k := k - sOld -- updating
n := n - y -- updating
-- having found all m-cm non-zero entries we change the structure
-- of the tree and determine the non-zero positions
nonZeroPos : L I := reverse subSet(m, m-cm, k)
-- building the partition
for i in 1..m-cm repeat partition.(1+nonZeroPos.i) := nonZeros.i
entries partition

subSet(n, m, k) ==
k < 0 or n < 0 or m < 0 or m > n =>
    error "improper argument to subSet"
bin : I := binomial$ICF (n, m)
k >= bin =>
    error "there are not so many subsets"
l : L I := []
n = 0 => l
mm : I := k
s : I := m
for t in 0..(m-1) repeat
    for y in (s-1)..(n+1) repeat
        if binomial$ICF (y, s) > mm then leave
    l := append (l, list(y-1)$(L I))
    mm := mm - binomial$ICF (y-1, s)
    s := s-1
l

nextLatticePermutation(lambda, lattP, constructNotFirst) ==
lprime : L I := conjugate(lambda)$PartitionsAndPermutations
columns : NNI := (first(lambda)$(L I))::NNI
rows : NNI := (first(lprime)$(L I))::NNI
n : NNI := (+/lambda)::NNI
not constructNotFirst => -- first lattice permutation
    lattP := null$(L I)
    for i in columns..1 by -1 repeat
        for l in 1..lprime(i) repeat
            lattP := cons(i, lattP)
lattP
help : V I := new(columns, 0) -- entry help(i) stores the number
-- of occurrences of number i on our way from right to left
rightPosition : NNI := n
leftEntry : NNI := lattP(rightPosition)::NNI
ready : B := false
until (ready or (not constructNotFirst)) repeat
    rightEntry : NNI := leftEntry
leftEntry := lattP(rightPosition-1)::NNI
help(rightEntry) := help(rightEntry) + 1
-- search backward decreasing neighbour elements
if rightEntry > leftEntry then
  if ((lprime(leftEntry)-help(leftEntry)) >
      (lprime(rightEntry)-help(rightEntry)+1)) then
    -- the elements may be swapped because the number of occurences
    -- of leftEntry would still be greater than those of rightEntry
    ready := true
  j : NNI := leftEntry + 1
  -- search among the numbers leftEntry+1..rightEntry for the
  -- smallest one which can take the place of leftEntry.
  -- negation of condition above:
  while (help(j)=0) or ((lprime(leftEntry)-lprime(j))
    < (help(leftEntry)-help(j)+2)) repeat j := j + 1
  lattP(rightPosition-1) := j
  help(j) := help(j)-1
  help(leftEntry) := help(leftEntry) + 1
-- reconstruct the rest of the list in increasing order
for l in rightPosition..n repeat
  j := 0
  while help(1+j) = 0 repeat j := j + 1
  lattP(1::NNI) := j+1
  help(1+j) := help(1+j) - 1
-- end of "if rightEntry > leftEntry"
rightPosition := (rightPosition-1)::NNI
if rightPosition = 1 then constructNotFirst := false
-- end of repeat-loop
not constructNotFirst => nil$(L I)
lattP

makeYoungTableau(lambda,gitter) ==
lprime : L I := conjugate(lambda)$PartitionsAndPermutations
columns : NNI := (first(lambda)$(L I))::NNI
rows : NNI := (first(lprime)$L I))::NNI
ytab : M I := new(rows,columns,0)
help : V I := new(columns,1)
i : I := -1 -- this makes the entries ranging from 0,...,n-1
  -- i := 0 would make it from 1,...,n.
j : I := 0
for l in 1..maxIndex gitter repeat
  j := gitter(l)
i := i + 1
  ytab(help(j),j) := i
  help(j) := help(j) + 1
ytab

-- coerce(ytab) ==
listYoungTableaus(lambda) ==
lattice : L I
ytab : M I
younglist : L M I := nil$(L M I)
lattice := nextLatticePermutation(lambda,lattice,false)
until null lattice repeat
  ytab := makeYoungTableau(lambda,lattice)
younglist := append(younglist,[ytab]$(L M I))$(L M I)
lattice := nextLatticePermutation(lambda,lattice,true)
younglist

nextColeman(alpha,beta,C) ==
nrow : NNI := #beta
ncol : NNI := #alpha
vnull : V I := vector(nil()$(L I)) -- empty vector
vzero : V I := new(ncol,0)
vrest : V I := new(ncol,0)
cnull : M I := new(1,1,0)
coleman := copy C
if coleman ^= cnull then
  -- look for the first row of "coleman" that has a succeeding
  -- partition, this can be atmost row nrow-1
  i : NNI := (nrow-1)::NNI
  vrest := row(coleman,i) + row(coleman,nrow)
  -- for k in 1..ncol repeat
  --  vrest(k) := coleman(i,k) + coleman(nrow,k)
  succ := nextPartition(vrest,row(coleman, i),beta(i))
  while (succ = vnull) repeat
    if i = 1 then return cnull -- part is last partition
    i := (i - 1)::NNI
    -- for k in 1..ncol repeat
    --  vrest(k) := vrest(k) + coleman(i,k)
    vrest := vrest + row(coleman,i)
    succ := nextPartition(vrest, row(coleman, i), beta(i))
  j : I := i
  coleman := setRow_!(coleman, i, succ)
  -- for k in 1..ncol repeat
  --  vrest(k) := vrest(k) - coleman(i,k)
  vrest := vrest - row(coleman,i)
else
  vrest := vector alpha
  -- for k in 1..ncol repeat
  -- vrest(k) := alpha(k)
  coleman := new(nrow,ncol,0)
  j := 0
for i in (j+1):NNI..nrow-1 repeat
  succ := nextPartition(vrest,vnull,beta(i))
  coleman := setRow!(coleman, i, succ)
  vrest := vrest - succ
  -- for k in 1..ncol repeat
  -- vrest(k) := vrest(k) - succ(k)
setRow!(coleman, nrow, vrest)

nextPartition(gamma:V I, part:V I, number:I) ==
  nextPartition(entries gamma, part, number)

nextPartition(gamma:L I,part:V I,number:I) ==
  n := #gamma
  vnull : V I := vector(nil()$(L I)) -- empty vector
if part ^= vnull then
  i := 2
  sum := part(1)
  while (part(i) = gamma(i)) or (sum = 0) repeat
    sum := sum + part(i)
    i := i + 1
  if i = 1+n then return vnull -- part is last partition
  sum := sum - 1
  part(i) := part(i) + 1
else
  sum := number
  part := new(n,0)
  i := 1+n
  j := 1
  while sum > gamma(j) repeat
    part(j) := gamma(j)
    sum := sum - gamma(j)
    j := j + 1
  part(j) := sum
  for k in j+1..i-1 repeat
    part(k) := 0
  part

inverseColeman(alpha,beta,C) ==
  pi := nil$(L I)
  nrow := #beta
  ncol := #alpha
help : V I := new(nrow,0)
sum : I := 1
for i in 1..nrow repeat
    help(i) := sum
    sum := sum + beta(i)
for j in 1..ncol repeat
    for i in 1..nrow repeat
        for k in 2..1+C(i,j) repeat
            pi := append(pi,list(help(i))$(L I))
            help(i) := help(i) + 1
    pi

tmp
coleman(alpha,beta,pi) ==
nrow : NNI := #beta
ncol : NNI := #alpha
temp : L L I := nil$(L L I)
help : L I := nil$(L I)
comatrix : M I := new(nrow,ncol,0)
betasum : NNI := 0
alphasum : NNI := 0
for i in 1..ncol repeat
    help := nil$(L I)
    for j in alpha(i)..1 by-1 repeat
        help := cons(pi(j::NNI+alphasum),help)
        alphasum := (alphasum + alpha(i))::NNI
    temp := append(temp,list(help)$(L L I))
for i in 1..nrow repeat
    help := nil$(L I)
    for j in beta(i)..1 by-1 repeat
        help := cons(j::NNI+betasum, help)
        betasum := (betasum + beta(i))::NNI
    for j in 1..ncol repeat
        comatrix(i,j) := #intersect(brace(help),brace(temp(j)))
comatrix

——

— SGCF.dotabb —

"SGCF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SGCF"]
"FSAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FSAGG"]
"SGCF" -> "FSAGG"

——
package SYMFUNC SymmetricFunctions

--- SymmetricFunctions.input ---

)set break resume
)sys rm -f SymmetricFunctions.output
)spool SymmetricFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show SymmetricFunctions
--R
--R SymmetricFunctions(R: Ring) is a package constructor
--R Abbreviation for SymmetricFunctions is SYMFUNC
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SYMFUNC
--R
--R-------------------------------- Operations --------------------------------
--R symFunc : List(R) -> Vector(R)
--R symFunc : (R,PositiveInteger) -> Vector(R)
--R
--E 1

)spool
)lisp (bye)

---

--- SymmetricFunctions.help ---

================================================================================
SymmetricFunctions examples
================================================================================

Computes all the symmetric functions in n variables.

See Also:
o )show SymmetricFunctions

---
SymmetricFunctions (SYMFUNC)

Exports:
symFunc

— package SYMFUNC SymmetricFunctions —

)abbrev package SYMFUNC SymmetricFunctions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 28 Jun 1990
++ Description:
++ Computes all the symmetric functions in n variables.

SymmetricFunctions(R:Ring): Exports == Implementation where
UP ==> SparseUnivariatePolynomial R

Exports ==> with
symFunc: List R -> Vector R
++ symFunc([r1,...,rn]) returns the vector of the
++ elementary symmetric functions in the \spad{ri's}:
++ \spad{[r1 + ... + rn, r1 r2 + ... + r(n-1) rn, ..., r1 r2 ... rn]}.
symFunc: (R, PositiveInteger) -> Vector R
++ symFunc(r, n) returns the vector of the elementary
++ symmetric functions in \spad{[r,r,...,r]} \spad{n} times.

Implementation ==> add
signFix: (UP, NonNegativeInteger) -> Vector R

symFunc(x, n) == signFix((monomial(1, 1)$UP - x::UP) ** n, i + n)
symFunc 1 ==
  signFix(*/[monomial(1, 1)$UP - a::UP for a in 1], i + #1)

signFix(p, n) ==
m := minIndex(v := vectorise(p, n)) + 1
for i in 0..(#v quo 2) - 1::NonNegativeInteger repeat
\begin{verbatim}
qsetelt_!(v, 2*i + m, - qelt(v, 2*i + m))
reverse_! v
\end{verbatim}

— SYMFUNC.dotabb —

"SYMFUNC" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SYMFUNC"]
"IVECTOR" [color="#88FF44",href="bookvol10.3.pdf#nameddest=IVECTOR"]
"SYMFUNC" -> "IVECTOR"
Chapter 21

Chapter T

package TABLBUMP TableauxBumpers

— TableauxBumpers.input —

)set break resume
)sys rm -f TableauxBumpers.output
)spool TableauxBumpers.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TableauxBumpers
--R
--R TableauxBumpers(S: OrderedSet) is a package constructor
--R Abbreviation for TableauxBumpers is TABLBUMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TABLBUMP
--R
--R------------------------------- Operations --------------------------------
--R inverse : List(S) -> List(S)  lex : List(List(S)) -> List(List(S))
--R slex : List(S) -> List(List(S))  tab : List(S) -> Tableau(List(S))
--R bat : Tableau(List(S)) -> List(List(S))
--R bat1 : List(List(List(S))) -> List(List(S))
--R bumprow : (((S,S) -> Boolean),List(S),List(List(S))) -> Record(fs: Boolean,sd: List(S),td: List(List(S)))
--R bumptab : (((S,S) -> Boolean),List(S),List(List(List(S)))) -> List(List(List(S)))
--R bumptab1 : (List(S),List(List(List(S)))) -> List(List(List(S)))
--R maxrow : (List(S),List(List(List(S))),List(List(S)),List(List(List(S))),List(List(List(S))),List(List(List(S)))) -> Record(f1: List(S),f2: List(List(List(S))),f3: List(List(S)),f4: List(List(List(S))))
--R mr : List(List(List(S))) -> Record(f1: List(S),f2: List(List(List(S))),f3: List(List(S)),f4: List(List(List(S))))
--R tab1 : List(List(S)) -> List(List(List(S)))
--R untab : (List(List(S)),List(List(List(S)))) -> List(List(S))

3941
TableauxBumpers implements the Schenstead-Knuth correspondence between sequences and pairs of Young tableaux. The 2 Young tableaux are represented as a single tableau with pairs as components.

See Also:
o )show TableauxBumpers

Exports:
bat bat1 bumprow bumtab bumtab1
inverse lex maxrow mr slex
tab tab1 untab

— package TABLBUMP TableauxBumpers —

)abbrev package TABLBUMP TableauxBumpers
++ Author: William H. Burge
++ Date Created: 1987
++ Date Last Updated: 23 Sept 1991
++ Description:
++ TableauBumpers implements the Schenstead-Knuth correspondence between sequences and pairs of Young tableaux.
++ The 2 Young tableaux are represented as a single tableau with ++ pairs as components.

TableauxBumpers(S:OrderedSet):T==C where
L==>List
ST==>Stream
B==>Boolean
ROW==>Record(fs:B,sd:L S,td:L L S)
RC==>Record(f1:L S,f2:L L L S,f3:L L S,f4:L L L S)
PAIR==>L S
T== with
bumprow:((S,S)->B,PAIR,L PAIR)->ROW
++ bumprow(cf,pr,r) is an auxiliary function which
++ bumps a row r with a pair pr
++ using comparison function cf, and returns a record
bumptab:((S,S)->B,PAIR,L L PAIR)->L L PAIR
++ bumptab(cf,pr,t) bumps a tableau t with a pair pr
++ using comparison function cf, returning a new tableau
bumptab1:(PAIR,L L PAIR)->L L PAIR
++ bumptab1(pr,t) bumps a tableau t with a pair pr
++ using comparison function \spadfun{<},
++ returning a new tableau
untab: (L PAIR,L L PAIR)->L PAIR
++ untab(lp,llp) is an auxiliary function
++ which unbumps a tableau llp,
++ using lp to accumulate pairs
bat1:L L PAIR->L PAIR
++ bat1(llp) unbumps a tableau llp.
++ Operation bat1 is the inverse of tab1.
bat:Tableau(L S)->L L S
++ bat(ls) unbumps a tableau ls
tab1:L PAIR->L L PAIR
++ tab1(lp) creates a tableau from a list of pairs lp
tab:L S->Tableau(L S)
++ tab(ls) creates a tableau from ls by first creating
++ a list of pairs using slex,
++ then creating a tableau using tab1.
lex:L PAIR->L PAIR
++ lex(ls) sorts a list of pairs to lexicographic order
slex:L S->L PAIR
++ slex(ls) sorts the argument sequence ls, then zips (see map) the
++ original argument sequence with the sorted result to
++ a list of pairs
inverse:L S->L S
++ inverse(ls) forms the inverse of a sequence ls
maxrow:(PAIR,L L PAIR,L PAIR,L L PAIR,L L PAIR,L L PAIR)->RC
++ maxrow(a,b,c,d,e) is an auxiliary function for mr
mr:L L PAIR->RC
++ mr(t) is an auxiliary function which
++ finds the position of the maximum element of a tableau t
++ which is in the lowest row, producing a record of results
C== add
cf:(S,S)->B
bumprow(cf,x:(PAIR),lls:(L PAIR))==
  if null lls
  then [false,x,[x]]$ROW
  else (y:(PAIR):=first lls;
     if cf(x.2,y.2)
     then [true,[x.1,y.2],cons([y.1,x.2],rest lls)]$ROW
     else (rw:ROW:=bumprow(cf,x,rest lls);
         [rw.fs,rw.sd,cons(first lls,rw.td)]$ROW ))
bumptab(cf,x:(PAIR),lllls:(L L PAIR))==
  if null lllls
  then [[x]]
  else (rw:ROW:=bumprow(cf,x,first lllls);
     if rw.fs
     then cons(rw.td, bumptab(cf,rw.sd,rest lllls))
     else cons(rw.td,rest lllls))
bumptab1(x,lllls)==bumptab((s1,s2) +-> s1<s2, x, lllls)
rd==> reduce$StreamFunctions2(PAIR,L L PAIR)
tab1(llls:(L PAIR))== rd([],bumptab1,lllls::(ST PAIR))
srt==>sort$(PAIR)
lexorder:(PAIR,PAIR)->B
lexorder(p1,p2)==if p1.1=p2.1 then p1.2<p2.2 else p1.1<p2.1
lex lp== (sort$(L PAIR))((s1,s2) +-> lexorder(s1,s2), lp)
slex ls==lex([[[i,j] for i in srt((s1, s2) +-> s1<s2, ls) for j in ls]])
inverse ls==[ls.2 for lss in
   lex([[j,i] for i in srt((s1,s2) +-> s1<s2, ls) for j in ls]])
tab(ls:(PAIR))==(tableau tab1 slex ls )
maxrow(n,a,b,c,d,lllls)==
  if null lllls or null(first lllls)
  then [n,a,b,c]$RC
  else (fst:=first first lllls;rst:=rest first lllls;
  if fst.1>n.1
  then maxrow(fst,d,rst,rest lllls,cons(first lllls,d),rest lllls)
  else maxrow(n,a,b,c,cons(first lllls,d),rest lllls))
mr llls==maxrow(first first llls,[],rest first llls,rest llls,
     [],llls)
untab(lp, llls)==
  if null llls
  then lp
  else (rc:RC:=mr llls;
     rv:=reverse (bumptab((s1:S,s2:S):B +-> s2<s1, rc. f1, rc. f2));
     untab(cons(first first rv,lp)
   ,append(rest rv,
         if null rc.f3
         then []
         else cons(rc.f3,rc.f4))))
bat1 llls==untab([],[reverse lls for lls in llls])
bat tb==bat1(listOfLists tb)

— TABLBUMP.dotabb —
"TABLBUMP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TABLEBUMP"]
"FLAGG" [color="#44B3FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"FLAGG-" [color="#88FF44",href="bookvol10.3.pdf#nameddest=FLAGG"]
"TABLEBUMP" -> "FLAGG-"
"TABLEBUMP" -> "FLAGG"

package TBCMPPK TabulatedComputationPackage

— TabulatedComputationPackage.input —

)set break resume
)sys rm -f TabulatedComputationPackage.output
)spool TabulatedComputationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TabulatedComputationPackage
--R
--R TabulatedComputationPackage(Key: SetCategory,Entry: SetCategory) is a package constructor
--R Abbreviation for TabulatedComputationPackage is TBCMPPK
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TBCMPPK
--R
--R---------------------------------------------------------- Operations ---------------------
--R clearTable!: () -> Void                       initTable!: () -> Void
--R insert!: (Key,Entry) -> Void                   makingStats?: () -> Boolean
--R printInfo!: (String,String) -> Void            printStats!: () -> Void
--R printingInfo?: () -> Boolean                   startStats!: String -> Void
--R usingTable?: () -> Boolean                     usingTable?: () -> Boolean
--R extractIfCan : Key -> Union(Entry,"failed")    --R
--R
)spool
)lisp (bye)

-----

— TabulatedComputationPackage.help —

====================================================================
TabulatedComputationPackage examples
====================================================================

TabulatedComputationPackage(Key,Entry) provides some modest support for dealing with operations with type Key -> Entry. The result of such operations can be stored and retrieved with this package by using a hash-table. The user does not need to worry about the management of this hash-table. However, only one hash-table is built by calling TabulatedComputationPackage(Key,Entry).

See Also:
  o )show TabulatedComputationPackage

-----
TabulatedComputationPackage (TBCMPPK)

Exports:
clearTable! extractIfCan initTable! insert! makingStats?
printInfo! printStats! printingInfo? startStats! usingTable?

— package TBCMPPK TabulatedComputationPackage —

)abbrev package TBCMPPK TabulatedComputationPackage
++ Author: Marc Moreno Maza
++ Date Created: 09/09/1998
++ Date Last Updated: 12/16/1998
++ Description:
++ \texttt{TabulatedComputationPackage(Key ,Entry)} provides some modest support
++ for dealing with operations with type \texttt{Key -\rightarrow Entry}. The result of
++ such operations can be stored and retrieved with this package by using
++ a hash-table. The user does not need to worry about the management of
++ this hash-table. However, only one hash-table is built by calling
++ \texttt{TabulatedComputationPackage(Key ,Entry)}.

TabulatedComputationPackage(Key ,Entry): Exports == Implementation where
  Key: SetCategory
  Entry: SetCategory
  N ==> NonNegativeInteger
  H ==> HashTable(Key, Entry, "UEQUAL")
  iprintpack ==> InternalPrintPackage()

Exports == with
  initTable!: () -> Void
  ++ \texttt{initTable!()} initializes the hash-table.
  printInfo!: (String, String) -> Void
  ++ \texttt{printInfo!(x,y)} initializes the messages to be printed
  ++ when manipulating items from the hash-table. If
  ++ a key is retrieved then \texttt{axiom(x)} is displayed. If an item is
  ++ stored then \texttt{axiom(y)} is displayed.
  startStats!: (String) -> Void
  ++ \texttt{startStats!(x)} initializes the statistics process and
++ sets the comments to display when statistics are printed
printStats!: () -> Void
++ \texttt{printStats!()} prints the statistics.
clearTable!: () -> Void
++ \texttt{clearTable!()} clears the hash-table and assumes that
++ it will no longer be used.
usingTable?: () -> Boolean
++ \texttt{usingTable?()} returns true iff the hash-table is used
printingInfo?: () -> Boolean
++ \texttt{printingInfo?()} returns true iff messages are printed
++ when manipulating items from the hash-table.
makingStats?: () -> Boolean
++ \texttt{makingStats?()} returns true iff the statistics process
++ is running.
extractIfCan: Key -> Union(Entry,"failed")
++ \texttt{extractIfCan(x)} searches the item whose key is \texttt{x}.
insert!: (Key, Entry) -> Void
++ \texttt{insert!(x,y)} stores the item whose key is \texttt{x} and whose
++ entry is \texttt{y}.

\textbf{Implementation == add}

\texttt{table?: Boolean := false}
\texttt{t: H := empty()}
\texttt{info?: Boolean := false}
\texttt{stats?: Boolean := false}
\texttt{used: NonNegativeInteger := 0}
\texttt{ok: String := "o"}
\texttt{ko: String := "+"}
\texttt{domainName: String := empty()}$String

\texttt{initTable!(): Void ==}
\texttt{table? := true}
\texttt{t := empty()}
\texttt{void()}

\texttt{printInfo!(s1: String, s2: String): Void ==}
\texttt{(empty? s1) or (empty? s2) => void()}
\texttt{not usingTable? =>
  error "in printInfo!()$TBCMPPK: not allowed to use hashtable"
info? := true
ok := s1
ko := s2
void()}

\texttt{startStats!(s: String): Void ==}
\texttt{empty? s => void()}
\texttt{not table? =>
  error "in startStats!()$TBCMPPK: not allowed to use hashtable"
stats? := true
used := 0
domainName := s
void()}

printStats!(): Void ==
  not table? =>
    error "in printStats!()$TBCMPPK: not allowed to use hashtable"
  not stats? =>
    error "in printStats!()$TBCMPPK: statistics not started"
  output(" ")$OutputPackage
  title: String := concat("*** ", concat(domainName," Statistics ***"))
  output(title)$OutputPackage
  n: N := #t
  output(" Table size: ", n::OutputForm)$OutputPackage
  output(" Entries reused: ", used::OutputForm)$OutputPackage

clearTable!(): Void ==
  not table? =>
    error "in clearTable!()$TBCMPPK: not allowed to use hashtable"
  t := empty()
  table? := false
  info? := false
  stats? := false
  domainName := empty()$String
  void()

usingTable?() == table?
printingInfo?() == info?
makingStats?() == stats?

extractIfCan(k: Key): Union(Entry,"failed") ==
  not table? => "failed" :: Union(Entry,"failed")
  s: Union(Entry,"failed") := search(k,t)
  s case Entry =>
    if info? then iprint(ok)$iprintpack
    if stats? then used := used + 1
    return s
  "failed" :: Union(Entry,"failed")

insert!(k: Key, e:Entry): Void ==
  not table? => void()
  t.k := e
  if info? then iprint(ko)$iprintpack
  void()
package TANEXP TangentExpansions

— TangentExpansions.input —

)set break resume
)sys rm -f TangentExpansions.output
)spool TangentExpansions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TangentExpansions
--R
--R TangentExpansions(R: Field) is a package constructor
--R Abbreviation for TangentExpansions is TANEXP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TANEXP
--R
--R--------------------------------------- Operations ---------------------------------------
--R tanNa : (R,Integer) -> R           tanSum : List(R) -> R
--R tanAn : (R,PositiveInteger) -> SparseUnivariatePolynomial(R)
--R
--E 1

)spool
)lisp (bye)

——

— TangentExpansions.help —

====================================================================
TangentExpansions examples
====================================================================

Expands tangents of sums and scalar products.

See Also:
  o )show TangentExpansions

——
TangentExpansions (TANEXP)

Exports:

- tanAn
- tanNa
- tanSum

--- package TANEXP TangentExpansions ---

)abbrev package TANEXP TangentExpansions
++ Author: Manuel Bronstein
++ Date Created: 13 Feb 1989
++ Date Last Updated: 20 Apr 1990
++ Description:
++ Expands tangents of sums and scalar products.

TangentExpansions(R:Field): Exports == Implementation where

PI ==> PositiveInteger
Z  ==> Integer
UP ==> SparseUnivariatePolynomial R
QF ==> Fraction UP

Exports ==> with

- tanSum: List R -> R
  ++ tanSum([a1, ..., an]) returns \( f(a1, ..., an) \) such that
  ++ if \( ai = \tan(ui) \) then \( f(ai, ..., an) = \tan(u1 + ... + un) \).
- tanAn : (R, PI) -> UP
  ++ tanAn(a, n) returns \( P(x) \) such that
  ++ if \( \spad{a = \tan(u)} \) then \( P(\tan(u/n)) = 0 \).
- tanNa : (R, Z) -> R
  ++ tanNa(a, n) returns \( f(a) \) such that
  ++ if \( \spad{a = \tan(u)} \) then \( f(a) = \tan(n * u) \).

Implementation ==> add

- import SymmetricFunctions(R)
- import SymmetricFunctions(UP)

mitoN : Integer -> Integer
tanPIa: PI -> QF
mitoN n == (odd? n => -1; 1)
tanAn(a, n) == a * denom(q := tanPIa n) - numer q

tanNa(a, n) ==
zero? n => 0
negative? n => - tanNa(a, -n)
(numer(t := tanPIa(n::PI)) a) / ((denom t) a)

tanSum l ==
m := minIndex(v := symFunc l)
+/[mitoN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
/ +/[mitoN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]

-- tanPIa(n) returns P(a)/Q(a) such that
-- if a = tan(u) then P(a)/Q(a) = tan(n * u);
tanPIa n ==
m := minIndex(v := symFunc(monomial(1, 1)$UP, n))
+/[mitoN(i+1) * v(2*i - 1 + m) for i in 1..(#v quo 2)]
/ +/[mitoN(i) * v(2*i + m) for i in 0..((#v - 1) quo 2)]

——

— TANEXP.dotabb —
"TANEXP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TANEXP"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"TANEXP" -> "PFECAT"

——

package UTSSOL TaylorSolve

— TaylorSolve.input —

)set break resume
)sys rm -f TaylorSolve.output
)spool TaylorSolve.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TaylorSolve
PACKAGE UTSSOL TAYLORSOLVE

--R
--R TaylorSolve(F: Field,UTSF: UnivariateTaylorSeriesCategory(F),UTSSUPF: UnivariateTaylorSeriesCategory(SparseUnivariatePolynomialExpressions(F))) is a package constructor
--R Abbreviation for TaylorSolve is UTSSOL
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UTSSOL
--R
--R----------------------------------- Operations -----------------------------------
--R seriesSolve : ((UTSSUPF -> UTSSUPF),List(F)) -> UTSF
--R
--E 1

)spool
)lisp (bye)

---

--- TaylorSolve.help ---

====================================================================
TaylorSolve examples
====================================================================

UTSSOL is a facility to compute the first few coefficients of a Taylor series given only implicitly by a function f that vanishes when applied to the Taylor series.

It uses the method of undetermined coefficients.

Could I either
take a function UTSCAT F $\rightarrow$ UTSCAT F and still be able to compute with undetermined coefficients, or
take a function F $\rightarrow$ F, and do likewise?

Let's see.

Try to compute the equation without resorting to power series. I.e., % c: SUP SUP F, and f: SUP SUP F $\rightarrow$ SUP SUP F. Won't this make the computation of coefficients terribly slow?

I could also try to replace transcendental kernels with variables\dots

Unfortunately, SUP F does not have TRANFUN -- well, it can't, of course. However, I'd like to be able to compute % sin(1+monomial(1,1)$UFPS SUP EXPR INT).

See Also:
o )show TaylorSolve

---
TaylorSolve (UTSSOL)

Exports:
seriesSolve

--- package UTSSOL TaylorSolve ---

)abbrev package UTSSOL TaylorSolve
++ Description:
++ This package has no description

TaylorSolve(F, UTSF, UTSSUPF): Exports == Implementation where
  F: Field
  SUP ==> SparseUnivariatePolynomialExpressions
  UTSF: UnivariateTaylorSeriesCategory F
  UTSSUPF: UnivariateTaylorSeriesCategory SUP F
  NNI ==> NonNegativeInteger

Exports == with
  seriesSolve: (UTSSUPF -> UTSSUPF, List F) -> UTSF

Implementation == add
\getchunk{implementation: UTSSOL TaylorSolve}

--- implementation: UTSSOL TaylorSolve ---

seriesSolve(f, l) ==
c1 :=
  map((x:F):SUP F +-> x::(SUP F), l)
$ListFunctions2(F, SUP F)::(Stream SUP F)
coeffs: Stream SUP F := concat(c1, generate(monomial(1$F,1$NNI))))
-- coeffs: Stream SUP F := concat(c1, monomial(1$F,1$NNI))

coeffs is the stream of the already computed coefficients of the solution, plus one which is so far undetermined. We store in st.2 the complete stream and in st.1 the stream starting with the first coefficient that has possibly not yet been computed.

The mathematics is not quite worked out. If coeffs is initialized as stream with all coefficients set to the \emph{same} transcendental value, and not enough initial values are given, then the missing ones are implicitly assumed to be all identical. It may well happen that a solution is produced, although it is not uniquely determined dots

— implementation: UTSSOL TaylorSolve —

st: List Stream SUP F := [coeffs, coeffs]

Consider an arbitrary equation \( f(x, y(x)) = 0 \). When setting \( x = 0 \), we obtain \( f(0, y(0)) = 0 \). It is not necessarily the case that this determines \( y(0) \) uniquely, so we need one initial value that satisfies this equation.

\[ \text{seriesSolve} \text{ should check that the given initial values satisfy } f(0, y(0), y'(0), \ldots) = 0. \]

Now consider the derivatives of \( f \), where we write \( y \) instead of \( y(x) \) for better readability:

\[ \frac{d}{dx} f(x, y) = f_1(x, y) + f_2(x, y) y' \]

and

\[ \frac{d^2}{dx^2} f(x, y) = f_{1,1}(x, y) + f_{1,2}(x, y) y' + f_{2,1}(x, y) y'' + f_{2,2}(x, y) (y')^2 + f_2(x, y) y'''. \]

In general, \( \frac{d^2}{dx^2} f(x, y) \) depends only linearly on \( y'' \).

This points to another possibility: Since we know that we only need to solve linear equations, we could compute two values and then use interpolation. This might be a bit slower, but more importantly: can we still check that we have enough initial values? Furthermore, we then really need that \( f \) is analytic, i.e., operators are not necessarily allowed anymore. However, it seems that composition is allowed.
| implementation: UTSSOL TaylorSolve |

next: () -> F :=

   nr := st.1
   res: F

   if ground?(coeff: SUP F := nr.1)$(SUP F)

| |

If the next element was already calculated, we can simply return it:

| implementation: UTSSOL TaylorSolve |

   then
      res := ground coeff
      st.1 := rest nr
   else

| |

Otherwise, we have to find the first non-satisfied relation and solve it. It should be linear, or a single non-constant monomial. That is, the solution should be unique.

| implementation: UTSSOL TaylorSolve |

   ns := st.2
   eqs: Stream SUP F := coefficients f series ns
   while zero? first eqs repeat eqs := rest eqs
   eq: SUP F := first eqs
   if degree eq > 1 then
      if monomial? eq then res := 0
      else
         output(hconcat("The equation is: ", eq::OutputForm)) $OutputPackage
         error "seriesSolve: equation for coefficient not linear"
      else res := (-coefficient(eq, 0$NNI)$(SUP F)
               /coefficient(eq, 1$NNI)$(SUP F))

   nr.1 := res::SUP F
   --
   concat!(st.2, monomial(1$F,1$NNI))
   st.1 := rest nr

   res

series generate next
package TEMUTL TemplateUtilities

TemplateUtilities is a package constructor
Abbreviation for TemplateUtilities is TEMUTL
This constructor is exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for TEMUTL

Operations
interpretString : String \rightarrow Any
stripCommentsAndBlanks : String \rightarrow String

TemplateUtilities examples
This package provides functions for template manipulation

See Also:
  o )show TemplateUtilities

TemplateUtilities (TEMUTL)

Exports:
  interpretString  stripCommentsAndBlanks

--- package TEMUTL TemplateUtilities ---

)abbrev package TEMUTL TemplateUtilities
++ Author: Mike Dewar
++ Date Created: October 1992
++ Description:
++ This package provides functions for template manipulation

TemplateUtilities(): Exports == Implementation where

Exports == with
  interpretString : String -> Any
    ++ interpretString(s) treats a string as a piece of AXIOM input, by
    ++ parsing and interpreting it.
  stripCommentsAndBlanks : String -> String
    ++ stripCommentsAndBlanks(s) treats s as a piece of AXIOM input, and
    ++ removes comments, and leading and trailing blanks.

Implementation == add
import InputForm

stripC(s:String,u:String):String ==
  i : Integer := position(u,s,1)
  i = 0 => s
  delete(s,i..)

stripCommentsAndBlanks(s:String):String ==
  trim(stripC(stripC(s,"++"),"--"),char " ")

parse(s:String):InputForm ==
  ncParseFromString(s)$Lisp::InputForm

interpretString(s:String):Any ==
  interpret parse s

package TEX1 TexFormat1

| TexFormat1.input |
| set break resume |
| sys rm -f TexFormat1.output |
| spool TexFormat1.output |
| set message test on |
| set message auto off |
| clear all |
|--S 1 of 1 |
| show TexFormat1 |
|--R |
|--R TexFormat1(S: SetCategory) is a package constructor |
|--R Abbreviation for TexFormat1 is TEX1 |
|--R This constructor is exposed in this frame.
TEXFormat1 (TEX1)

Exports:
coerce

— package TEX1 TexFormat1 —
)abbrev package TEX1 TexFormat1
++ Author: Robert S. Sutor
++ Date Created: 1987 through 1990
++ Description:
++ \spadtype{TexFormat1} provides a utility coercion for changing
++ to TeX format anything that has a coercion to the standard output format.

TexFormat1(S : SetCategory): public == private where
    public == with
        coerce: S -> TexFormat()
        ++ coerce(s) provides a direct coercion from a domain S to
        ++ TeX format. This allows the user to skip the step of first
        ++ manually coercing the object to standard output format before
        ++ it is coerced to TeX format.

    private == add
        import TexFormat()

        coerce(s : S): TexFormat ==
            coerce(s :: OutputForm)$TexFormat

    — TEX1.dotabb —

"TEX1" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TEX1"]
"BASTYPE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=BASTYPE"]
"KOERCE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=KOERCE"]
"TEX1" -> "BASTYPE"
"TEX1" -> "KOERCE"

— package TOOLSIGN ToolsForSign —

( set break resume
( sys rm -f ToolsForSign.output
( spool ToolsForSign.output
( set message test on
( set message auto off
( clear all
ToolsForSign (TOOLSIGN)
Exports:
   direction  nonQsign  sign

— package TOOLSIGN ToolsForSign —

)abbrev package TOOLSIGN ToolsForSign
++ Author: Manuel Bronstein
++ Date Created: 25 August 1989
++ Date Last Updated: 26 November 1991
++ Description:
++ Tools for the sign finding utilities.

ToolsForSign(R:Ring): with
   sign : R -> Union(Integer, "failed")
      ++ sign(r) \ undocumented
   nonQsign : R -> Union(Integer, "failed")
      ++ nonQsign(r) \ undocumented
   direction: String -> Integer
      ++ direction(s) \ undocumented
== add

if R is AlgebraicNumber then
   nonQsign r ==
      sign((r pretend AlgebraicNumber)::Expression(Integer))$ElementaryFunctionSign(Integer, Expression Integer)
else
   nonQsign r == "failed"

if R has RetractableTo Fraction Integer then
   sign r ==
      (u := retractIfCan(r)@Union(Fraction Integer, "failed"))
      case Fraction(Integer) => sign(u::Fraction Integer)
      nonQsign r
else
   if R has RetractableTo Integer then
      sign r ==
         (u := retractIfCan(r)@Union(Integer, "failed"))
         case "failed" => "failed"
         sign(u::Integer)
   else
      sign r ==
         zero? r => 0
         -- one? r => 1
         r = 1 => 1
         r = -1 => -1
         "failed"
direction st ==
  st = "right" => 1
  st = "left" => -1
error "Unknown option"

— TOOLSIGN.dotabb —

"TOOLSIGN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TOOLSIGN"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"TOOLSIGN" -> "FS"

package DRAW TopLevelDrawFunctions

— TopLevelDrawFunctions.input —

)set break resume
)sys rm -f TopLevelDrawFunctions.output
)spool TopLevelDrawFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TopLevelDrawFunctions

--R
--R TopLevelDrawFunctions(Ex: Join(ConvertibleTo(InputForm),SetCategory)) is a package constructor
--R Abbreviation for TopLevelDrawFunctions is DRAW
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DRAW

--R----------------------------- Operations -------------------------------
--R draw : (Ex,SegmentBinding(Float),List(DrawOption)) -> TwoDimensionalViewport
--R draw : (Ex,SegmentBinding(Float)) -> TwoDimensionalViewport
--R draw : (ParametricPlaneCurve(Ex),SegmentBinding(Float),List(DrawOption)) -> TwoDimensionalViewport
--R draw : (ParametricPlaneCurve(Ex),SegmentBinding(Float)) -> TwoDimensionalViewport
--R draw : (ParametricSpaceCurve(Ex),SegmentBinding(Float),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : (ParametricSpaceCurve(Ex),SegmentBinding(Float)) -> ThreeDimensionalViewport
--R draw : (Ex,SegmentBinding(Ex),SegmentBinding(Ex),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : (Ex,SegmentBinding(Ex),SegmentBinding(Ex)) -> ThreeDimensionalViewport
--R draw : (ParametricSurface(Ex),SegmentBinding(Float),SegmentBinding(Float),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : (ParametricSurface(Ex),SegmentBinding(Float),SegmentBinding(Float)) -> ThreeDimensionalViewport
TopLevelDrawFunctions provides top level functions for drawing graphics of expressions.

See Also:

- )show TopLevelDrawFunctions

Exports:

- draw
- makeObject
— package DRAW TopLevelDrawFunctions —

)abbrev package DRAW TopLevelDrawFunctions
++ Author: Clifton J. Williamson
++ Date Created: 23 January 1990
++ Date Last Updated: October 1991 by Jon Steinbach
++ Description:
++ TopLevelDrawFunctions provides top level functions for
++ drawing graphics of expressions.

TopLevelDrawFunctions(Ex:Join(ConvertibleTo InputForm,SetCategory)):
Exports == Implementation where
  B    ==> Boolean
  BIND ==> SegmentBinding Float
  L    ==> List
  SF   ==> DoubleFloat
  DROP ==> DrawOption

  PPC  ==> ParametricPlaneCurve Ex
  PPCF ==> ParametricPlaneCurve(SF -> SF)
  PSC  ==> ParametricSpaceCurve Ex
  PSCF ==> ParametricSpaceCurve(SF -> SF)
  PSF  ==> ParametricSurface Ex
  PSFF ==> ParametricSurface((SF,SF) -> SF)
  SPACE3 ==> ThreeSpace(SF)
  VIEW2 ==> TwoDimensionalViewport
  VIEW3 ==> ThreeDimensionalViewport

Exports ==> with

--% Two Dimensional Function Plots

draw: (Ex,BIND,L DROP) -> VIEW2
  ++ draw(f(x),x = a..b,l) draws the graph of \spad{y = f(x)} as x
  ++ ranges from \spad{\min(a,b)} to \spad{\max(a,b)}; \spad{f(x)} is the
  ++ default title, and the options contained in the list l of
  ++ the domain \spad{DrawOption} are applied.
draw: (Ex,BIND) -> VIEW2
  ++ draw(f(x),x = a..b) draws the graph of \spad{y = f(x)} as x
  ++ ranges from \spad{\min(a,b)} to \spad{\max(a,b)}; \spad{f(x)} appears
  ++ in the title bar.

--% Parametric Plane Curves

draw: (PPC,BIND,L DROP) -> VIEW2
  ++ draw(curve(f(t),g(t)),t = a..b,l) draws the graph of the parametric
  ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{\min(a,b)} to
  ++ \spad{\max(a,b)}; \spad{(f(t),g(t))} is the default title, and the
  ++ options contained in the list l of the domain \spad{DrawOption}
++ are applied.

draw: (PPC,BIND) -> VIEW2
++ draw(curve(f(t),g(t)),t = a..b) draws the graph of the parametric
++ curve \spad{x = f(t), y = g(t)} as \spad{t} ranges from \spad{\min(a,b)} to
++ \spad{\max(a,b)}; \spad{(f(t),g(t))} appears in the title bar.

--% Parametric Space Curves

draw: (PSC,BIND,L DROP) -> VIEW3
++ draw(curve(f(t),g(t),h(t)),t = a..b,l) draws the graph of the
++ parametric curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)}
++ as \spad{t} ranges from \spad{\min(a,b)} to \spad{\max(a,b)}; \spad{h(t)}
++ is the default title, and the options contained in the list \spad{l} of
++ the domain \spadtype{DrawOption} are applied.
draw: (PSC,BIND) -> VIEW3
++ draw(curve(f(t),g(t),h(t)),t = a..b) draws the graph of the parametric
++ curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)} as \spad{t} ranges
++ from \spad{\min(a,b)} to \spad{\max(a,b)}; \spad{h(t)} is the default
++ title.
makeObject: (PSC,BIND,L DROP) -> SPACE3
++ makeObject(curve(f(t),g(t),h(t)),t = a..b,l) returns a space of
++ the domain \spadtype{ThreeSpace} which contains the graph of the
++ parametric curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)}
++ as \spad{t} ranges from \spad{\min(a,b)} to \spad{\max(a,b)}; \spad{h(t)}
++ is the default title, and the options contained in the list \spad{l} of
++ the domain \spadtype{DrawOption} are applied.
makeObject: (PSC,BIND) -> SPACE3
++ makeObject(curve(f(t),g(t),h(t)),t = a..b) returns a space of the
++ domain \spadtype{ThreeSpace} which contains the graph of the
++ parametric curve \spad{x = f(t)}, \spad{y = g(t)}, \spad{z = h(t)}
++ as \spad{t} ranges from \spad{\min(a,b)} to \spad{\max(a,b)}; \spad{h(t)} is
++ the default title.

--% Three Dimensional Function Plots

draw: (Ex,BIND,BIND,L DROP) -> VIEW3
++ draw(f(x,y),x = a..b,y = c..d,l) draws the graph of \spad{z = f(x,y)}
++ as \spad{x} ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and \spad{y} ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{f(x,y)} is the default
++ title, and the options contained in the list \spad{l} of the domain
++ \spadtype{DrawOption} are applied.
draw: (Ex,BIND,BIND) -> VIEW3
++ draw(f(x,y),x = a..b,y = c..d) draws the graph of \spad{z = f(x,y)}
++ as \spad{x} ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and \spad{y} ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{f(x,y)} appears in the title bar.
makeObject: (Ex,BIND,BIND,L DROP) -> SPACE3
++ makeObject(f(x,y),x = a..b,y = c..d,l) returns a space of the
++ domain \spadtype{ThreeSpace} which contains the graph of
++ \spad{z = f(x,y)} as \spad{x} ranges from \spad{\min(a,b)} to \spad{\max(a,b)}
++ and \spad{y} ranges from \spad{\min(c,d)} to \spad{\max(c,d)}; \spad{f(x,y)}
++ is the default title, and the options contained in the list 1 of the
++ domain \spad{DrawOption} are applied.
makeObject: (EX,BIND,BIND) -> SPACE3
++ makeObject(f(x,y),x = a..b,y = c..d) returns a space of the domain
++ \spadtype{ThreeSpace} which contains the graph of \spad{z = f(x,y)}
++ as x ranges from \spad{min(a,b)} to \spad{max(a,b)} and y ranges from
++ \spad{min(c,d)} to \spad{max(c,d)}; \spad{f(x,y)} appears as the
++ default title.

--% Parametric Surfaces

draw: (PSF,BIND,BIND,L DROP) -> VIEW3
++ draw(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d,l) draws the
++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
++ the default title, and the options contained in the list 1 of
++ the domain \spad{DrawOption} are applied.
draw: (PSF,BIND,BIND) -> VIEW3
++ draw(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d) draws the
++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
++ the default title.
makeObject: (PSF,BIND,BIND,L DROP) -> SPACE3
++ makeObject(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d,l) returns
++ a space of the domain \spadtype{ThreeSpace} which contains the graph
++ of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
++ the default title, and the options contained in the list 1 of
++ the domain \spad{DrawOption} are applied.
makeObject: (PSF,BIND,BIND) -> SPACE3
++ makeObject(surface(f(u,v),g(u,v),h(u,v)),u = a..b,v = c..d) returns
++ a space of the domain \spadtype{ThreeSpace} which contains the
++ graph of the parametric surface \spad{x = f(u,v)}, \spad{y = g(u,v)},
++ \spad{z = h(u,v)} as u ranges from \spad{min(a,b)} to \spad{max(a,b)}
++ and v ranges from \spad{min(c,d)} to \spad{max(c,d)}; \spad{h(t)} is
++ the default title.

Implementation ==> add
import TopLevelDrawFunctionsForCompiledFunctions
import MakeFloatCompiledFunction(Ex)
import ParametricPlaneCurve(SF -> SF)
import ParametricSpaceCurve(SF -> SF)
import ParametricSurface((SF,SF) -> SF)
import ThreeSpace(SF)
--% Two Dimensional Function Plots

\[
draw(f:Ex,bind:BIND, l:L DROP) ==
-- create title if necessary
if not option?(l,"title" :: Symbol) then
  s:String := unparse(convert(f)@InputForm)
  if sayLength(s)$DisplayPackage > 50 then
    l := concat(title "AXIOM2D", l)
  else l := concat(title s, l)
-- call 'draw'
draw(makeFloatFunction(f, variable bind), segment bind, l)
\]

\[
draw(f:Ex,bind:BIND) == draw(f, bind, nil())
\]

--% Parametric Plane Curves

\[
draw(ppc:PPC, bind:BIND, l:L DROP) ==
f := coordinate(ppc, 1); g := coordinate(ppc, 2)
-- create title if necessary
if not option?(l,"title" :: Symbol) then
  s:String := unparse(convert(f)@InputForm)
  if sayLength(s)$DisplayPackage > 50 then
    l := concat(title "AXIOM2D", l)
  else l := concat(title s, l)
-- create curve with functions as coordinates
curve : PPCF := curve(makeFloatFunction(f, variable bind),
                     makeFloatFunction(g, variable bind))$PPCF
-- call 'draw'
draw(curve, segment bind, l)
\]

\[
draw(ppc:PPC, bind:BIND) == draw(ppc, bind, nil())
\]

-- 3D - Curves (given by formulas)

\[
makeObject(psc:PSC, tBind:BIND, l:L DROP) ==
-- obtain dependent variable and coordinate functions
t := variable tBind; tSeg := segment tBind
f := coordinate(psc, 1); g := coordinate(psc, 2); h := coordinate(psc, 3)
-- create title if necessary
if not option?(l,"title" :: Symbol) then
  s:String := unparse(convert(f)@InputForm)
  if sayLength(s)$DisplayPackage > 50 then
    l := concat(title "AXIOM3D", l)
  else l := concat(title s, l)
-- indicate draw style if necessary
if not option?(l,"style" :: Symbol) then
\]
l := concat(style unparse(convert(f)@InputForm),l)
-- create curve with functions as coordinates
curve : PSCF := curve(makeFloatFunction(f,t),_
    makeFloatFunction(g,t),_
    makeFloatFunction(h,t))
-- call 'draw'
makeObject(curve,tSeg,l)

makeObject(psc:PSC,tBind:BIND) ==
    makeObject(psc,tBind,nil())

draw(psc:PSC,tBind:BIND,l:L DROP) ==
    -- obtain dependent variable and coordinate functions
    t := variable tBind; tSeg := segment tBind
    f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
    -- create title if necessary
    if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
            l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
    -- indicate draw style if necessary
    if not option?(l,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
    -- create curve with functions as coordinates
    curve : PSCF := curve(makeFloatFunction(f,t),_
        makeFloatFunction(g,t),_
        makeFloatFunction(h,t))
    -- call 'draw'
    draw(curve,tSeg,l)

draw(psc:PSC,tBind:BIND) ==
    draw(psc,tBind,nil())

------------------------------------------------------------------------
-- 3D - Surfaces (given by formulas)
------------------------------------------------------------------------
--% Three Dimensional Function Plots
makeObject(f:Ex,xBind:BIND,yBind:BIND,l:L DROP) ==
    -- create title if necessary
    if not option?(l,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
            l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
    -- indicate draw style if necessary
    if not option?(l,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
-- obtain dependent variables and their ranges
x := variable xBind; xSeg := segment xBind
y := variable yBind; ySeg := segment yBind
-- call 'draw'
makeObject(makeFloatFunction(f,x,y),xSeg,ySeg,l)

makeObject(f:Ex,xBind:BIND,yBind:BIND) ==
    makeObject(f,xBind,yBind,nil())

draw(f:Ex,xBind:BIND,yBind:BIND,1:L DROP) ==
    -- create title if necessary
    if not option?(1,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
            l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
    -- indicate draw style if necessary
    if not option?(1,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
    -- obtain dependent variables and their ranges
    x := variable xBind; xSeg := segment xBind
    y := variable yBind; ySeg := segment yBind
    -- call 'draw'
    draw(makeFloatFunction(f,x,y),xSeg,ySeg,l)

draw(f:Ex,xBind:BIND,yBind:BIND) ==
    draw(f,xBind,yBind,nil())

--% parametric surface
makeObject(s:PSF,uBind:BIND,vBind:BIND,1:L DROP) ==
    f := coordinate(s,1); g := coordinate(s,2); h := coordinate(s,3)
    if not option?(1,"title" :: Symbol) then
        s:String := unparse(convert(f)@InputForm)
        if sayLength(s)$DisplayPackage > 50 then
            l := concat(title "AXIOM3D",l)
        else l := concat(title s,l)
    if not option?(1,"style" :: Symbol) then
        l := concat(style unparse(convert(f)@InputForm),l)
    u := variable uBind; uSeg := segment uBind
    v := variable vBind; vSeg := segment vBind
    surf : PSFF := surface(makeFloatFunction(f,u,v), _
        makeFloatFunction(g,u,v), _
        makeFloatFunction(h,u,v))
    makeObject(surf,uSeg,vSeg,l)

makeObject(s:PSF,uBind:BIND,vBind:BIND) ==
    makeObject(s,uBind,vBind,nil())

draw(s:PSF,uBind:BIND,vBind:BIND,1:L DROP) ==
f := coordinate(s,1); g := coordinate(s,2); h := coordinate(s,3)

-- create title if necessary
if not option?(l,"title" :: Symbol) then
  s:String := unparse(convert(f)@InputForm)
  if sayLength(s)$DisplayPackage > 50 then
    l := concat(title "AXIOM3D",l)
  else l := concat(title s,l)

-- indicate draw style if necessary
if not option?(l,"style" :: Symbol) then
  l := concat(style unparse(convert(f)@InputForm),l)

-- obtain dependent variables and their ranges
u := variable uBind; uSeg := segment uBind
v := variable vBind; vSeg := segment vBind

-- create surface with functions as coordinates
surf : PSFF := surface(makeFloatFunction(f,u,v), _
  makeFloatFunction(g,u,v), _
  makeFloatFunction(h,u,v))

-- call 'draw'
draw(surf,uSeg,vSeg,l)

draw(s:PSF,uBind:BIND,vBind:BIND) ==
  draw(s,uBind,vBind,nil())

——

— DRAW.dotabb —

"DRAW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=DRAW"]
"ALIST" [color="#88FF44",href="bookvol10.3.pdf#nameddest=ALIST"]
"DRAW" -> "ALIST"

——

package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves

— TopLevelDrawFunctionsForAlgebraicCurves.input —

)set break resume
)sys rm -f TopLevelDrawFunctionsForAlgebraicCurves.output
)spool TopLevelDrawFunctionsForAlgebraicCurves.output
)set message test on
)set message auto off
---

---

---

---

---

---

---
TopLevelDrawFunctionsForAlgebraicCurves (DRAWCURV)

Exports:
draw

— package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves —

)abbrev package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves
++ Author: Clifton J. Williamson, Jon Steinbach
++ Date Created: 26 June 1990
++ Date Last Updated: October 1991
++ Description:
++ TopLevelDrawFunctionsForAlgebraicCurves provides top level
++ functions for drawing non-singular algebraic curves.

TopLevelDrawFunctionsForAlgebraicCurves(R,Ex):Exports ==> Implementation where
R : Join(IntegralDomain,OrderedSet,RetractableTo Integer)
Ex : FunctionSpace(R)

ANY1 ==> AnyFunctions1
DROP ==> DrawOption
EQ ==> Equation
F ==> Float
FRAC ==> Fraction
I ==> Integer
L ==> List
P ==> Polynomial
RN ==> Fraction Integer
SEG ==> Segment
SY ==> Symbol
VIEW2 ==> TwoDimensionalViewport

Exports ==> with
draw: (EQ Ex,SY,SY,L DROP) -> VIEW2
++ draw(f(x,y) = g(x,y),x,y,l) draws the graph of a polynomial
++ equation. The list l of draw options must specify a region
++ in the plane in which the curve is to be sketched.

Implementation ==> add
import ViewportPackage
import PlaneAlgebraicCurvePlot
import ViewDefaultsPackage
import GraphicsDefaults
import DrawOptionFunctions0
import SegmentFunctions2(RN,F)
import SegmentFunctions2(F,RN)
import AnyFunctions1(L SEG RN)

drawToScaleRanges: (SEG F,SEG F) -> L SEG F
drawToScaleRanges(xVals,yVals) ==
  -- warning: assumes window is square
  xHi := hi xVals; xLo := lo xVals
  yHi := hi yVals; yLo := lo yVals
  xDiff := xHi - xLo; yDiff := yHi - yLo
  pad := abs(yDiff - xDiff)/2
  yDiff > xDiff =>
    [segment(xLo - pad,xHi + pad),yVals]
    [xVals,segment(yLo - pad,yHi + pad)]

intConvert: R -> I
intConvert r ==
  (nn := retractIfCan(r)@Union(I,"failed")) case "failed" =>
    error "draw: polynomial must have rational coefficients"
    nn :: I

polyEquation: EQ Ex -> P I
polyEquation eq ==
  ff := lhs(eq) - rhs(eq)
  (r := retractIfCan(ff)@Union(FRAC P R,"failed")) case "failed" =>
    error "draw: not a polynomial equation"
  rat := r :: FRAC P R
  retractIfCan(denom rat)@Union(R,"failed") case "failed" =>
    error "draw: non-constant denominator"
  map(intConvert,numer rat)$PolynomialFunctions2(R,I)

draw(eq,x,y,l) ==
  -- obtain polynomial equation
  p := polyEquation eq
  -- extract ranges from option list
  floatRange := option(l,"rangeFloat" :: Symbol)
  ratRange := option(l,"rangeRat" :: Symbol)
  (floatRange case "failed") and (ratRange case "failed") =>
    error "draw: you must specify ranges for an implicit plot"
  ranges : L SEG RN := nil() -- dummy value
  floatRanges : L SEG F := nil() -- dummy value
xRange : SEG RN := segment(0,0) -- dummy value
yRange : SEG RN := segment(0,0) -- dummy value
xRangeFloat : SEG F := segment(0,0) -- dummy value
yRangeFloat : SEG F := segment(0,0) -- dummy value
if not ratRange case "failed" then
    ranges := retract(ratRange :: Any)$ANY1(L SEG RN)
    not size?(ranges,2) => error "draw: you must specify two ranges"
xRange := first ranges; yRange := second ranges
xRangeFloat := map((s:RN):F+->convert(s)@Float,xRange)@(SEG F)
yRangeFloat := map((s:RN):F+->convert(s)@Float,yRange)@(SEG F)
floatRanges := [xRangeFloat,yRangeFloat]
else
    floatRanges := retract(floatRange :: Any)$ANY1(L SEG F)
    not size?(floatRanges,2) =>
        error "draw: you must specify two ranges"
xRangeFloat := first floatRanges
yRangeFloat := second floatRanges
xRange := map((s:F):RN+->retract(s)@RN,xRangeFloat)@(SEG RN)
yRange := map((s:F):RN+->retract(s)@RN,yRangeFloat)@(SEG RN)
ranges := [xRange,yRange]
-- create curve plot
acplot := makeSketch(p,x,y,xRange,yRange)
-- process scaling information
if toScale(l,drawToScale()) then
    scaledRanges := drawToScaleRanges(xRangeFloat,yRangeFloat)
    -- add scaled ranges to list of options
    l := concat(ranges scaledRanges,l)
else
    -- add ranges to list of options
    l := concat(ranges floatRanges,l)
-- process color information
ptCol := pointColorPalette(l,pointColorDefault())
crCol := curveColorPalette(l,lineColorDefault())
-- draw
drawCurves(listBranches acplot,ptCol,crCol,pointSizeDefault(),l)
package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions

— TopLevelDrawFunctionsForCompiledFunctions.input —

)set break resume
)sys rm -f TopLevelDrawFunctionsForCompiledFunctions.output
)spool TopLevelDrawFunctionsForCompiledFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TopLevelDrawFunctionsForCompiledFunctions

--R
--R TopLevelDrawFunctionsForCompiledFunctions is a package constructor
--R Abbreviation for TopLevelDrawFunctionsForCompiledFunctions is DRAWCFUN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for DRAWCFUN
--R
--R----------------------------------- Operations -----------------------------------
--R draw : ((DoubleFloat -> DoubleFloat),Segment(Float),List(DrawOption)) -> TwoDimensionalViewport
--R draw : ((DoubleFloat -> DoubleFloat),Segment(Float)) -> TwoDimensionalViewport
--R draw : (ParametricPlaneCurve((DoubleFloat -> DoubleFloat)),Segment(Float),List(DrawOption)) -> TwoDimensionalViewport
--R draw : (ParametricPlaneCurve((DoubleFloat -> DoubleFloat)),Segment(Float)) -> TwoDimensionalViewport
--R draw : (ParametricSpaceCurve((DoubleFloat -> DoubleFloat)),Segment(Float),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : (ParametricSpaceCurve((DoubleFloat -> DoubleFloat)),Segment(Float)) -> ThreeDimensionalViewport
--R draw : ((DoubleFloat -> Point(DoubleFloat)),Segment(Float),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : ((DoubleFloat -> Point(DoubleFloat)),Segment(Float)) -> ThreeDimensionalViewport
--R draw : (((DoubleFloat,DoubleFloat) -> DoubleFloat),Segment(Float),Segment(Float),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : (((DoubleFloat,DoubleFloat) -> DoubleFloat),Segment(Float),Segment(Float)) -> ThreeDimensionalViewport
--R draw : (((DoubleFloat,DoubleFloat) -> Point(DoubleFloat)),Segment(Float),Segment(Float),List(DrawOption)) -> ThreeDimensionalViewport
--R draw : (((DoubleFloat,DoubleFloat) -> Point(DoubleFloat)),Segment(Float),Segment(Float)) -> ThreeDimensionalViewport
--R makeObject : (ParametricSpaceCurve((DoubleFloat -> DoubleFloat)),Segment(Float),List(DrawOption)) -> ThreeSpace(DoubleFloat)
--R makeObject : (ParametricSpaceCurve((DoubleFloat -> DoubleFloat)),Segment(Float)) -> ThreeSpace(DoubleFloat)
--R makeObject : ((DoubleFloat -> Point(DoubleFloat)),Segment(Float),List(DrawOption)) -> ThreeSpace(DoubleFloat)
--R makeObject : ((DoubleFloat -> Point(DoubleFloat)),Segment(Float)) -> ThreeSpace(DoubleFloat)
--R makeObject : (((DoubleFloat,DoubleFloat) -> DoubleFloat),Segment(Float),Segment(Float),List(DrawOption)) -> ThreeSpace(DoubleFloat)
--R makeObject : (((DoubleFloat,DoubleFloat) -> DoubleFloat),Segment(Float),Segment(Float)) -> ThreeSpace(DoubleFloat)
--R makeObject : (((DoubleFloat,DoubleFloat) -> Point(DoubleFloat)),Segment(Float),Segment(Float),List(DrawOption)) -> ThreeSpace(DoubleFloat)
--R makeObject : (((DoubleFloat,DoubleFloat) -> Point(DoubleFloat)),Segment(Float),Segment(Float)) -> ThreeSpace(DoubleFloat)
--R makeObject : (ParametricSurface(((DoubleFloat,DoubleFloat) -> DoubleFloat)),Segment(Float),Segment(Float),List(DrawOption)) -> ThreeSpace(DoubleFloat)
--R makeObject : (ParametricSurface(((DoubleFloat,DoubleFloat) -> DoubleFloat)),Segment(Float),Segment(Float)) -> ThreeSpace(DoubleFloat)
--R recolor : (((DoubleFloat,DoubleFloat) -> Point(DoubleFloat)),((DoubleFloat,DoubleFloat,DoubleFloat) -> DoubleFloat)) -> ((DoubleFloat,DoubleFloat) -> Point(DoubleFloat))
TopLevelDrawFunctionsForCompiledFunctions helps draw graphics of expressions.

See Also:
- `show TopLevelDrawFunctionsForCompiledFunctions`

---

**Exports:**
- `draw`
- `makeObject`
- `recolor`

---

`)abbrev package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions`  
++ Author: Clifton J. Williamson, Scott Morrison  
++ Date Created: 22 June 1990  
++ Date Last Updated: January 1992
++ Description:
++ TopLevelDrawFunctionsForCompiledFunctions provides top level
++ functions for drawing graphics of expressions.

TopLevelDrawFunctionsForCompiledFunctions():
Exports == Implementation where
    ANY1 ==> AnyFunctions1
    B  ==> Boolean
    F  ==> Float
    L  ==> List
    SEG ==> Segment Float
    SF  ==> DoubleFloat
    DROP ==> DrawOption
    PLOT ==> Plot
    PPC ==> ParametricPlaneCurve(SF -> SF)
    PSC ==> ParametricSpaceCurve(SF -> SF)
    PSF ==> ParametricSurface((SF,SF) -> SF)
    Pt  ==> Point SF
    PSFUN ==> (SF, SF) -> Pt
    PCFUN ==> SF -> Pt
    SPACE3 ==> ThreeSpace(SF)
    VIEW2 ==> TwoDimensionalViewport
    VIEW3 ==> ThreeDimensionalViewport

Exports ==> with

--% Two Dimensional Function Plots

draw: (SF -> SF,SEG,L DROP) -> VIEW2
   ++ draw(f,a..b,l) draws the graph of \spad{y = f(x)} as x
   ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.
   ++ The options contained in the list l of the
   ++ domain \spad{DrawOption} are applied.
draw: (SF -> SF,SEG) -> VIEW2
   ++ draw(f,a..b) draws the graph of \spad{y = f(x)} as x
   ++ ranges from \spad{min(a,b)} to \spad{max(a,b)}.

--% Parametric Plane Curves

draw: (PPC,SEG,L DROP) -> VIEW2
   ++ draw(curve(f,g),a..b,l) draws the graph of the parametric
   ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
   ++ \spad{max(a,b)}.
   ++ The options contained in the list l of the domain \spad{DrawOption}
   ++ are applied.
draw: (PPC,SEG) -> VIEW2
   ++ draw(curve(f,g),a..b) draws the graph of the parametric
   ++ curve \spad{x = f(t), y = g(t)} as t ranges from \spad{min(a,b)} to
   ++ \spad{max(a,b)}. 
--% Parametric Space Curves

\begin{verbatim}
draw: (PSC,SEG,L DROP) -> VIEW3
   ++ draw(curve(f,g,h),a..b,l) draws the graph of the parametric
   ++ curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
   ++ \spad{\min(a,b)} to \spad{\max(a,b)}.
   ++ The options contained in the list l of the domain
   ++ \spad{DrawOption} are applied.
draw: (PSC,SEG) -> VIEW3
   ++ draw(curve(f,g,h),a..b,l) draws the graph of the parametric
   ++ curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
   ++ \spad{\min(a,b)} to \spad{\max(a,b)}.
draw: (PCFUN,SEG,L DROP) -> VIEW3
   ++ draw(f,a..b,l) draws the graph of the parametric
   ++ curve \spad{f} as t ranges from
   ++ \spad{\min(a,b)} to \spad{\max(a,b)}.
   ++ The options contained in the list l of the domain
   ++ \spad{DrawOption} are applied.
draw: (PCFUN,SEG) -> VIEW3
   ++ draw(f,a..b,l) draws the graph of the parametric
   ++ curve \spad{f} as t ranges from
   ++ \spad{\min(a,b)} to \spad{\max(a,b)}.
makeObject: (PSC,SEG,L DROP) -> SPACE3
   ++ makeObject(curve(f,g,h),a..b,l) returns a space of the
   ++ domain \spad{ThreeSpace} which contains the graph of the
   ++ parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
   ++ \spad{\min(a,b)} to \spad{\max(a,b)}.
   ++ The options contained in the list l of the domain
   ++ \spad{DrawOption} are applied.
makeObject: (PSC,SEG) -> SPACE3
   ++ makeObject(sp,curve(f,g,h),a..b) returns the space \spad{sp}
   ++ of the domain \spad{ThreeSpace} with the addition of the graph
   ++ of the parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t
   ++ ranges from \spad{\min(a,b)} to \spad{\max(a,b)}.
makeObject: (PCFUN,SEG,L DROP) -> SPACE3
   ++ makeObject(curve(f,g,h),a..b,l) returns a space of the
   ++ domain \spad{ThreeSpace} which contains the graph of the
   ++ parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t ranges from
   ++ \spad{\min(a,b)} to \spad{\max(a,b)}.
   ++ The options contained in the list l of the domain
   ++ \spad{DrawOption} are applied.
makeObject: (PCFUN,SEG) -> SPACE3
   ++ makeObject(sp,curve(f,g,h),a..b) returns the space \spad{sp}
   ++ of the domain \spad{ThreeSpace} with the addition of the graph
   ++ of the parametric curve \spad{x = f(t), y = g(t), z = h(t)} as t
   ++ ranges from \spad{\min(a,b)} to \spad{\max(a,b)}.
\end{verbatim}

--% Three Dimensional Function Plots
draw: ((SF,SF) -> SF,SEG,SEG,L DROP) -> VIEW3
++ draw(f,a..b,c..d,l) draws the graph of \spad{z = f(x,y)}
++ as x ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and y ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)}.
++ and the options contained in the list l of the domain
++ \spad{DrawOption} are applied.
draw: ((SF,SF) -> SF,SEG,SEG) -> VIEW3
++ draw(f,a..b,c..d) draws the graph of \spad{z = f(x,y)}
++ as x ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and y ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)}.
makeObject: ((SF,SF) -> SF,SEG,SEG,L DROP) -> SPACE3
++ makeObject(f,a..b,c..d,l) returns a space of the domain
++ \spadtype{ThreeSpace} which contains the graph of \spad{z = f(x,y)}
++ as x ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and y ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)}, and the options contained in the
++ list l of the domain \spad{DrawOption} are applied.
makeObject: ((SF,SF) -> SF,SEG,SEG) -> SPACE3
++ makeObject(f,a..b,c..d) returns a space of the domain
++ \spadtype{ThreeSpace} which contains the graph of \spad{z = f(x,y)}
++ as x ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and y ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)}.

--% Parametric Surfaces

draw: (PSFUN, SEG, SEG, L DROP) -> VIEW3
++ draw(f,a..b,c..d) draws the graph of the parametric surface \spad{f(u,v)}
++ as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)}
++ and v ranges from \spad{\min(c,d)} to \spad{\max(c,d)}.
++ The options contained in the
++ list l of the domain \spad{DrawOption} are applied.
draw: (PSFUN, SEG, SEG) -> VIEW3
++ draw(f,a..b,c..d) draws the graph of the parametric surface \spad{f(u,v)}
++ as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)}
++ and v ranges from \spad{\min(c,d)} to \spad{\max(c,d)}
++ The options contained in the list
++ l of the domain \spad{DrawOption} are applied.
makeObject: (PSFUN, SEG, SEG, L DROP) -> SPACE3
++ makeObject(f,a..b,c..d,l) returns a space of the domain
++ \spadtype{ThreeSpace} which contains the graph of the parametric surface \spad{f(u,v)}
++ as u ranges from \spad{\min(a,b)} to \spad{\max(a,b)} and v ranges from
++ \spad{\min(c,d)} to \spad{\max(c,d)};
++ The options contained in the
++ list l of the domain \spad{DrawOption} are applied.
makeObject: (PSFUN, SEG, SEG) -> SPACE3
++ makeObject(f,a..b,c..d) returns a space of the domain
++ \spadtype{ThreeSpace} which contains the graph of the parametric surface \spad{f(u,v)}
++ as u ranges from \texttt{min(a,b)} to \texttt{max(a,b)} and v ranges from \texttt{min(c,d)} to \texttt{max(c,d)}.

\texttt{draw: (PSF,SEG,SEG,L DROP) \rightarrow VIEW3}
++ \texttt{draw(surface(f,g,h),a..b,c..d)} draws the
++ graph of the parametric surface \texttt{spad{x = f(u,v)}, spad{y = g(u,v)},}
++ \texttt{spad{z = h(u,v)}} as u ranges from \texttt{spad{min(a,b)}} to \texttt{spad{max(a,b)}}
++ and v ranges from \texttt{spad{min(c,d)}} to \texttt{spad{max(c,d)}};
++ The options contained in the
++ list 1 of the domain \texttt{spad{DrawOption}} are applied.

\texttt{draw: (PSF,SEG,SEG) \rightarrow VIEW3}
++ \texttt{draw(surface(f,g,h),a..b,c..d)} draws the
++ graph of the parametric surface \texttt{spad{x = f(u,v)}, spad{y = g(u,v)},}
++ \texttt{spad{z = h(u,v)}} as u ranges from \texttt{spad{min(a,b)}} to \texttt{spad{max(a,b)}}
++ and v ranges from \texttt{spad{min(c,d)}} to \texttt{spad{max(c,d)}}.

\texttt{makeObject: (PSF,SEG,SEG,L DROP) \rightarrow SPACE3}
++ \texttt{makeObject(surface(f,g,h),a..b,c..d,l)} returns a
++ space of the domain \texttt{spadtype{ThreeSpace}} which contains the
++ graph of the parametric surface \texttt{spad{x = f(u,v)}, spad{y = g(u,v)},}
++ \texttt{spad{z = h(u,v)}} as u ranges from \texttt{spad{min(a,b)}} to \texttt{spad{max(a,b)}}
++ and v ranges from \texttt{spad{min(c,d)}} to \texttt{spad{max(c,d)}}.
++ The options contained in the
++ list 1 of the domain \texttt{spad{DrawOption}} are applied.

\texttt{makeObject: (PSF,SEG,SEG) \rightarrow SPACE3}
++ \texttt{makeObject(surface(f,g,h),a..b,c..d,l)} returns a
++ space of the domain \texttt{spadtype{ThreeSpace}} which contains the
++ graph of the parametric surface \texttt{spad{x = f(u,v)}, spad{y = g(u,v)},}
++ \texttt{spad{z = h(u,v)}} as u ranges from \texttt{spad{min(a,b)}} to \texttt{spad{max(a,b)}}
++ and v ranges from \texttt{spad{min(c,d)}} to \texttt{spad{max(c,d)}}.

\texttt{recolor: ((SF,SF) \rightarrow Pt,(SF,SF,SF) \rightarrow SF) \rightarrow ((SF,SF) \rightarrow Pt)}
++ \texttt{recolor()}, uninteresting to top level user; exported in order to
++ compile package.

\textbf{Implementation} \Rightarrow \texttt{add}

I have had to work my way around the following bug in the compiler: When a local variable is given a mapping as a value, e.g.

\texttt{foo : SF \rightarrow SF := makeFloatFunction(f,t)},

the compiler cannot distinguish that local variable from a local function defined elsewhere in the package. Thus, when 'foo' is passed to a function, e.g.

\texttt{bird := fcn(foo)},

foo will often be compiled as DRAW;foo rather than foo. This, of course, causes a run-time error.

To avoid this problem, local variables are not given mappings as values, but rather (singleton) lists of mappings. The first element of the list can always be extracted and everything
goes through as before. There is no major loss in efficiency, as the computation of points will always dominate the computation time.
- cjw, 22 June MCMXC

--- package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions ---

```lisp
import PLOT
import TwoDimensionalPlotClipping
import GraphicsDefaults
import ViewportPackage
import ThreeDimensionalViewport
import DrawOptionFunctions0
import MakeFloatCompiledFunction(Ex)
import MeshCreationRoutinesForThreeDimensions
import SegmentFunctions2(SF,Float)
import ViewDefaultsPackage
import AnyFunctions1(Pt -> Pt)
import AnyFunctions1((SF,SF,SF) -> SF)
import DrawOptionFunctions0
import SPACE3

EXTOVARERROR : String := _
"draw: when specifying function, left hand side must be a variable"
SMALLRANGEERROR : String := _
"draw: range is in interval with only one point"
DEPVARERROR : String := _
"draw: independent variable appears on lhs of function definition"
```

```
-- 2D - draw’s

```

drawToScaleRanges: (Segment SF,Segment SF) -> L SEG
drawToScaleRanges(xVals,yVals) ==
  -- warning: assumes window is square
  xHi := convert(hi xVals)@Float; xLo := convert(lo xVals)@Float
  yHi := convert(hi yVals)@Float; yLo := convert(lo yVals)@Float
  xDiff := xHi - xLo; yDiff := yHi - yLo
  pad := abs(yDiff - xDiff)/2
  yDiff > xDiff =>
    [segment(xLo - pad,xHi + pad),map(x +-> convert(x)@Float,yVals)]
  [map(x +-> convert(x)@Float,xVals),segment(yLo - pad,yHi + pad)]

drawPlot: (PLOT,L DROP) -> VIEW2
drawPlot(plot,l) ==
  branches := listBranches plot
  xRange := xRange plot; yRange := yRange plot
  -- process clipping information
  if (cl := option(l,"clipSegment" :: Symbol)) case "failed" then
```
if clipBoolean(l,clipPointsDefault()) then
  clipInfo :=
    parametric? plot => clipParametric plot
clip plot
branches := clipInfo.brans
xRange := clipInfo.xValues; yRange := clipInfo.yValues
else
  "No explicit user-specified clipping"
else
  segList := retract(cl :: Any)$ANY1(L SEG)
  empty? segList =>
    error "draw: you may specify at least 1 segment for 2D clipping"
  more?(segList,2) =>
    error "draw: you may specify at most 2 segments for 2D clipping"
xLo : SF := 0; xHi : SF := 0; yLo : SF := 0; yHi : SF := 0
if empty? rest segList then
  xLo := lo xRange; xHi := hi xRange
yRangeF := first segList
yLo := convert(lo yRangeF)@SF; yHi := convert(hi yRangeF)@SF
else
  xRangeF := first segList
  xLo := convert(lo xRangeF)@SF; xHi := convert(hi xRangeF)@SF
yRangeF := second segList
  yLo := convert(lo yRangeF)@SF; yHi := convert(hi yRangeF)@SF
clipInfo := clipWithRanges(branches,xLo,xHi,yLo,yHi)
branches := clipInfo.brans
xRange := clipInfo.xValues; yRange := clipInfo.yValues
-- process scaling information
if toScale(l,drawToScale()) then
  scaledRanges := drawToScaleRanges(xRange,yRange)
  -- add scaled ranges to list of options
  l := concat(ranges scaledRanges,l)
else
  xRangeFloat : SEG := map(x +-> convert(x)@Float,xRange)
yRangeFloat : SEG := map(x +-> convert(x)@Float,yRange)
  -- add ranges to list of options
  l := concat(ranges(ll : L SEG := 
xRangeFloat,yRangeFloat)),l)
-- process color information
ptCol := pointColorPalette(l,pointColorDefault())
crCol := curveColorPalette(l,lineColorDefault())
-- draw
drawCurves(branches,ptCol,crCol,pointSizeDefault(),l)

normalize: SEG -> Segment SF
normalize seg ==
  -- normalize [a,b]:
  -- error if a = b, returns [a,b] if a < b, returns [b,a] if b > a
  a := convert(lo seg)@SF; b := convert(hi seg)@SF
  a = b => error SMALLRANGEERROR
  a < b => segment(a,b)
The function `myTrap1` is a local function for used in creating maps SF → Point SF (two dimensional). The range of this function is SingleFloat. As originally coded it would return $\text{NaN}$ value $\text{Lisp}$ which is outside the range. Since this function is only used internally by the draw package we handle the “failed” case by returning zero. We handle the out-of-range case by returning the maximum or minimum SingleFloat value.

```lisp
package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions

segment(b,a)

myTrap1: (SF→ SF, SF) -> SF
myTrap1(ff:SF→ SF, f:SF):SF ==
  s := trapNumericErrors(ff(f))$Lisp :: Union(SF, "failed")
  s case "failed" => 0
  r:=s::SF
  r >max()$SF => max()$SF
  r < min()$SF => min()$SF
  r

makePt2: (SF,SF) -> Point SF
makePt2(x,y) == point(l : List SF := [x,y])

--% Two Dimensional Function Plots

draw(f:SF -> SF,seg:SEG,l:L DROP) ==
  -- set adaptive plotting off or on
  oldAdaptive := adaptive?()$PLOT
  setAdaptive(adaptive(l,oldAdaptive))$PLOT
  -- create function SF → Point SF
  ff : L(SF -> Point SF) := [x +-> makePt2(myTrap1(f,x),x)]
  -- process change of coordinates
  if (c := option(l,"coordinates" :: Symbol)) case "failed" then
    -- default coordinate transformation
    ff := [x +-> makePt2(x,myTrap1(f,x))]
  else
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    ff := [x +-> (first cc)((first ff)(x))]
  -- create PLOT
  pl := pointPlot(first ff,normalize seg)
  -- reset adaptive plotting
  setAdaptive(oldAdaptive)$PLOT
  -- draw
  drawPlot(pl,l)

draw(f:SF -> SF,seg:SEG) == draw(f,seg,nil())

--% Parametric Plane Curves
```
draw(ppc: PPC, seg: SEG, l:L DROP) ==
   -- set adaptive plotting off or on
   oldAdaptive := adaptive?()$PLOT
   setAdaptive((adaptive(l, oldAdaptive))$PLOT
   -- create function SF -> Point SF
   f := coordinate(ppc,1); g := coordinate(ppc,2)
   fcn := L(SF -> Pt) := [x +-> makePt2(myTrap1(f,x),myTrap1(g,x))]
   -- process change of coordinates
   if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
      cc := L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
      fcn := [x +-> (first cc)((first fcn)(x))]
   -- create PLOT
   pl := pointPlot(first fcn, normalize seg)
   -- reset adaptive plotting
   setAdaptive(oldAdaptive)$PLOT
   -- draw
   drawPlot(pl,l)

draw(ppc: PPC, seg: SEG) == draw(ppc, seg, nil())

------------------------------------------------------------------------
-- 3D - Curves
------------------------------------------------------------------------

--% functions for creation of maps SF -> Point SF (three dimensional)
makePt4: (SF,SF,SF,SF) -> Point SF
makePt4(x,y,z,c) == point(l : List SF := [x,y,z,c])

--% Parametric Space Curves
id: SF -> SF
id x == x
zCoord: (SF,SF,SF) -> SF
zCoord(x,y,z) == z

colorPoints: (List List Pt,(SF,SF,SF) -> SF) -> List List Pt
colorPoints(llp,func) ==
   for lp in llp repeat for p in lp repeat
      p.4 := func(p.1,p.2,p.3)
   llp

makeObject(psc:PSC, seg: SEG, l:L DROP) ==
   sp := space l
   -- obtain dependent variable and coordinate functions
   f := coordinate(psc,1); g := coordinate(psc,2); h := coordinate(psc,3)
   -- create function SF -> Point SF with default or user-specified
   -- color function
fcn : L(SF -> Pt) := [x +-> makePt4(myTrap1(f,x), myTrap1(g,x), myTrap1(h,x), myTrap1(id,x))]
pointsColored? : Boolean := false
if not (c1 := option(l,"colorFunction1" :: Symbol)) case "failed" then
pointsColored? := true
fcn := [x +-> makePt4(myTrap1(f,x), myTrap1(g,x), myTrap1(h,x),
        retract(c1 :: Any)$ANY1(SF -> SF)(x))]
-- process change of coordinates
if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
  cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
  fcn := [x +-> (first cc)((first fcn)(x))]
-- create PLOT
pl := pointPlot(first fcn,normalize seg)$Plot3D
-- create ThreeSpace
s := sp
-- draw Tube
-- print(pl::OutputForm)
option?(l,"tubeRadius" :: Symbol) =>
  pts := tubePoints(l,1,8)
  rad := convert(tubeRadius(l,0.25))@DoubleFloat
  tub := tube(pl,rad,pts)$NumericTubePlot(Plot3D)
  loops := listLoops tub
-- color points if this has not been done already
if not pointsColored? then
  if (c3 := option(l,"colorFunction3" :: Symbol)) case "failed" then
    then colorPoints(loops, zCoord) -- default color function
  else colorPoints(loops, retract(c3 :: Any)$ANY1((SF,SF,SF) -> SF))
  mesh(s,loops,false,false)
  s
-- draw curve
br := listBranches pl
for b in br repeat curve(s,b)
s
makeObject(psc(PCFUN,seg:SEG,l:L DROP) ==
  sp := space l
  -- create function SF -> Point SF with default or user-specified
  -- color function
  fcn : L(SF -> Pt) := [psc]
  pointsColored? : Boolean := false
  if not (c1 := option(l,"colorFunction1" :: Symbol)) case "failed" then
    pointsColored? := true
    fcn := [x +-> concat(psc(x), retract(c1 :: Any)$ANY1(SF -> SF)(x))]
  -- process change of coordinates
  if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
    cc := L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    fcn := [x +-> (first cc)((first fcn)(x))]
  -- create PLOT
  pl := pointPlot(first fcn,normalize seg)$Plot3D
  -- create ThreeSpace
s := sp
-- draw Tube
option?(l,"tubeRadius" :: Symbol) =>
  pts := tubePoints(l,8)
  rad := convert(tubeRadius(l,0.25))@DoubleFloat
  tub := tube(pl,rad,pts)$NumericTubePlot(Plot3D)
  loops := listLoops tub
-- color points if this has not been done already
mesh(s,loops,false,false)
  s
-- draw curve
br := listBranches pl
for b in br repeat curve(s,b)
  s

makeObject(psc:PSC,seg:SEG) ==
  makeObject(psc,seg,nil())
makeObject(psc:PCFUN,seg:SEG) ==
  makeObject(psc,seg,nil())
draw(psc:PSC,seg:SEG,1:L DROP) ==
  sp := makeObject(psc,seg,1)
  makeViewport3D(sp, 1)
draw(psc:PSC,seg:SEG) ==
  draw(psc,seg,nil())
draw(psc:PCFUN,seg:SEG,1:L DROP) ==
  sp := makeObject(psc,seg,1)
  makeViewport3D(sp, 1)
draw(psc:PCFUN,seg:SEG) ==
  draw(psc,seg,nil())

------------------------------------------------------------------------
-- 3D - Surfaces

The function myTrap2 is a local function for used in creating maps SF -i Point SF (three dimensional). The range of this function is SingleFloat. As originally coded it would return $NaNValue$Lisp which is outside the range. Since this function is only used internally by the draw package we handle the “failed” case by returning zero. We handle the out-of-range case by returning the maximum or minimum SingleFloat value.

— package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions —
myTrap2: ((SF, SF) -> SF, SF, SF) -> SF
myTrap2(ff:(SF, SF) -> SF, u:SF, v:SF):SF ==
s := trapNumericErrors(ff(u, v))$Lisp :: Union(SF, "failed")
s case "failed" => 0
r:SF := s::SF
r > max()$SF => max()$SF
r < min()$SF => min()$SF
r
recolor(ptFunc,colFunc) ==
  (f1,f2) ->
  pt := ptFunc(f1,f2)
  pt.4 := colFunc(pt.1,pt.2,pt.3)
  pt

xCoord: (SF,SF) -> SF
xCoord(x,y) == x

--% Three Dimensional Function Plots

makeObject(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG,l:L DROP) ==
  sp := space l
  -- process color function of two variables
  col2 : L((SF,SF) -> SF) := [xCoord]  -- dummy color function
  pointsColored? : Boolean := false
  if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
    pointsColored? := true
    col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
  fcn : L((SF,SF) -> Pt) :=
    [(x,y) +-> makePt4(myTrap2(f,x,y),x,y,(first col2)(x,y))]
  -- process change of coordinates
  if (c := option(l,"coordinates" :: Symbol)) case "failed" then
    -- default coordinate transformation
    fcn := [(x,y) +-> makePt4(x,y,myTrap2(f,x,y),(first col2)(x,y))]
  else
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
  -- process color function of three variables, if there was no
  -- color function of two variables
  if not pointsColored? then
    c := option(l,"colorFunction3" :: Symbol)
    fcn :=
    c case "failed" => [recolor((first fcn),zCoord)]
    [recolor((first fcn),retract(c :: Any)$ANY1((SF,SF,SF) -> SF))]
  -- create mesh
  mesh := meshPar2Var(sp,first fcn,normalize xSeg,normalize ySeg,1)
  mesh

makeObject(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG) ==
  makeObject(f,xSeg,ySeg,nil())
draw(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG,l:L DROP) ==
  sp := makeObject(f, xSeg, ySeg, l)
  makeViewport3D(sp, l)

draw(f:(SF,SF) -> SF,xSeg:SEG,ySeg:SEG) ==
  draw(f, xSeg, ySeg, nil())

--% parametric surface

makeObject(s:PSF,uSeg:SEG,vSeg:SEG,l:L DROP) ==
  sp := space l
  -- create functions from expressions
  f : L((SF,SF) -> SF) := [coordinate(s,1)]
  g : L((SF,SF) -> SF) := [coordinate(s,2)]
  h : L((SF,SF) -> SF) := [coordinate(s,3)]
  -- process color function of two variables
  col2 : L((SF,SF) -> SF) := [xCoord] -- dummy color function
  pointsColored? : Boolean := false
  if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
    pointsColored? := true
    col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
  fcn : L((SF,SF) -> Pt) :=
    [(x,y) +-> makePt4(myTrap2((first f),x,y),myTrap2((first g),x,y),
      myTrap2((first h),x,y), myTrap2((first col2),x,y))]
  -- process change of coordinates
  if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
    cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
    fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
  -- process color function of three variables, if there was no
  -- color function of two variables
  if not pointsColored? then
    col3 : L((SF,SF,SF) -> SF) := [zCoord] -- default color function
    if not (c := option(l,"colorFunction3" :: Symbol)) case "failed" then
      col3 := [retract(c :: Any)$ANY1((SF,SF,SF) -> SF)]
    fcn := [recolor((first fcn),(first col3))]
  -- create mesh
  mesh := meshPar2Var(sp,first fcn,normalize uSeg,normalize vSeg,l)
  mesh

makeObject(s:PSFUN,uSeg:SEG,vSeg:SEG,l:L DROP) ==
  sp := space l
  -- process color function of two variables
  col2 : L((SF,SF) -> SF) := [xCoord] -- dummy color function
  pointsColored? : Boolean := false
  if not (c2 := option(l,"colorFunction2" :: Symbol)) case "failed" then
    pointsColored? := true
    col2 := [retract(c2 :: Any)$ANY1((SF,SF) -> SF)]
  fcn : L((SF,SF) -> Pt) :=
    pointsColored? => [(x,y) +-> concat(s(x, y), (first col2)(x, y))]
-- process change of coordinates
if not (c := option(l,"coordinates" :: Symbol)) case "failed" then
  cc : L(Pt -> Pt) := [retract(c :: Any)$ANY1(Pt -> Pt)]
  fcn := [(x,y) +-> (first cc)((first fcn)(x,y))]
-- create mesh
mesh := meshPar2Var(sp,first fcn,normalize uSeg,normalize vSeg,l)
  mesh

makeObject(s:PSF,uSeg:SEG,vSeg:SEG) ==
  makeObject(s,uSeg,vSeg,nil())

draw(s:PSF,uSeg:SEG,vSeg:SEG,l:L DROP) ==
  -- draw
  mesh := makeObject(s,uSeg,vSeg,l)
  makeViewport3D(mesh,l)

draw(s:PSF,uSeg:SEG,vSeg:SEG) ==
  draw(s,uSeg,vSeg,nil())

makeObject(s:PSFUN,uSeg:SEG,vSeg:SEG) ==
  makeObject(s,uSeg,vSeg,nil())

draw(s:PSFUN,uSeg:SEG,vSeg:SEG,l:L DROP) ==
  -- draw
  mesh := makeObject(s,uSeg,vSeg,l)
  makeViewport3D(mesh,l)

draw(s:PSFUN,uSeg:SEG,vSeg:SEG) ==
  draw(s,uSeg,vSeg,nil())

package DRAWPT TopLevelDrawFunctionsForPoints

— TopLevelDrawFunctionsForPoints.input —
TopLevelDrawFunctionsForPoints provides top level functions for
drawing curves and surfaces described by sets of points.

See Also:
o )show TopLevelDrawFunctionsForPoints
Exports:
draw

— package DRAWPOT TopLevelDrawFunctionsForPoints —

)abbrev package DRAWPOT TopLevelDrawFunctionsForPoints
++ Author: Mike Dewar
++ Date Created: 24 May 1995
++ Date Last Updated: 25 November 1996
++ Description:
++ TopLevelDrawFunctionsForPoints provides top level functions
++ for drawing curves and surfaces described by sets of points.

TopLevelDrawFunctionsForPoints(): Exports == Implementation where

DROP ==> DrawOption
L ==> List
SF ==> DoubleFloat
Pt ==> Point SF
VIEW2 ==> TwoDimensionalViewport
VIEW3 ==> ThreeDimensionalViewport

Exports ==> with
draw: (L SF,L SF) -> VIEW2
  ++ draw(lx,ly) plots the curve constructed of points (x,y) for x
  ++ in \spad{lx} for y in \spad{ly}.
draw: (L SF,L SF,L DROP) -> VIEW2
  ++ draw(lx,ly,l) plots the curve constructed of points (x,y) for x
  ++ in \spad{lx} for y in \spad{ly}.
  ++ The options contained in the list l of
  ++ the domain \spad{DrawOption} are applied.
draw: (L Pt) -> VIEW2
  ++ draw(lp) plots the curve constructed from the list of points lp.
draw: (L Pt,L DROP) -> VIEW2
  ++ draw(lp,l) plots the curve constructed from the list of points lp.
The options contained in the list l of the domain \spad{DrawOption} are applied.

draw: (L SF, L SF, L SF) -> VIEW3
++ draw(lx,ly,lz) draws the surface constructed by projecting the values in the \axiom{lz} list onto the rectangular grid formed by the
++ \axiom{lx x ly}.

draw: (L SF, L SF, L SF, L DROP) -> VIEW3
++ draw(lx,ly,lz,l) draws the surface constructed by
++ projecting the values
++ in the \axiom{lz} list onto the rectangular grid formed by the
++ The options contained in the list l of the domain \spad{DrawOption}
++ are applied.

Implementation ==> add

\begin{verbatim}
\texttt{draw}(lp:L Pt,l:L DROP):VIEW2 ==
  makeViewport2D(makeGraphImage([lp])$GraphImage,l)$VIEW2

\texttt{draw}(lp:L Pt):VIEW2 == draw(lp,[
]

\texttt{draw}(lx: L SF, ly: L SF, l:L DROP):VIEW2 ==
  draw([point([x,y])$Pt for x in lx for y in ly],l)

\texttt{draw}(lx: L SF, ly: L SF):VIEW2 == draw(lx,ly,[
]

\texttt{draw}(x:L SF,y:L SF,z:L SF):VIEW3 == draw(x,y,z,[
]

\texttt{draw}(x:L SF,y:L SF,z:L SF,l:L DROP):VIEW3 ==
m : Integer := \#x
  zero? m => error "No X values"
  n : Integer := \#y
  zero? n => error "No Y values"
  zLen : Integer := \#z
  zLen ~= (m*n) => error "Too many Z-values to fit grid"
  error "Not enough Z-values to fit grid"
  points : L L Pt := [
  for j in n..1 by -1 repeat
    row : L Pt := [
    for i in m..1 by -1 repeat
      zval := (j-1)*m+i
      row := cons(point([x.i,y.j,z.zval,z.zval]),row)
  points := cons(row,points)
  makeViewport3D(mesh points,l)
\end{verbatim}

— DRAWPT.dotabb —
package TOPSP TopLevelThreeSpace

--- TopLevelThreeSpace.input ---

)set break resume
)sys rm -f TopLevelThreeSpace.output
)spool TopLevelThreeSpace.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TopLevelThreeSpace
--R
--R TopLevelThreeSpace is a package constructor
--R Abbreviation for TopLevelThreeSpace is TOPSP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TOPSP
--R
--R---------------------------------------------------------- Operations -------------------------------
--R createThreeSpace : () -> ThreeSpace(DoubleFloat)
--R
--E 1

)spool
)lisp (bye)

---

--- TopLevelThreeSpace.help ---

====================================================================
TopLevelThreeSpace examples
====================================================================
This package exports a function for making a ThreeSpace.

See Also:
o )show TopLevelThreeSpace

---

TopLevelThreeSpace (TOPSP)

Exports:
createThreeSpace

— package TOPSP TopLevelThreeSpace —

)abbrev package TOPSP TopLevelThreeSpace
++ Description:
++ This package exports a function for making a \spadtype{ThreeSpace}

TopLevelThreeSpace(): with
  createThreeSpace: () -> ThreeSpace DoubleFloat
  ++ createThreeSpace() creates a \spadtype{ThreeSpace(DoubleFloat)} object
  ++ capable of holding point, curve, mesh components and any combination.
  ++
  + add
  createThreeSpace() == create3Space()$ThreeSpace(DoubleFloat)

— TOPSP.dotabb —

"TOPSP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=TOPSP"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
package INTHERTR TranscendentalHermiteIntegration

— TranscendentalHermiteIntegration.input —

)set break resume
)sys rm -f TranscendentalHermiteIntegration.output
)spool TranscendentalHermiteIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TranscendentalHermiteIntegration
--R TranscendentalHermiteIntegration(F: Field,UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for TranscendentalHermiteIntegration is INTHERTR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTHERTR
--R
--R-------------------------------- Operations --------------------------------
--R HermiteIntegrate : (Fraction(UP),(UP -> UP)) -> Record(answer: Fraction(UP),logpart: Fraction(UP),specpart: Fraction(UP),polypart: UP)
--E 1

)spool
)lisp (bye)

— TranscendentalHermiteIntegration.help —

====================================================================
TranscendentalHermiteIntegration examples
====================================================================

Hermite integration, transcendental case.

See Also:
o )show TranscendentalHermiteIntegration
TranscendentalHermiteIntegration (INHERTR)

Exports:
HermiteIntegrate

— package INHERTR TranscendentalHermiteIntegration —

)abbrev package INHERTR TranscendentalHermiteIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 12 August 1992
++ Description:
++ Hermite integration, transcendental case.

TranscendentalHermiteIntegration(F, UP): Exports == Implementation where
  F : Field
  UP : UnivariatePolynomialCategory F

  N ==> NonNegativeInteger
  RF ==> Fraction UP
  REC ==> Record(answer:RF, lognum:UP, logden:UP)
  HER ==> Record(answer:RF, logpart:RF, specpart:RF, polypart:UP)

Exports == with
  HermiteIntegrate: (RF, UP -> UP) -> HER
    ++ HermiteIntegrate(f, D) returns \spad{[g, h, s, p]}
    ++ such that \spad{f = Dg + h + s + p},
    ++ h has a squarefree denominator normal w.r.t. D,
    ++ and all the squarefree factors of the denominator of s are
    ++ special w.r.t. D. Furthermore, h and s have no polynomial parts.
    ++ D is the derivation to use on \spadtype{UP}.
Implementation ==> add
import MonomialExtensionTools(F, UP)

normalHermiteIntegrate: (RF,UP->UP) -> Record(answer:RF,lognum:UP,logden:UP)

HermiteIntegrate(f, derivation) ==
rec := decompose(f, derivation)
hi := normalHermiteIntegrate(rec.normal, derivation)
qr := divide(hi.lognum, hi.logden)
[hi.answer, qr.remainder / hi.logden, rec.special, qr.quotient + rec.poly]

-- Hermite Reduction on f, every squarefree factor of denom(f) is normal wrt D
-- this is really a "parallel" Hermite reduction, in the sense that
-- every multiple factor of the denominator gets reduced at each pass
-- so if the denominator is P1 P2**2 ... Pn**n, this requires O(n)
-- reduction steps instead of O(n**2), like Mack's algorithm
-- returns [g, b, d] s.t. f = g' + b/d and d is squarefree and normal wrt D
normalHermiteIntegrate(f, derivation) ==
a := numer f
q := denom f
p:UP := 0
mult:UP := 1
qhat := (q exquo (g0 := g := gcd(q, differentiate q)))::UP
while(degree(qbar := g) > 0) repeat
  qbarhat := (qbar exquo (g := gcd(qbar, differentiate qbar)))::UP
  qtil := - ((qhat * (derivation qbar)) exquo qbar)::UP
  bc :=
    extendedEuclidean(qtil, qbarhat, a)::Record(coef1:UP, coef2:UP)
  qr := divide(bc.coef1, qbarhat)
a := bc.coef2 + qtil * qr.quotient - derivation(qr.remainder)
  * (qhat exquo qbarhat)::UP
p := p + mult * qr.remainder
mult := mult * qbarhat
[p / g0, a, qhat]
package INTTR TranscendentalIntegration

-- TranscendentalIntegration.input --

)set break resume
)sys rm -f TranscendentalIntegration.output
)spool TranscendentalIntegration.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TranscendentalIntegration
--R
--R TranscendentalIntegration(F: Field,UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for TranscendentalIntegration is INTTR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for INTTR
--R
--R------------------------------- Operations --------------------------------
--R expextendedint : (Fraction(UP),(UP -> UP),((Integer,F) -> Record(ans: F,right: F,sol?: Boolean)),Fraction(UP)) -> Union(Record(answer: Fraction(UP),a0: F),Record(ratpart: Fraction(UP),coeff: Fraction(UP)),"failed")
--R expintegrate : (Fraction(UP),(UP -> UP),((Integer,F) -> Record(ans: F,right: F,sol?: Boolean))) -> Record(answer: IntegrationResult(Fraction(UP)),a0: F)
--R expintfldpoly : (LaurentPolynomial(F,UP),((Integer,F) -> Record(ans: F,right: F,sol?: Boolean))) -> Union(LaurentPolynomial(F,UP),"failed")
--R explimitedint : (Fraction(UP),(UP -> UP),((Integer,F) -> Record(ans: F,right: F,sol?: Boolean)),List(Fraction(UP))) ... Record(mainpart: Fraction(UP),limitedlogs: List(Record(coeff: Fraction(UP),logand: Fraction(UP)))),a0: F),"failed")
--R monomialIntPoly : (UP,(UP -> UP)) -> Record(answer: UP,polypart: UP)
--R monomialIntegrate : (Fraction(UP),(UP -> UP)) -> Record(ir: IntegrationResult(Fraction(UP)),
--R primextendedint : (Fraction(UP),(UP -> UP),((Integer,F,F) -> Union(List(F),"failed")))
--R primextintfrac : (Fraction(UP),(UP -> UP),Fraction(UP)) -> Union(Record(ratpart: Fraction(UP),coeff: Fraction(UP)),"failed")
--R primintegrate : (Fraction(UP),(UP -> UP),((Integer,F,F) -> Union(Record(ratpart: F,coeff: F),"failed")))
--R primintfldpoly : (UP,(F -> Union(Record(ratpart: F,coeff: F),"failed"))),F) -> Union(UP,"failed")
--R primlimintfrac : (Fraction(UP),(UP -> UP),List(Fraction(UP))) -> Union(Record(mainpart: Fraction(UP),
--R primlimitedint : (Fraction(UP),(UP -> UP),((Integer,F,F) -> Union(List(F),"failed"))) -> Record
--R
--E 1

)spool
)lisp (bye)


-- TranscendentalIntegration.help --

====================================================================
TranscendentalIntegration examples
====================================================================

This package provides functions for the transcendental case of the
Risch algorithm.

See Also:
o )show TranscendentalIntegration

TranscendentalIntegration (INTTR)

Exports:

expextendint   expintegrate
expintfldpoly  explimitedint
monomialIntPoly monomialIntegrate
primextendint  primextintfrac
primintegrate  primintlfdfrac
primlimintfrac primlimitedint
tanintegrate

— package INTTR TranscendentalIntegration —

)abbrev package INTTR TranscendentalIntegration
++ Author: Manuel Bronstein
++ Date Created: 1987
++ Date Last Updated: 24 October 1995
++ Description:
++ This package provides functions for the transcendental
++ case of the Risch algorithm.
-- Internally used by the integrator

TranscendentalIntegration(F, UP): Exports == Implementation where
  F : Field
  UP : UnivariatePolynomialCategory F
N ==> NonNegativeInteger
Z ==> Integer
Q ==> Fraction Z
GP ==> LaurentPolynomial(F, UP)
UP2 ==> SparseUnivariatePolynomial UP
RF ==> Fraction UP
UPR ==> SparseUnivariatePolynomial RF
IR ==> IntegrationResult RF
LOG ==> Record(scalar:Q, coeff:UPR, logand:UPR)
LLG ==> List Record(coeff:RF, logand:RF)
NE ==> Record(integrand:RF, intvar:RF)
NL ==> Record(mainpart:RF, limitedlogs:LLG)
UPF ==> Record(answer:UP, a0:F)
RFF ==> Record(answer:RF, a0:F)
IRF ==> Record(answer:IR, a0:F)
NLF ==> Record(answer:NL, a0:F)
GPF ==> Record(answer:GP, a0:F)
UPUP ==> Record(elem:UP, notelem:UP)
GPGP ==> Record(elem:GP, notelem:GP)
RFRF ==> Record(elem:RF, notelem:RF)
FF ==> Record(ratpart:F, coeff:F)
FFR ==> Record(ratpart:RF, coeff:RF)
UF ==> Union(FF, "failed")
UF2 ==> Union(List F, "failed")
REC ==> Record(ir:IR, specpart:RF, polypart:UP)
PSOL ==> Record(ans:F, right:F, sol?:Boolean)
FAIL ==> error "Sorry - cannot handle that integrand yet"

Exports ==> with
primintegrate : (RF, UP -> UP, F -> UF) -> IRF
++ primintegrate(f, ', foo) returns \spad{[g, a]} such that
++ \spad{f = g' + a}, and \spad{a} has no integral in UP.
++ Argument foo is an extended integration function on F.
expintegrate : (RF, UP -> UP, (Z, F) -> PSOL) -> IRF
++ expintegrate(f, ', foo) returns \spad{[g, a]} such that
++ \spad{f = g' + a}, and \spad{a} has no integral in F;
++ Argument foo is a Risch differential equation solver on F;
tanintegrate : (RF, UP -> UP, (Z, F, F) -> UF2) -> IRF
++ tanintegrate(f, ', foo) returns \spad{[g, a]} such that
++ \spad{f = g' + a}, and \spad{a} has no integral in F;
++ Argument foo is a Risch differential system solver on F;
primextendedint:(RF, UP -> UP, F->UF, RF) -> Union(RFF,FFR,"failed")
++ primextendedint(f, ', foo, g) returns either \spad{[v, c]} such that
++ \spad{f = v' + c g} and \spad{c'} = 0, or \spad{[v, a]} such that
++ \spad{f = g' + a}, and \spad{a} has no integral in UP.
++ Returns "failed" if neither case can hold.
++ Argument foo is an extended integration function on F.
expextendedint:(RF,UP->UP,(Z,F)->PSOL, RF) -> Union(RFF,FFR,"failed")
++ expextendedint(f, ', foo, g) returns either \spad{[v, c]} such that
++ \spad{f = v' + c g} and \spad{c'} = 0, or \spad{[v, a]} such that
++ \spad{f = g' + a}, and \spad{a} has no integral in UP.
++ \spad{f = g' + a}, and \spad{a = 0} or \spad{a} has no integral in \spad{F}.
++ Returns "failed" if neither case can hold.
++ Argument foo is a Risch differential equation function on \spad{F}.
primlimitedint:(RF, UP -> UP, F->UF, List RF) -> Union(NLF,"failed")
++ primlimitedint(f, ', foo, [u1,...,un]) returns
++ \spad{[v, [ci,...,cn], a]} such that \spad{ci' = 0},
++ \spad{f = v' + a + reduce(+,[ci * ui'/ui])},
++ and \spad{a = 0} or \spad{a} has no integral in \spad{UP}.
++ Returns "failed" if no such v, ci, a exist.
++ Argument foo is an extended integration function on \spad{F}.
explimitedint:(RF, UP->UP,(Z,F)->PSOL,List RF) -> Union(NLF,"failed")
++ explimitedint(f, ', foo, [u1,...,un]) returns
++ \spad{[v, [ci,...,cn], a]} such that \spad{ci' = 0},
++ \spad{f = v' + a + reduce(+,[ci * ui'/ui])},
++ and \spad{a = 0} or \spad{a} has no integral in \spad{F}.
++ Returns "failed" if no such v, ci, a exist.
++ Argument foo is a Risch differential equation function on \spad{F}.
primextintfrac : (RF, UP -> UP, RF) -> Union(FFR, "failed")
++ primextintfrac(f, ', g) returns \spad{[v, c]} such that
++ \spad{f = v' + c g} and \spad{c' = 0}.
++ Error: if \spad{degree numer f >= degree denom f} or
++ if \spad{degree numer g >= degree denom g} or
++ if \spad{denom g} is not squarefree.
primlimintfrac : (RF, UP -> UP, List RF) -> Union(NL, "failed")
++ primlimintfrac(f, ', [u1,...,un]) returns \spad{[v, [ci,...,cn]]}
++ such that \spad{ci' = 0} and \spad{f = v' + /[ci * ui'/ui]}.   
++ Error: if \spad{degree numer f >= degree denom f}.
primintfldpoly : (UP, F -> UF, F) -> Union(UP, "failed")
++ primintfldpoly(p, ', t') returns q such that \spad{p' = q} or
++ "failed" if no such q exists. Argument \spad{\spad{t'}} is the derivative of
++ the primitive generating the extension.
expintfldpoly : (GP, (Z, F) -> PSOL) -> Union(GP, "failed")
++ expintfldpoly(p, foo) returns q such that \spad{p' = q} or
++ "failed" if no such q exists.
++ Argument foo is a Risch differential equation function on \spad{F}.
monomialIntegrate : (RF, UP -> UP) -> REC
++ monomialIntegrate(f, ') returns \spad{[ir, s, p]} such that
++ \spad{f = ir' + s + p} and all the squarefree factors of the
++ denominator of s are special w.r.t the derivation '.
monomialIntPoly : (UP, UP -> UP) -> Record(answer:UP, polypart:UP)
++ monomialIntPoly(p, ') returns [q, r] such that
++ \spad{p = q' + r} and \spad{degree(r) < degree(t')}. 
++ Error if \spad{degree(t') < 2}.

Implementation ==> add
import SubResultantPackage(UP, UP2)
import MonomialExtensionTools(F, UP)
import TranscendentalHermiteIntegration(F, UP)
import CommuteUnivariatePolynomialCategory(F, UP, UP2)
primintegratepoly : (UP, F -> UF, F) -> Union(UPF, UPUP)
expintegratepoly : (GP, (Z, F) -> PSOL) -> Union(GPF, GPGP)
expextintfrac : (RF, UP -> UP, RF) -> Union(FFR, "failed")
explimintfrac : (RF, UP -> UP, List RF) -> Union(NL, "failed")
limitedLogs : (RF, RF -> RF, List RF) -> Union(LLG, "failed")
logprmderiv : (RF, UP -> UP) -> RF
logexpderiv : (RF, UP -> UP, F) -> RF
tanintegrateSpecial: (RF, RF -> RF, (Z, F, F) -> UF2) -> Union(RFF, RFRF)

UP2UP2 : UP -> UP2
UP2UPR : UP -> UPR
UP22UPR : UP2 -> UPR
notelementary : REC -> IR
kappa : (UP, UP -> UP) -> UP

dummy:RF := 0

logprmderiv(f, derivation) == differentiate(f, derivation) / f

UP2UP2 p ==
    map(x+->x::UP, p)$UnivariatePolynomialCategoryFunctions2(F, UP, UP, UP2)

UP2UPR p ==
    map(x+->x::UP::RF,p)$UnivariatePolynomialCategoryFunctions2(F,UP,RF,UPR)

UP22UPR p ==
    map(x+->x::RF, p)$SparseUnivariatePolynomialFunctions2(UP, RF)

-- given p in k[z] and a derivation on k[t] returns the coefficient lifting
-- in k[z] of the restriction of D to k.
kappa(p, derivation) ==
    ans:UP := 0
    while p ^= 0 repeat
        ans := ans + derivation(leadingCoefficient(p)::UP)*monomial(1,degree p)
        p := reductum p
    ans

-- works in any monomial extension
monomialIntegrate(f, derivation) ==
    zero? f => [0, 0, 0]
    r := HermiteIntegrate(f, derivation)
    zero?(inum := numer(r.logpart)) => [r.answer::IR, r.specpart, r.polypart]
    iden := denom(r.logpart)
    x := monomial(1, 1)$UP
    resultvec := subresultantVector(UP2UP2 inum -
        (x::UP2) * UP2UP2 derivation iden, UP2UP2 iden)
    respoly := primitivePart leadingCoefficient resultvec 0
    rec := splitSquarefree(respoly, x1 +-> kappa(x1, derivation))
    logs:List(LOG) := [1, UP2UPR(term.factor),
        UP22UPR swap primitivePart(resultvec(term.exponent),term.factor)]
for term in factors(rec.special)]

dlog :=
  -- one? derivation x => r.logpart
  ((derivation x) = 1) => r.logpart
  differentiate(mkAnswer(0, logs, empty()),
    (x1:RF):RF <-> differentiate(x1, derivation))
  (u := retractIfCan(p := r.logpart - dlog)@Union(UP, "failed")) case UP =>
    [mkAnswer(r.answer, logs, empty(), r.specpart, r.polypart + u::UP]
    [mkAnswer(r.answer, logs, [[p, dummy]], r.specpart, r.polypart]

-- returns [q, r] such that p = q' + r and degree(r) < degree(dt)
-- must have degree(derivation t) >= 2

monomialIntPoly(p, derivation) ==
  (d := degree(dt := derivation monomial(1,1))::Z) < 2 =>
    error "monomIntPoly: monomial must have degree 2 or more"
  l := leadingCoefficient dt
  ans:UP := 0
  while (n := 1 + degree(p)::Z - d) > 0 repeat
    ans := ans + (term := monomial(leadingCoefficient(p) / (n * l), n::N))
    p := p - derivation term -- degree(p) must drop here
    [ans, p]

-- returns either
-- (q in GP, a in F) st p = q' + a, and a=0 or a has no integral in F
-- or (q in GP, r in GP) st p = q' + r, and r has no integral elem/UP

expintegratepoly(p, FRDE) ==
  coef0:F := 0
  notelm := answr := 0$GP
  while p ^= 0 repeat
    ans1 := FRDE(n := degree p, a := leadingCoefficient p)
    answr := answr + monomial(ans1.ans, n)
    if ~ans1.sol? then -- Risch d.e. has no complete solution
      missing := a - ans1.right
      if zero? n then coef0 := missing
      else notelm := notelm + monomial(missing, n)
    p := reductum p
  zero? notelm => [answr, coef0]
  [answr, notelm]

-- f is either 0 or of the form p(t)/(1 + t**2)**n
-- returns either
-- (q in RF, a in F) st f = q' + a, and a=0 or a has no integral in F
-- or (q in RF, r in RF) st f = q' + r, and r has no integral elem/UP

tanintegratespecial(f, derivation, FRDE) ==
  ans:RF := 0
  p := monomial(1, 2)$UP + 1
  while (n := degree(denom f) quo 2) ^= 0 repeat
    r := numer(f) rem p
    a := coefficient(r, 1)
    b := coefficient(r, 0)
(u := FRDE(n, a, b)) case "failed" => return [ans, f]
l := u::List(F)
term:RF := (monomial(first l, 1)$UP + second(l)::UP) / denom f
ans := ans + term
f := f - derivation term -- the order of the pole at 1+t^2 drops
zero?(c0 := retract(retract(f)@UP)@F) or
(u := FRDE(0, c0, 0)) case "failed" => [ans, c0]
[ans + first(u::List(F))::UP::RF, 0::F]

-- returns (v in RF, c in RF) s.t. f = v' + cg, and c' = 0, or "failed"
-- g must have a squarefree denominator (always possible)
-- g must have no polynomial part and no pole above t = 0
-- f must have no polynomial part and no pole above t = 0
expextintfrac(f, derivation, g) ==
zero? f => [0, 0]
degree numer f >= degree denom f => error "Not a proper fraction"
order(denom f,monomial(1,1)) ^= 0 => error "Not integral at t = 0"
r := HermiteIntegrate(f, derivation)
zero? g =>
r.logpart ^= 0 => "failed"
[r.answer, 0]
degree numer g >= degree denom g => error "Not a proper fraction"
order(denom g,monomial(1,1)) ^= 0 => error "Not integral at t = 0"
differentiate(c := r.logpart / g, derivation) ^= 0 => "failed"
[r.answer, c]

limitedLogs(f, logderiv, lu) ==
zero? f => empty()
empty? lu => "failed"
empty? rest lu =>
logderiv(c0 := f / logderiv(u0 := first lu)) ^= 0 => "failed"
[[c0, u0]]
num := numer f
den := denom f
l1:List Record(logand2:RF, contrib:UP) :=
-- \{[u, numer v] for u in lu | one? denom(v := den * logderiv u)\}
-- \{[u, numer v] for u in lu | (denom(v := den * logderiv u) = 1)\}
rows := max(degree den,
1 + reduce(max, [degree(u.contrib) for u in l1], 0)$List(N))
m:Matrix(F) := zero(rows, cols := 1 + #l1)
for i in 0..rows-1 repeat
for pp in l1 for j in minColIndex m .. maxColIndex m - 1 repeat
qsetelt_!(m, i + minRowIndex m, j, coefficient(pp.contrib, i))
qsetelt_!(m,i+minRowIndex m, maxColIndex m, coefficient(num, i))
m := rowEchelon m
ans := empty()$LLG
for i in minRowIndex m .. maxRowIndex m |
qelt(m, i, maxColIndex m) ^= 0 repeat
OK := false
for pp in l1 for j in minColIndex m .. maxColIndex m - 1
while not OK repeat
  if qelt(m, i, j) ^= 0 then
    OK := true
    c := qelt(m, i, maxColIndex m) / qelt(m, i, j)
    logderiv(c0 := c::UP::RF) ^= 0 => return "failed"
    ans := concat([c0, pp.logand2], ans)
  not OK => return "failed"

-- returns q in UP s.t. p = q', or "failed"
primintfldpoly(p, extendedint, t') ==
  (u := primintegratepoly(p, extendedint, t')) case UPUP => "failed"
  u.a0 ^= 0 => "failed"
  u.answer

-- returns q in GP st p = q', or "failed"
expintfldpoly(p, FRDE) ==
  (u := expintegratepoly(p, FRDE)) case GPGP => "failed"
  u.a0 ^= 0 => "failed"
  u.answer

-- returns (v in RF, c1...cn in RF, a in F) s.t. ci' = 0,
-- and f = v' + a + +/
-- and a = 0 or a has no integral in UP
primlimitedint(f, derivation, extendedint, lu) ==
  qr := divide(numer f, denom f)
  (u1 := primlimintfrac(qr.remainder / (denom f), derivation, lu))
  case "failed" => "failed"
  (u2 := primintegratepoly(qr.quotient, extendedint, retract derivation monomial(1, 1))) case UPUP => "failed"
  [[[u1.mainpart + u2.answer::RF, u1.limitedlogs], u2.a0]

-- returns (v in RF, c1...cn in RF, a in F) s.t. ci' = 0,
-- and f = v' + a + +/
-- and a = 0 or a has no integral in F
explimitedint(f, derivation, FRDE, lu) ==
  qr := separate(f)$GP
  (u1 := explimintfrac(qr.fracPart,derivation, lu)) case "failed" => "failed"
  (u2 := expintegratepoly(qr.polyPart, FRDE)) case GPGP => "failed"
  [[u1.mainpart + convert(u2.answer)@RF, u1.limitedlogs], u2.a0]

-- returns [v, c1...cn] s.t. f = v' + +/[ci * ui'/ui]
-- f must have no polynomial part (degree numer f < degree denom f)
primlimintfrac(f, derivation, lu) ==
  zero? f => [0, empty()]
  degree numer f => degree denom f => error "Not a proper fraction"
  r := HermiteIntegrate(f, derivation)
  zero?(r.logpart) => [r.answer, empty()]
  (u := limitedLogs(r.logpart, x1 +-> logprmderv(x1, derivation), lu))
case "failed" => "failed"
[r.answer, u::LLG]

-- returns [v, c1...cn] s.t. f = v' + +/[ci * ui'/ui]
-- f must have no polynomial part (degree numer f < degree denom f)
-- f must be integral above t = 0
explimintfrac(f, derivation, lu) ==
zero? f => [0, empty()]
degree numer f >= degree denom f => error "Not a proper fraction"
order(denom f, monomial(1,1)) > 0 => error "Not integral at t = 0"
r := HermiteIntegrate(f, derivation)
zero?(r.logpart) => [r.answer, empty()]
eta' := coefficient(differentiation monomial(1, 1), 1)
(u := limitedLogs(r.logpart, x1 +-> logexpderiv(x1,derivation,eta'), lu))
case "failed" => "failed"
[r.answer - eta'::UP *
+/[((degree numer(v.logand))::Z - (degree denom(v.logand))::Z) *
  v.coeff for v in u], u::LLG]

logexpderiv(f, derivation, eta') ==
(differentiate(f, derivation) / f) -
  (((degree numer f)::Z - (degree denom f)::Z) * eta')::UP::RF

notelementary rec ==
rec.ir + integral(rec.polypart::RF + rec.specpart, monomial(1,1)$UP :: RF)

-- returns
primintegrate(f, derivation, extendedint) ==
rec := monomialIntegrate(f, derivation)
not elem?(i1 := rec.ir) => [notelementary rec, 0]
(u2 := primintegratepoly(rec.polypart, extendedint,
  retract derivation monomial(1, 1))) case UPUP =>
[i1 + u2.elem::RF::IR + integral(u2(notelem::RF, monomial(1,1)$UP :: RF), 0]
[i1 + u2.answer::RF::IR, u2.a0]

-- returns
expintegrate(f, derivation, FRDE) ==
rec := monomialIntegrate(f, derivation)
not elem?(i1 := rec.ir) => [notelementary rec, 0]
rec.specpart is either 0 or of the form p(t)/t**n
special := rec.polypart::GP +
(numer(rec.specpart)::GP exquo denom(rec.specpart)::GP)::GP
(u2 := expintegratepoly(special, FRDE)) case GP =>
[i1 + convert(u2.elem)@RF::IR + integral(convert(u2(notelem)@RF,
  monomial(1,1)$UP :: RF), 0]
[i1 + convert(u2.answer)@RF::IR, u2.a0]
-- returns
-- (g in IR, a in F) st f = g' + a, and a = 0 or a has no integral in F
integrate(f, derivation, FRDE) ==
  rec := monomialIntegrate(f, derivation)
  not elem?(i1 := rec.ir) => [notelementary rec, 0]
  r := monomialIntPoly(rec.polypart, derivation)
  t := monomial(1, 1)$UP
  c := coefficient(r.polypart, 1) / leadingCoefficient(derivation t)
  derivation(c::UP) ^= 0 =>
    [i1 + mkAnswer(r.answer::RF, empty(),
    [r.polypart::RF + rec.specpart, dummy]$NE)], 0)

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- or (v in RF, a in F) s.t. f = v' + a
-- and a = 0 or a has no integral in UP
primextendint(f, derivation, extendedint, g) ==
  fqr := divide(numer f, denom f)
  gqr := divide(numer g, denom g)
  (u1 := primextintfrac(fqr.remainder / (denom f), derivation,
  gqr.remainder / (denom g)) case "failed" => "failed"
    zero?(gqr.remainder) =>
      zero? c => empty()
      [[1, monomial(1,1)$UPR - (c/(2::F))::UP::RF::UPR, (1 + t**2)::RF::UPR]]
  c0 := coefficient(r.polypart, 0)
  (u := tanintegrateSpecial(rec.specpart, x+->differentiate(x, derivation),
  FRDE)) case RFRF =>
    [i1+mkAnswer(r.answer::RF + u.elem, logs, [[u.notelem, dummy]$NE]), c0]
  [i1 + mkAnswer(r.answer::RF + u.answer, logs, empty()), u.a0 + c0]

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- or (v in RF, a in F) s.t. f = v' + a
-- and a = 0 or a has no integral in F
extendint(f, derivation, FRDE, g) ==
  qf := separate(f)$GP
  qg := separate g
  (u1 := expextintfrac(qf.fracPart, derivation, qg.fracPart)) case "failed" => "failed"
  zero?(qg.fracPart) =>
    zero? c => empty()
    [[1, monomial(1,1)$UPR - (c/(2::F))::UP::RF::UPR, (1 + t**2)::RF::UPR]]
  c0 := coefficient(r.polypart, 0)
  (u2 := primintfldpoly(fqr.quotient - retract(u1.coeff)@UP *
  gqr.quotient, extendedint, retract derivation monomial(1, 1))) case UPUP => "failed"
  [u2::UP::RF + u1.ratpart, u1.coeff]

-- returns either (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- or (v in RF, a in F) s.t. f = v' + a
-- and a = 0 or a has no integral in F
extendint(f, derivation, FRDE, g) ==
  qf := separate(f)$GP
  qg := separate g
  (u1 := expextintfrac(qf.fracPart, derivation, qg.fracPart)) case "failed" => "failed"
  zero?(qg.fracPart) =>
    the following FAIL's cannot occur if the primitives are all logs
degree(gqr.quotient) > 0 => FAIL
  (u3 := primintegratepoly(fqr.quotient, extendedint,
  retract derivation monomial(1, 1))) case UPUP => "failed"
  [u1.ratpart + u3.answer::RF, u3.a0]
  (u2 := primintfldpoly(fqr.quotient - retract(u1.coeff)@UP *
  gqr.quotient, extendedint, retract derivation monomial(1, 1))) case "failed" => "failed"
  [u2::UP::RF + u1.ratpart, u1.coeff]
(u3 := expintegratepoly(qf.polyPart, FRDE)) case GPGP => "failed"
[u1.ratpart + convert(u3.answer)@RF, u3.a0]
(u2 := expintfldpoly(qf.polyPart - retract(u1.coeff)@UP :: GP
* qg.polyPart, FRDE)) case "failed" => "failed"
[convert(u2::GP)@RF + u1.ratpart, u1.coeff]
-- returns either
-- (q in UP, a in F) st p = q'+ a, and a=0 or a has no integral in UP
-- or (q in UP, r in UP) st p = q'+ r, and r has no integral elem/UP

primintegratepoly(p, extendedint, t') ==
  zero? p => [0, 0$F]
  ans:UP := 0
  while (d := degree p) > 0 repeat
    (ans1 := extendedint leadingCoefficient p) case "failed" =>
      return([ans, p])
    p := reductum p - monomial(d * t' * ans1.ratpart, (d - 1)::N)
    ans := ans + monomial(ans1.ratpart, d)
    + monomial(ans1.coeff / (d + 1)::F, d + 1)
  (ans1 := extendedint(rp := retract(p)@F)) case "failed" => [ans, rp]
  [monomial(ans1.coeff, 1) + ans1.ratpart::UP + ans, 0$F]

-- returns (v in RF, c in RF) s.t. f = v' + cg, and c' = 0
-- g must have a squarefree denominator (always possible)
-- g must have no polynomial part (degree numer g < degree denom g)
-- f must have no polynomial part (degree numer f < degree denom f)

primextintfrac(f, derivation, g) ==
  zero? f => [0, 0]
  degree numer f >= degree denom f => error "Not a proper fraction"
  r := HermiteIntegrate(f, derivation)
  zero? g =>
    r.logpart ^= 0 => "failed"
    [r.answer, 0]
  degree numer g >= degree denom g => error "Not a proper fraction"
  differentiate(c := r.logpart / g, derivation) ^= 0 => "failed"
  [r.answer, c]

----------

— INTTR.dotabb —

"INTTR" [color="#FF4488",href="bookvol10.4.pdf#nameddest=INTTR"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"INTTR" -> "PFECAT"
package TRMANIP TranscendentalManipulations

— TranscendentalManipulations.input —

)set break resume
)sys rm -f TranscendentalManipulations.output
)spool TranscendentalManipulations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TranscendentalManipulations

--R
--R TranscendentalManipulations(R: Join(OrderedSet,GcdDomain),F: Join(FunctionSpace(R),TranscendentalFunctionCategory)) is a package constructor
--R Abbreviation for TranscendentalManipulations is TRMANIP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TRMANIP
--R
--R------------------------------- Operations --------------------------------
--R cos2sec : F -> F
cosh2sech : F -> F
--R cot2tan : F -> F
cot2trig : F -> F
--R coth2tanh : F -> F
coth2trigh : F -> F
--R csc2sin : F -> F
csch2sinh : F -> F
--R expand : F -> F
expandLog : F -> F
--R expandPower : F -> F
htrigs : F -> F
--R removeCosSq : F -> F
removeCoshSq : F -> F
--R removeSinSq : F -> F
removeSinhSq : F -> F
--R sec2cos : F -> F
sech2cosh : F -> F
--R simplify : F -> F
simplifyExp : F -> F
--R simplifyLog : F -> F
sin2csc : F -> F
--R sinh2csch : F -> F
tan2cot : F -> F
--R tan2trig : F -> F
tanh2coth : F -> F
--R tan2trigh : F -> F
--R expandTrigProducts : F -> F if F has KONVERT(PATTERN(R)) and F has PATMAB(R) and R has KONVERT(PATTERN(R)) and R has PATMAB(R)
--R
--E 1

)spool
)lisp (bye)

— TranscendentalManipulations.help —

====================================================================
TranscendentalManipulations examples
====================================================================
TranscendentalManipulations provides functions to simplify and expand expressions involving transcendental operators.

See Also:
- show TranscendentalManipulations

---

TranscendentalManipulations (TRMANIP)

Exports:
- cos2sec
- cot2trig
- csc2sin
- expandLog
- removeCosSq
- removeSinhSq
- simplify
- sin2csc
- tan2trig
- expandTrigProducts

- cosh2sech
- coth2tanh
- csch2sinh
- expandPower
- removeCoshSq
- removeSinSq
- sech2cosh
- sinh2csch
- tanh2coth
- tanh2trigh

---

)abbrev package TRMANIP TranscendentalManipulations
++ Author: Bob Sutor, Manuel Bronstein
++ Date Created: Way back
++ Date Last Updated: 22 January 1996, added simplifyLog MCD.
++ Description:
++ TranscendentalManipulations provides functions to simplify and expand expressions involving transcendental operators.
TranscendentalManipulations(R, F): Exports == Implementation where
R : Join(OrderedSet, GcdDomain)
F : Join(FunctionSpace R, TranscendentalFunctionCategory)
Z ==> Integer
K ==> Kernel F
P ==> SparseMultivariatePolynomial(R, K)
UP ==> SparseUnivariatePolynomial P
POWER ==> "%power"::Symbol
POW ==> Record(val: F,exponent: Z)
PRODUCT ==> Record(coef : Z, var : K)
FPR ==> Fraction Polynomial R

Exports ==> with
expand : F -> F
++ expand(f) performs the following expansions on f:
++ \item 1. logs of products are expanded into sums of logs,
++ \item 2. trigonometric and hyperbolic trigonometric functions
++ of sums are expanded into sums of products of trigonometric
++ and hyperbolic trigonometric functions.
++ \item 3. formal powers of the form \spad{(a/b)**c} are expanded into
++ \spad{a**c * b**(-c)}.
++ \end{items}
simplify : F -> F
++ simplify(f) performs the following simplifications on f:
++ \item 1. rewrites trigs and hyperbolic trigs in terms
++ of \spad{\sin}, \spad{\cos}, \spad{\sinh}, \spad{\cosh}.
++ \item 2. rewrites \spad{\sin**2} and \spad{\sinh**2} in terms
++ of \spad{\cos} and \spad{\cosh},
++ \item 3. rewrites \spad{\exp(a)*\exp(b)} as \spad{\exp(a+b)}.
++ \item 4. rewrites \spad{\exp((a**(1/n))**m * (a**(1/s))**t} as a single
++ power of a single radical of \spad{\exp(a)}.
++ \end{items}
htrigs : F -> F
++ htrigs(f) converts all the exponentials in f into
++ hyperbolic sines and cosines.
simplifyExp: F -> F
++ simplifyExp(f) converts every product \spad{\exp(a)*\exp(b)}
++ appearing in f into \spad{\exp(a+b)}.
simplifyLog : F -> F
++ simplifyLog(f) converts every \spad{\log(a) - \log(b)} appearing in f
++ into \spad{\log(a/b)}, every \spad{\log(a) + \log(b)} into \spad{\log(a*b)}
++ and every \spad{a*log(a)} into \spad{\log(a^n)}.
expandPower: F -> F
++ expandPower(f) converts every power \spad{(a/b)**c} appearing
++ in f into \spad{a**c * b**(-c)}.
expandLog : F -> F
++ expandLog(f) converts every \spad{\log(a/b)} appearing in f into
++ \spad{\log(a) - \log(b)}, and every \spad{\log(a*b)} into
++ \spad{\log(a) + \log(b)}..
cos2sec : F -> F
++ cos2sec(f) converts every \spad{cos(u)} appearing in f into
++ \spad{1/sec(u)}.
cosh2sech : F -> F
++ cosh2sech(f) converts every \spad{cosh(u)} appearing in f into
++ \spad{1/sech(u)}.
cot2trig : F -> F
++ cot2trig(f) converts every \spad{cot(u)} appearing in f into
++ \spad{cos(u)/sin(u)}.
coth2trigh : F -> F
++ coth2trigh(f) converts every \spad{coth(u)} appearing in f into
++ \spad{cosh(u)/sinh(u)}.
csc2sin : F -> F
++ csc2sin(f) converts every \spad{csc(u)} appearing in f into
++ \spad{1/sin(u)}.
csch2sinh : F -> F
++ csch2sinh(f) converts every \spad{csch(u)} appearing in f into
++ \spad{1/sinh(u)}.
sec2cos : F -> F
++ sec2cos(f) converts every \spad{sec(u)} appearing in f into
++ \spad{1/cos(u)}.
sech2cosh : F -> F
++ sech2cosh(f) converts every \spad{sech(u)} appearing in f into
++ \spad{1/cosh(u)}.
sin2csc : F -> F
++ sin2csc(f) converts every \spad{sin(u)} appearing in f into
++ \spad{1/csc(u)}.
sinh2csch : F -> F
++ sinh2csch(f) converts every \spad{sinh(u)} appearing in f into
++ \spad{1/csch(u)}.
tan2trig : F -> F
++ tan2trig(f) converts every \spad{tan(u)} appearing in f into
++ \spad{sin(u)/cos(u)}.
tanh2trigh : F -> F
++ tanh2trigh(f) converts every \spad{tanh(u)} appearing in f into
++ \spad{sinh(u)/cosh(u)}.
tan2cot : F -> F
++ tan2cot(f) converts every \spad{tan(u)} appearing in f into
++ \spad{1/cot(u)}.
tanh2coth : F -> F
++ tanh2coth(f) converts every \spad{tanh(u)} appearing in f into
++ \spad{1/coth(u)}.
cot2tan : F -> F
++ cot2tan(f) converts every \spad{cot(u)} appearing in f into
++ \spad{1/tan(u)}.
coth2tanh : F -> F
++ coth2tanh(f) converts every \spad{coth(u)} appearing in f into
++ \spad{1/tanh(u)}.
removeCosSq: F -> F
++ removeCosSq(f) converts every \spad{cos(u)**2} appearing in f into
++ \spad{1 - sin(x)**2}, and also reduces higher
++ powers of \spad{\cos(u)} with that formula.
removeSinSq: F -> F
++ removeSinSq(f) converts every \spad{\sin(u)**2} appearing in f into
++ \spad{1 - \cos(x)**2}, and also reduces higher powers of
++ \spad{\sin(u)} with that formula.
removeCoshSq:F -> F
++ removeCoshSq(f) converts every \spad{\cosh(u)**2} appearing in f into
++ \spad{1 - \sinh(x)**2}, and also reduces higher powers of
++ \spad{\cosh(u)} with that formula.
removeSinhSq:F -> F
++ removeSinhSq(f) converts every \spad{\sinh(u)**2} appearing in f into
++ \spad{1 - \cosh(x)**2}, and also reduces higher powers
++ of \spad{\sinh(u)} with that formula.
if R has PatternMatchable(R) and R has ConvertibleTo(Pattern(R))
and F has ConvertibleTo(Pattern(R)) and F has PatternMatchable R then
expandTrigProducts : F -> F
++ expandTrigProducts(e) replaces \spad{\sin(x)*\sin(y)} by
++ \spad{(\cos(x-y)-\cos(x+y))/2}, \spad{\cos(x)*\cos(y)} by
++ \spad{(\cos(x-y)+\cos(x+y))/2}, and \spad{\sin(x)*\cos(y)} by
++ \spad{(\sin(x-y)+\sin(x+y))/2}. Note that this operation uses
++ the pattern matcher and so is relatively expensive. To avoid
++ getting into an infinite loop the transformations are applied
++ at most ten times.

Implementation ==> add
import FactoredFunctions(P)
import PolynomialCategoryLifting(IndexedExponents K, K, R, P, F)
import
PolynomialCategoryQuotientFunctions(IndexedExponents K,K,R,P,F)
th2th : F -> F
ch2th : F -> F
ch2sh : F -> F
sh2ch : F -> F
sh2ch2 : F -> F
simplify0 : F -> F
simplifyLog1 : F -> F
logArgs : List F -> F

import F
import List F

if R has PatternMatchable R and R has ConvertibleTo Pattern R
and F has ConvertibleTo(Pattern(R)) and F has PatternMatchable R then
    XX : F := coerce new()$Symbol
    YY : F := coerce new()$Symbol
    sinCosRule : RewriteRule(R,R,F) :=
        rule(cos(XX)*sin(YY),(sin(XX+YY)-sin(XX-YY))/2::F)
    sinSinRule : RewriteRule(R,R,F) :=
        rule(sin(XX)*sin(YY),(cos(XX-YY)-cos(XX+YY))/2::F)
    cosCosRule : RewriteRule(R,R,F) :=
        rule(cos(XX)*cos(YY),(cos(XX-YY)+cos(XX+YY))/2::F)
    sinhSum : RewriteRule(R,R,F) :=
        rule(sinh(XX+YY),(sinh(XX)*cosh(YY)+cosh(XX)*sinh(YY))::F)
    coshSum : RewriteRule(R,R,F) :=
        rule(cosh(XX+YY),(cosh(XX)*cosh(YY)+sinh(XX)*sinh(YY))::F)
    tanhSum : RewriteRule(R,R,F) :=
        rule(tanh(XX+YY),((tanh(XX)+tanh(YY))/(1+tanh(XX)*tanh(YY)))::F)
    cothSum : RewriteRule(R,R,F) :=
        rule(coth(XX+YY),((coth(XX)*coth(YY)+1)/(coth(YY)+coth(XX)))::F)
    sinhpsinh : RewriteRule(R,R,F) :=
        rule(sinh(XX)+sinh(YY),(2*sinh(1/2*(XX+YY))*cosh(1/2*(XX-YY)))::F)
    sinhmsinh : RewriteRule(R,R,F) :=
        rule(sinh(XX)-sinh(YY),(2*cosh(1/2*(XX+YY))*sinh(1/2*(XX-YY)))::F)
    coshpcosh : RewriteRule(R,R,F) :=
        rule(cosh(XX)+cosh(YY),(2*cosh(1/2*(XX+YY))*cosh(1/2*(XX-YY)))::F)
    coshmcsinh : RewriteRule(R,R,F) :=
        rule(cosh(XX)-cosh(YY),(2*sinh(1/2*(XX+YY))*sinh(1/2*(XX-YY)))::F)

expandTrigProducts(e:F):F ==
    applyRules([sinCosRule,sinSinRule,cosCosRule,
        sinhSum,coshSum,tanhSum,cothSum,
        sinhpsinh,sinhmsinh,coshpcosh,
        coshmcsinh],e,10)$ApplyRules(R,R,F)

logArgs(l:List F):F ==
    -- This function will take a list of Expressions (implicitly a sum) and
    -- add them up, combining log terms. It also replaces n*log(x) by
    -- log(x^n).
    import K
    sum : F := 0
arg : F := 1
for term in l repeat
  is?(term,"log"::Symbol) =>
    arg := arg * simplifyLog(first(argument(first(kernels(term))))))
-- Now look for multiples, including negative ones.
prod : Union(PRODUCT, "failed") := isMult(term)
(prod case PRODUCT and is?(prod.var,"log"::Symbol) =>
  arg := arg * simplifyLog ((first argument(prod.var))**(prod.coef))
sum := sum+term
sum+log(arg)

simplifyLog(e:F):F ==
simplifyLog1(numerator e)/simplifyLog1(denominator e)

simplifyLog1(e:F):F ==
  freeOf?(e,"log"::Symbol) => e
-- Check for n*log(u)
prod : Union(PRODUCT, "failed") := isMult(e)
(prod case PRODUCT and is?(prod.var,"log"::Symbol) =>
  log simplifyLog ((first argument(prod.var))**(prod.coef))

termList : Union(List(F),"failed") := isTimes(e)
-- I’m using two variables, termList and terms, to work round a
-- bug in the old compiler.
not (termList case "failed") =>
  -- We want to simplify each log term in the product and then multiply
  -- then together. However, if there is a constant or arithmetic
  -- expression (i.e. something which looks like a Polynomial) we would
  -- like to combine it with a log term.
terms :List F := [simplifyLog(term) for term in termList::List(F)]
exprs :List F := []
for i in 1..#terms repeat
  if retractIfCan(terms.i)@Union(FPR,"failed") case FPR then
    exprs := cons(terms.i,exprs)
terms := delete!(terms,i)
  if not empty? exprs then
    foundLog := false
    i : NonNegativeInteger := 0
    while (not(foundLog) and (i < #terms)) repeat
      i := i+1
      if is?(terms.i,"log"::Symbol) then
        args : List F := argument(retract(terms.i)@K)
        setelt(terms,i, log simplifyLog1(first(args)**(*exprs)))
        foundLog := true
-- The next line deals with a situation which shouldn't occur,
-- since we have checked whether we are freeOf log already.
if not foundLog then terms := append(exprs,terms)
*/terms
terms : Union(List(F),"failed") := isPlus(e)
not (terms case "failed") => logArgs(terms)

expt : Union(Pow,"failed") := isPower(e)
-- (expt case Pow) and not one? expt.exponent =>
not (expt case Pow) and not (expt.exponent = 1) =>
  simplifyLog(expt.val)**(expt.exponent)

kers : List K := kernels e
-- not(one?(#kers)) => e -- Have a constant
not(((#kers) = 1)) => e -- Have a constant
  kernel(operator first kers,[simplifyLog(u) for u in argument first kers])

if R has RetractableTo Integer then
  simplify x == rootProduct(simplify0 x)$AlgebraicManipulations(R,F)
else simplify x == simplify0 x

expandpow k ==
a := expandPower first(arg := argument k)
b := expandPower second arg
-- ne:F := (one? numer a => 1; numer(a)::F ** b)
ne:F := (((numer a) = 1) => 1; numer(a)::F ** b)
-- de:F := (one? denom a => 1; denom(a)::F ** (-b))
de:F := (((denom a) = 1) => 1; denom(a)::F ** (-b))
ne * de

termexp p ==
exponent:F := 0
coef := (leadingCoefficient p)::P
lpow := select((z:K):Boolean+->is?(z,POWER)$K, lk := variables p)$List(K)
for k in lk repeat
  d := degree(p, k)
  if is?(k, "exp"::Symbol) then
    exponent := exponent + d * first argument k
  else if not is?(k, POWER) then
    -- Expand arguments to functions as well ... MCD 23/1/97
    --coef := coef * monomial(1, k, d)
    coef := coef *
    monomial(1,
      kernel(operator k,
        [simplifyExp u for u in argument k], height k), d)
    coef::F * exp exponent * powersimp(p, lpow)

expandPower f ==
l : = select((z:K):Boolean +- is?(z, POWER)$K, kernels f)$List(K)
eval(f, l, [expandpow k for k in l])

-- l is a list of pure powers appearing as kernels in p
powersimp(p, l) ==
empty? l => 1
k := first l  -- k = a**b
a := first(arg := argument k)
exponent := degree(p, k) * second arg
empty?(lk := select((z:K):Boolean +-> a = first argument z, rest l)) =>
(a ** exponent) * powersimp(p, rest l)
for k0 in lk repeat
  exponent := exponent + degree(p, k0) * second argument k0
(a ** exponent) * powersimp(p, setDifference(rest l, lk))

ueval(x, s,f) == eval(x, s::Symbol, f)
ueval2(x,s,f) == eval(x, s::Symbol, 2, f)
cos2sec x == ueval(x, "cos", (z1:F):F +-> inv sec z1)
sin2csc x == ueval(x, "sin", (z1:F):F +-> inv csc z1)
csc2sin x == ueval(x, "csc", c2s)
sec2cos x == ueval(x, "sec", s2c)
tan2cot x == ueval(x, "tan", (z1:F):F +-> inv cot z1)
cot2tan x == ueval(x, "cot", (z1:F):F +-> inv tan z1)
tan2trig x == ueval(x, "tan", t2t)
cot2trig x == ueval(x, "cot", c2t)
cosh2sech x == ueval(x, "cosh", (z1:F):F +-> inv sech z1)
sinh2csch x == ueval(x, "sinh", (z1:F):F +-> inv csch z1)
csch2sinh x == ueval(x, "csch", ch2sh)
sech2cosh x == ueval(x, "sech", sh2ch)
tanh2coth x == ueval(x, "tanh", (z1:F):F +-> inv coth z1)
coth2tanh x == ueval(x, "coth", (z1:F):F +-> inv tanh z1)
tanh2trigh x == ueval(x, "tanh", th2th)
coth2trigh x == ueval(x, "coth", ch2th)
removeCosSq x == ueval2(x, "cos", (z1:F):F +-> 1 - (sin z1)**2)
removeSinSq x == ueval2(x, "sin", s2c2)
removeCoshSq x== ueval2(x, "cosh", (z1:F):F +-> 1 + (sinh z1)**2)
removeSinhSq x== ueval2(x, "sinh", sh2ch2)
expandLog x == smplog(numer x) / smplog(denom x)
simplifyExp x == (smpexp numer x) / (smpexp denom x)
expand x == (smpexpand numer x) / (smpexpand denom x)
smpexpand p == map(kerexpand, (r1:R):F +-> r1::F, p)
smplog p == map(logexpand, (r1:R):F +-> r1::F, p)
smp2htrigs p == map((k1:K):F +-> htrigs(k1::F), (r1:R):F +-> r1::F, p)
The htrigs function

The htrigs function can be used to replace and reduce hyperbolic trigonometric identities. The identity for \( \sinh(x) \) is \( (\exp(x) - \exp(-x))/2 \).

If we difference these we should get zero

\[
f := \sinh(x) - \frac{\exp(x) - \exp(-x)}{2}
\]

instead, by default, we get

\[
\frac{x - x}{2 \sinh(x) - e^x + e^{-x}}
\]

The function htrigs(\( f \)) gives 0.

This works as follows:

\[
m := \text{mainKernel } f \Rightarrow \sinh(x)
\]

which is coerced to the first part of the union:

\[
k := m::\text{Kernel(Expression(Integer))}
\]

and the operator is extracted:

\[
\text{op} := \text{operator(k)} \Rightarrow \sinh
\]

The argument function extracts the variable used as arguments:

\[
\text{argument k} \Rightarrow [x]
\]

At this point we have picked apart the main Kernel into its operator and its arguments. We now process the list of arguments.

The function htrigs is called on every element of the argument list, which in this case, returns a list:

\[
\text{arg} := [\text{htrigs } x \text{ for } x \text{ in } \text{argument } k]\$\text{List(Expression(Integer))} \\
=> [x]
\]

Type: List(Expression(Integer))
We form a polynomial by replacing the kernel in the numerator with ?

\[
\text{num} := \text{univariate(numer f, k)}
\]

\[
2x - e^{-x} + e^x
\]

Type: \(\text{SparseUnivariatePolynomial}\)

\(\text{SparseMultivariatePolynomial}\)

\(\text{Integer}, \text{Kernel(Expression(Integer)))}\)

and a polynomial of the denominator, replacing the kernel

\[
\text{den} := \text{univariate(denom f, k)}
\]

\[
2
\]

Type: \(\text{SparseUnivariatePolynomial}\)

\(\text{SparseMultivariatePolynomial}\)

\(\text{Integer}, \text{Kernel(Expression(Integer)))}\)

In this case the op is not the exponential so we are doing straight trig substitution. We reconstruct the function call using the op and arg values, that is:

\[
\text{g1} := \text{op arg} \Rightarrow \sinh(x)
\]

Type: \(\text{Expression(Integer)}\)

So sup2htrigs, which is a local function, is used to simplify the parts of the fraction. In this case,

\[
\text{sup2htrigs(num, g1 := op arg) \Rightarrow 0}
\]

Type: \(\text{Expression(Integer)}\)

\[
\text{sup2htrigs(den, g1) \Rightarrow 2}
\]

Type: \(\text{Expression(Integer)}\)

Thus, the result is 0

The identity for \(\cosh(x)\) is \((e^x + e^{-x})/2\)

If we difference these we should get zero

\[
f := \cosh(x) - (e^x + e^{-x})/2
\]

instead, by default, we get

\[
x - e^{-x} + e^x + 2\cosh(x)
\]

\[
------------------------
2
\]

and the function call \(htrigs(f)\) gives 0

This works as follows:
which is coerced to the first part of the union:

\[
x
\]

\[
k:=m::\text{Kernel(Expression(Integer))} => \%e
\]

and the operator is extracted:

\[
op:=\text{operator}(k) => \exp
\]

The argument function extracts the variable used as arguments:

\[
\text{argument } k ==> [x]
\]

At this point we have picked apart the main Kernel into its operator and its arguments. We now process the list of arguments.

The \text{htrigs} function is called on every element of the argument list, which in this case, returns a list:

\[
\text{arg}:=\{\text{htrigs } x \text{ for } x \text{ in argument } k\}\]$List(Expression(Integer))

We form polynomial by replacing the kernel in the numerator with ?

\[
\text{num} := \text{univariate(numer } f, k)
\]

\[
- x
- ? - \%e + 2\cosh(x)
\]

and a polynomial of the denominator, replacing the kernel

\[
\text{den} := \text{univariate(denom } f, k)
\]

\[
2
\]
In this case, the expression

\[ \text{is?(op, "exp":Symbol) => true} \]

so we form

\[ a := \text{first arg => x} \]
\[ \text{Type: Expression(Integer)} \]

since we know that

\[ \cosh(x)+\sinh(x) => \%e \]

we can form this use this expression in substitutions

\[ g1 := \cosh(a)+\sinh(a) => \sinh(x)+\cosh(x) \]
\[ \text{Type: Expression(Integer)} \]

since we know that

\[ -x \]
\[ \cosh(x)-\sinh(x) => -\%e \]

we can form this use this expression in substitutions

\[ g2 := \cosh(a)-\sinh(a) => -\sinh(x)+\cosh(x) \]
\[ \text{Type: Expression(Integer)} \]

\[ b := (\text{degree num}):Integer \text{ quo 2} => 0 \]
\[ \text{Type: NonNegativeInteger} \]

The supexp function is using the \( g1 \) and \( g2 \) identities to replace \( \exp(x) \)

\[ \text{supexp(num,g1,g2,b)} => \sinh(x)-\cosh(x)-\sinh(x)+2\cosh(x)-\cosh(x) \]
\[ \text{Type: Expression(Integer)} \]

\[ \text{supexp(den,g1,g2,b)} => 2 \]
\[ \text{Type: Expression(Integer)} \]

which is effectively

\[ \frac{t1}{t2} => \frac{(\sinh(x)-\cosh(x)-\sinh(x)+2\cosh(x)-\cosh(x))}{2} \]
\[ \text{Type: Expression(Integer)} \]

the last form of which can be rearranged as:

\[ \frac{(\sinh(x)-\sinh(x) + 2\cosh(x)-\cosh(x)-\cosh(x))}{2} => 0 \]
so the result is 0

— package TRMANIP TranscendentalManipulations —

htrigs f ==
  (m := mainKernel f) case "failed" => f
  op := operator(k := m::K)
  arg := [htrigs x for x in argument k]$List(F)
  num := univariate(numer f, k)
  den := univariate(denom f, k)
  is?(op, "exp"::Symbol) =>
    g1 := cosh(a := first arg) + sinh(a)
    g2 := cosh(a) - sinh(a)
    supexp(num,g1,g2,b:= (degree num)::Z quo 2)/supexp(den,g1,g2,b)
  sup2htrigs(num, g1:= op arg) / sup2htrigs(den, g1)

supexp(p, f1, f2, bse) ==
  ans:F := 0
  while p ^= 0 repeat
  g := htrigs(leadingCoefficient(p)::F)
  if ((d := degree(p)::Z - bse) >= 0) then
    ans := ans + g * f1 ** d
  else ans := ans + g * f2 ** (-d)
  p := reductum p
  ans

sup2htrigs(p, f) ==
  (map(smp2htrigs, p)$SparseUnivariatePolynomialFunctions2(P, F)) f

exlog p == +/\[r.coef * log(r.logand::F) for r in log squareFree p]

logexpand k ==
  nullary?(op := operator k) => k::F
  is?(op, "log"::Symbol) =>
    exlog(numer(x := expandLog first argument k)) - exlog denom x
    op [expandLog x for x in argument k]$List(F)

kerexpand k ==
  nullary?(op := operator k) => k::F
  is?(op, POWER) => expandpow k
  arg := first argument k
  is?(op, "sec"::Symbol) => inv expand cos arg
  is?(op, "csc"::Symbol) => inv expand sin arg
  is?(op, "log"::Symbol) =>
    exlog(numer(x := expand arg)) - exlog denom x
  num := numer arg
  den := denom arg
  (b := (reductum num) / den) ^= 0 =>
    a := (leadingMonomial num) / den
is?(op, "exp"::Symbol) => \(\exp(\exp a) \times \exp(\exp b)\)
is?(op, "sin"::Symbol) => 
\(\sin(\exp a) \times \exp(\cos b) + \cos(\exp a) \times \exp(\sin b)\)
is?(op, "cos"::Symbol) => 
\(\cos(\exp a) \times \exp(\cos b) - \sin(\exp a) \times \exp(\sin b)\)
is?(op, "tan"::Symbol) => 
\(\tan := \tan(\exp a)\)
\(\tanb := \exp(\cot b)\)
\(\frac{\tan + \tanb}{1 - \tan \cdot \tanb}\)
is?(op, "cot"::Symbol) => 
\(\cot := \cot(\exp a)\)
\(\cotb := \exp(\tan b)\)
\(\frac{\cot \cdot \cotb - 1}{\cotb + \cot}\)

\[\text{op} \text{ [expand x for x in argument k]} \text{List(F)}\]

\[\text{op} \text{ [expand x for x in argument k]} \text{List(F)}\]

\[
\text{smpexp p} = \\
\text{ans:F} := 0 \\
\text{while p} ^= 0 \text{ repeat} \\
\text{ans} := \text{ans} + \text{termexp leadingMonomial p} \\
\text{p} := \text{reductum} \text{ p} \\
\text{ans}
\]

\(--\text{this now works in 3 passes over the expression:}\)
\(--\text{pass1 rewrites trigs and htrigs in terms of sin, cos, sinh, cosh}\)
\(--\text{pass2 rewrites sin**2 and sinh**2 in terms of cos and cosh.}\)
\(--\text{pass3 groups exponentials together}\)
\[
\text{simplify0 x} = \\
\text{simplifyExp eval(eval(x,} \\
["\text{tan"::Symbol,"cot"::Symbol,"sec"::Symbol,"csc"::Symbol,} \\
"\text{tanh"::Symbol,"coth"::Symbol,"sech"::Symbol,"csch"::Symbol},} \\
[t2t,c2t,s2c,c2s,th2th,ch2th,sh2ch,ch2sh}],} \\
["\text{sin"::Symbol,"sinh"::Symbol}, [2,2], [s2c2, sh2ch2])
\]

— TRMANIP.dotabb —
package RDETR TranscendentalRischDE

— TranscendentalRischDE.input —

)set break resume
)sys rm -f TranscendentalRischDE.output
)spool TranscendentalRischDE.output
)set message test on
)set message auto off
)clear all

-- 1 of 1
)show TranscendentalRischDE
--R
--R TranscendentalRischDE(F: Join(Field,CharacteristicZero,RetractableTo(Integer)),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for TranscendentalRischDE is RDETR
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RDETR
--R
--R------------------------------- Operations --------------------------------
--R baseRDE : (Fraction(UP),Fraction(UP)) -> Record(ans: Fraction(UP),nosol: Boolean)
--R monomRDE : (Fraction(UP),Fraction(UP),(UP -> UP)) -> Union(Record(a: UP,b: Fraction(UP),c: Fraction(UP),t: UP),"failed")

--E 1

)spool
)lisp (bye)

——

— TranscendentalRischDE.help —

====================================================================
TranscendentalRischDE examples
====================================================================

Risch differential equation, transcendental case.

See Also:
  o )show TranscendentalRischDE

——
TranscendentalRischDE (RDETR)

Exports:

package RDETR TranscendentalRischDE

TranscendentalRischDE(F, UP): Exports == Implementation where

F : Join(Field, CharacteristicZero, RetractableTo Integer)
UP : UnivariatePolynomialCategory F

N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
REC ==> Record(a:UP, b:UP, c:UP, t:UP)
SPE ==> Record(b:UP, c:UP, m:Z, alpha:UP, beta:UP)
PSOL ==> Record(ans:UP, nosol:Boolean)
ANS ==> Union(ans:PSOL, eq:SPE)
PSQ ==> Record(ans:RF, nosol:Boolean)

Exports == with

monomRDE: (RF,RF,UP->UP) -> Union(Record(a:UP,b:RF,c:RF,t:UP), "failed")
"monomRDE(f,g,D) returns \spad{\{A, B, C, T\}} such that \spad{\{y' + f y = g\}} has a solution if and only if \spad{\{y = Q / T\}}, where Q satisfies \spad{\{A Q' + B Q = C\}} and has no normal pole. A and T are polynomials and B and C have no normal poles. D is the derivation to use.

baseRDE : (RF, RF) -> PSQ
"baseRDE(f, g) returns a \spad{\{y, b\}} such that \spad{\{y' + fy = g\}} if \spad{\{b = true\}}, y is a partial solution otherwise (no solution
++ in that case).
++ D is the derivation to use.

\begin{verbatim}
polyRDE : (UP, UP, Z, UP -> UP) -> ANS
++ polyRDE(a, B, C, n, D) returns either:
++ 1. \spad{[Q, b]} such that \spad{degree(Q) <= n} and
++ \spad{a Q' + B Q = C} if \spad{b = true}, Q is a partial solution
++ otherwise.
++ 2. \spad{[B1, C1, m, \alpha, \beta]} such that any polynomial solution
++ of degree at most n of \spad{A Q' + B Q = C} must be of the form
++ \spad{Q = \alpha H + \beta} where \spad{degree(H) <= m} and
++ H satisfies \spad{H' + B1 H = C1}.
++ D is the derivation to use.

Implementation ==> add
import MonomialExtensionTools(F, UP)

getBound : (UP, UP, Z) -> Z

SPDEnocancel1: (UP, UP, Z, UP -> UP) -> PSOL

SPDEnocancel2: (UP, UP, Z, Z, F, UP -> UP) -> ANS

SPDE : (UP, UP, UP, Z, UP -> UP) -> Union(SPE, "failed")
-- cancellation at infinity is possible, A is assumed nonzero
-- needs tagged union because of branch choice problem
-- always returns a PSOL in the base case (never a SPE)

polyRDE(aa, bb, cc, d, derivation) ==

n:Z
(u := SPDE(aa, bb, cc, d, derivation)) case "failed" => [[0, true]]

zero?(u.c) => [[u.beta, false]]

baseCase? := one?(dt := derivation monomial(1, 1))

n := degree(dt)::Z - 1

b0? := zero?(u.b)

b0? and (baseCase? or degree(u.b) > max(0, n)) =>

answ := SPDEnocancel1(u.b, u.c, u.m, derivation)
[[u.alpha * answ.ans + u.beta, answ.nosol]]

(n > 0) and (b0? or degree(u.b) < n) =>

uansw := SPDEnocancel2(u.b,u.c,u.m,n,leadingCoefficient dt,derivation)

uansw case ans=> [[u.alpha * uansw.ans.ans + u.beta, uansw.ans.nosol]]

[uansw.eq.b, uansw.eq.c, uansw.eq.m,
  u.alpha * uansw.eq.alpha, u.alpha * uansw.eq.beta + u.beta]]

b0? and baseCase? =>

degree(u.c) >= u.m => [[0, true]]

[[u.alpha * integrate(u.c) + u.beta, false]]
[u::SPE]

-- cancellation at infinity is possible, A is assumed nonzero
-- if u.b = 0 then u.a = 1 already, but no degree check is done
-- returns "failed" if a p' + b p = c has no soln of degree at most d,
-- otherwise [B, C, m, \alpha, \beta] such that any soln p of degree at
-- most d of a p' + b p = c must be of the form p = \alpha h + \beta,
-- where h' + B h = C and h has degree at most m
SPDE(aa, bb, cc, d, derivation) ==
zero? cc => [0, 0, 0, 0, 0]
d < 0 => "failed"
(u := cc exquo (g := gcd(aa, bb))) case "failed" => "failed"
aa := (aa exquo g)::UP
bb := (bb exquo g)::UP
cc := u::UP
(ra := retractIfCan(aa)@Union(F, "failed")) case F =>
a1 := inv(ra::F)
[a1 * bb, a1 * cc, d, 1, 0]
bc := extendedEuclidean(bb, aa, cc)::Record(coef1:UP, coef2:UP)
qr := divide(bc.coef1, aa)
r := qr.remainder -- z = bc.coef2 + b * qr.quotient
(v := SPDE(aa, bb + derivation aa,
    bc.coef2 + bb * qr.quotient - derivation r,
    d - degree(aa)::Z, derivation)) case "failed" => "failed"
[v.b, v.c, v.m, aa * v.alpha, aa * v.beta + r]

-- solves q' + b q = c with deg(q) <= d
-- case (B <> 0) and (D = d/dt or degree(B) > max(0, degree(Dt) - 1))
-- this implies no cancellation at infinity, BQ term dominates
-- returns [Q, flag] such that Q is a solution if flag is false,
-- a partial solution otherwise.
SPDEnocancel1(bb, cc, d, derivation) ==
q:UP := 0
db := (degree bb)::Z
lb := leadingCoefficient bb
while cc ^= 0 repeat
    d < 0 or (n := (degree cc)::Z - db) < 0 or n > d => return [q, true]
r := monomial((leadingCoefficient cc) / lb, n::N)
cc := cc - bb * r - derivation r
d := n - 1
q := q + r
[q, false]

-- case (t is a nonlinear monomial) and (B = 0 or degree(B) < degree(Dt) - 1)
-- this implies no cancellation at infinity, DQ term dominates or degree(Q) = 0
-- dtm1 = degree(Dt) - 1
SPDEnocancel2(bb, cc, d, dtm1, lt, derivation) ==
q:UP := 0
while cc ^= 0 repeat
    d < 0 or (n := (degree cc)::Z - dtm1) < 0 or n > d => return [[q, true]]
if n > 0 then
    r := monomial((leadingCoefficient cc) / (n * lt), n::N)
    cc := cc - bb * r - derivation r
d := n - 1
    q := q + r
else
    -- n = 0 so solution must have degree 0
    db:N := (zero? bb => 0; degree bb);
db ^= degree(cc) => return [[q, true]]
zero? db => return [[bb, cc, 0, 1, q]]
r := leadingCoefficient(cc) / leadingCoefficient(bb)
cc := cc - r * bb - derivation(r::UP)
d := - 1
q := q + r::UP
[[q, false]]

monomRDE(f, g, derivation) ==
gg := gcd(d := normalDenom(f,derivation), e := normalDenom(g,derivation))
tt := (gcd(e, differentiate e) exquo gcd(gg,differentiate gg))::UP
(u := ((tt * (aa := d * tt)) exquo e)) case "failed" => "failed"
[aa, aa * f - (d * derivation tt)::RF, u::UP * e * g, tt]

-- solve y' + f y = g for y in RF
-- assumes that f is weakly normalized (no finite cancellation)
-- base case: F' = 0
baseRDE(f, g) ==
(u := monomRDE(f, g, differentiate)) case "failed" => [0, true]
n := getBound(u.a,bb := retract(u.b)@UP,degree(cc := retract(u.c)@UP)::Z)
v := polyRDE(u.a, bb, cc, n, differentiate).ans
[v.ans / u.t, v.nosol]

-- return an a bound on the degree of a solution of A P' + B P = C, A ^= 0
-- cancellation at infinity is possible
-- base case: F' = 0
getBound(a, b, dc) ==
da := (degree a)::Z
zero? b => max(0, dc - da + 1)
db := (degree b)::Z
da > (db + 1) => max(0, dc - da + 1)
da < (db + 1) => dc - db
(n := retractIfCan(- leadingCoefficient(b) / leadingCoefficient(a)
)@Union(Z, "failed")) case Z => max(n::Z, dc - db)
dc - db
package RDETRS TranscendentalRischDESystem

— TranscendentalRischDESystem.input —

)set break resume
)sys rm -f TranscendentalRischDESystem.output
)spool TranscendentalRischDESystem.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TranscendentalRischDESystem
--R
--R TranscendentalRischDESystem(F: Join(Field,CharacteristicZero,RetractableTo(Integer)),UP: UnivariatePolynomialCategory(F)) is a package constructor
--R Abbreviation for TranscendentalRischDESystem is RDETRS
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for RDETRS
--R
--R-------------------------------- Operations --------------------------------
--R baseRDEsys : (Fraction(UP),Fraction(UP),Fraction(UP)) -> Union(List(Fraction(UP)),"failed")
--R monomRDEsys: (Fraction(UP),Fraction(UP),Fraction(UP),(UP -> UP)) -> Union(Record(a: UP,b: Fraction(UP))
--R
--E 1

)spool
)lisp (bye)

-----

— TranscendentalRischDESystem.help —

====================================================================
TranscendentalRischDESystem examples
====================================================================

Risch differential equation system, transcendental case.

See Also:
o )show TranscendentalRischDESystem

-----
TranscendentalRischDESystem (RDETRS)

Exports:

baseRDEsys  monomRDEsys

--- package RDETRS TranscendentalRischDESystem ---

\texttt{TranscendentalRischDESystem}(F, \texttt{UP}):: \texttt{Exports} == \texttt{Implementation} where
\begin{verbatim}
F : Join(Field, CharacteristicZero, RetractableTo Integer)
\end{verbatim}
\begin{verbatim}
UP : UnivariatePolynomialCategory F
\end{verbatim}
\begin{verbatim}
N ==> NonNegativeInteger
Z ==> Integer
RF ==> Fraction UP
V ==> Vector UP
U ==> Union(List UP, "failed")
REC ==> Record(z1:UP, z2:UP, r1:UP, r2:UP)
\end{verbatim}

Exports == with

\begin{verbatim}
monomRDEsys: (RF, RF, RF, UP -> UP) -> _
\end{verbatim}
\begin{verbatim}
   \texttt{Union(Record(a:UP, b:RF, h:UP, c1:RF, c2:RF, t:UP),"failed")}
\end{verbatim}
\begin{verbatim}
   \texttt{++ \texttt{monomRDEsys}(f,g1,g2,D) returns \texttt{\spad{[A, B, H, C1, C2, T]}} such that}
\end{verbatim}
\begin{verbatim}
   \texttt{++ \texttt{\spad{((y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2)}}} has a solution
\end{verbatim}
\begin{verbatim}
   \texttt{++ if and only if \texttt{\spad{\texttt{(y1 = Q1 / T, y2 = Q2 / T)}},}}
\end{verbatim}
\begin{verbatim}
   \texttt{++ where \texttt{\spad{\texttt{(B,C1,C2,Q1,Q2)}}} have no normal poles and satisfy}
\end{verbatim}
\begin{verbatim}
   \texttt{++ \texttt{\spad{\texttt{(Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2))}}}
\end{verbatim}
\begin{verbatim}
   \texttt{++ D is the derivation to use.}
\end{verbatim}
\begin{verbatim}
baseRDEsys: (RF, RF, RF) -> Union(List RF, "failed")
\end{verbatim}
\begin{verbatim}
   \texttt{++ \texttt{baseRDEsys}(f, g1, g2) returns fractions \texttt{\spad{y_1 \cdot y_2}} such that}
\end{verbatim}
\begin{verbatim}
   \texttt{++ \texttt{\spad{\texttt{\textbf{(y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2)}}}}}
\end{verbatim}
++ if \spad{y_1,y_2} exist, "failed" otherwise.

Implementation ==> add
import MonomialExtensionTools(F, UP)
import SmithNormalForm(UP, V, V, Matrix UP)

diophant: (UP, UP, UP, UP, UP) -> Union(REC, "failed")
getBound: (UP, UP, UP, UP, UP) -> Z
SPDEsys : (UP, UP, UP, UP, Z, UP -> UP, (F, F, UP, UP, UP, Z) -> U) -> U
DSPDEsys: (F, UP, UP, UP, Z, UP -> UP) -> U
DSPDEmix: (UP, UP, F, F, N, Z, UP -> UP) -> U
DSPDEbdom: (UP, UP, F, F, N, Z) -> U
DSPDEsys0: (F, UP, UP, UP, UP, F, F, Z, UP -> UP, (UP,UP,F,F,N) -> U) -> U

-- reduces (y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2) to
-- A (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2); Q1 = y1 T, Q2 = y2 T
-- where A and H are polynomials, and B,C1,C2,Q1 and Q2 have no normal poles.
-- assumes that f is weakly normalized (no finite cancellation)
monomRDEsys(f, g1, g2, derivation) ==
gg := gcd(d := normalDenom(f, derivation),
e := lcm(normalDenom(g1,derivation),normalDenom(g2,derivation)))
tt := (gcd(e, differentiate e) exquo gcd(gg,differentiate gg))::UP
(u := ((tt * (aa := d * tt)) exquo e)) case "failed" => "failed"
[aa, tt * d * f, - d * derivation tt, u::UP * e * g1, u::UP * e * g2, tt]

-- solve (y1', y2') + ((0, -f), (f, 0)) (y1,y2) = (g1,g2) for y1,y2 in RF
-- assumes that f is weakly normalized (no finite cancellation) and nonzero
-- base case: F' = 0
baseRDEsys(f, g1, g2) ==
zero? f => error "baseRDEsys: f must be nonzero"
zero? g1 and zero? g2 => [0, 0]
(u := monomRDEsys(f, g1, g2, differentiate)) case "failed" => "failed"
n := getBound(u.a, bb := retract(u.b), u.h,
cc1 := retract(u.c1), cc2 := retract(u.c2))
(v := SPDEsys(u.a, bb, u.h, cc1, cc2, n, differentiate,
(z1,z2,z3,z4,z5,z6) +->
DSPDEsys(z1, z2::UP, z3::UP, z4, z5, z6, differentiate)))
case "failed" => "failed"
l := v::List(UP)
[first(l) / u.t, second(l) / u.t]

-- solve
-- D1 = A Z1 + B R1 - C R2
-- D2 = A Z2 + C R1 + B R2
-- i.e. \((D1,D2) = ((A, 0, B, -C), (0, A, C, B))\) \((Z1, Z2, R1, R2)\)
-- for R1, R2 with degree(Ri) < degree(A)
-- assumes (A,B,C) = (1) and A and C are nonzero
diophant(a, b, c, d1, d2) ==
(u := diophantineSystem(matrix [[a,0,b,-c], [0,a,c,b]],
vector [d1,d2]).particular) case "failed" => "failed"

v := u::V
qr1 := divide(v 3, a)
qr2 := divide(v 4, a)
[v.1 + b * qr1.quotient - c * qr2.quotient,
v.2 + c * qr1.quotient + b * qr2.quotient, qr1.remainder, qr2.remainder]

-- solve
-- A (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2)
-- for polynomials Q1 and Q2 with degree <= n
-- A and B are nonzero
-- cancellation at infinity is possible
SPDEsys(a, b, h, c1, c2, n, derivation, degradation) ==
zero? c1 and zero? c2 => [0, 0]
n < 0 => "failed"
g := gcd(a, gcd(b, h))
((u1 := c1 exquo g) case "failed") or
((u2 := c2 exquo g) case "failed") => "failed"
a := (a exquo g)::UP
b := (b exquo g)::UP
h := (h exquo g)::UP
c1 := u1::UP
c2 := u2::UP
da := degree a) > 0 =>
(u := diophant(a, h, b, c1, c2)) case "failed" => "failed"
rec := u::REC
v := SPDEsys(a, b, h + derivation a, rec.z1 - derivation(rec.r1),
rec.z2 - derivation(rec.r2),n-da::Z,derivation,degradation)
v case "failed" => "failed"
l := v::List(UP)
[a * first(l) + rec.r1, a * second(l) + rec.r2]
ra := retract(a)@F
((rb := retractIfCan(b)@Union(F, "failed")) case "failed") or
((rh := retractIfCan(h)@Union(F, "failed")) case "failed") =>
DSPDEsys(ra, b, h, c1, c2, n, derivation)
degradation(ra, rb::F, rh::F, c1, c2, n)

-- solve
-- a (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2)
-- for polynomials Q1 and Q2 with degree <= n
-- A and B are nonzero, either B or H has positive degree
-- cancellation at infinity is not possible
DSPDEsys(a, b, h, c1, c2, n, derivation) ==
bb := degree(b)::Z
hh:Z :=
zero? h => 0
degree(h)::Z
lb := leadingCoefficient b
lh := leadingCoefficient h
bb < hh =>
DSPDEsys0(a, b, h, c1, c2, lb, lh, n, derivation, 
  (z1,z2,z3,z4,z5) +-> DSPDEhdom(z1,z2,z3,z4,z5,hh))
bb > hh =>
DSPDEsys0(a, b, h, c1, c2, lb, lh, n, derivation, 
  (z1,z2,z3,z4,z5) +-> DSPDEbdom(z1,z2,z3,z4,z5,bb))
det := lb * lb + lh * lh
DSPDEsys0(a, b, h, c1, c2, lb, lh, n, derivation, 
  (z1,z2,z3,z4,z5) +-> DSPDEmix(z1,z2,z3,z4,z5,bb,det))

DSPDEsys0(a, b, h, c1, c2, lb, lh, n, derivation, getlc) ==
ans1 := ans2 := 0::UP
repeat
  zero? c1 and zero? c2 => return [ans1, ans2]
n < 0 or (u := getlc(c1,c2,lb,lh,n::N)) case "failed" => return "failed"
lq := u::List(UP)
q1 := first lq
q2 := second lq
c1 := c1 - a * derivation(q1) - h * q1 + b * q2
c2 := c2 - a * derivation(q2) - b * q1 - h * q2
n := n - 1
ans1 := ans1 + q1
ans2 := ans2 + q2

DSPDEmix(c1, c2, lb, lh, n, d, det) ==
rh1:F :=
  zero? c1 => 0
  (d1 := degree(c1)::Z - d) < n => 0
  d1 > n => return "failed"
  leadingCoefficient c1
rh2:F :=
  zero? c2 => 0
  (d2 := degree(c2)::Z - d) < n => 0
  d2 > n => return "failed"
  leadingCoefficient c2
q1 := (rh1 * lh + rh2 * lb) / det
q2 := (rh2 * lh - rh1 * lb) / det
[monomial(q1, n), monomial(q2, n)]

DSPDEhdom(c1, c2, lb, lh, n, d) ==
q1:UP :=
  zero? c1 => 0
  (d1 := degree(c1)::Z - d) < n => 0
  d1 > n => return "failed"
  monomial(leadingCoefficient(c1) / lh, n)
q2:UP :=
  zero? c2 => 0
  (d2 := degree(c2)::Z - d) < n => 0
  d2 > n => return "failed"
  monomial(leadingCoefficient(c2) / lh, n)
[q1, q2]

DSPDEbdom(c1, c2, lb, lh, n, d) ==
q1:UP :=
  zero? c2 => 0
  (d2 := degree(c2)::Z - d) < n => 0
  d2 > n => return "failed"
  monomial(leadingCoefficient(c2) / lb, n)
q2:UP :=
  zero? c1 => 0
  (d1 := degree(c1)::Z - d) < n => 0
  d1 > n => return "failed"
  monomial(- leadingCoefficient(c1) / lb, n)
[q1, q2]

-- return a common bound on the degrees of a solution of
-- A (Q1', Q2') + ((H, -B), (B, H)) (Q1,Q2) = (C1,C2), Q1 = y1 T, Q2 = y2 T
-- cancellation at infinity is possible
-- a and b are nonzero
-- base case: F' = 0
getBound(a, b, h, c1, c2) ==
da := (degree a)::Z
dc :=
  zero? c1 => degree(c2)::Z
  zero? c2 => degree(c1)::Z
  max(degree c1, degree c2)::Z
hh :=
  zero? h => 0
  degree(h)::Z
db := max(hh, bb := degree(b)::Z)
da < db + 1 => dc - db
da > db + 1 => max(0, dc - da + 1)
bb >= hh => dc - db
(n := retractIfCan(leadingCoefficient(h) / leadingCoefficient(a)
  )@Union(Z, "failed")) case Z => max(n::Z, dc - db)
dc - db

—— RDETRS.dotabb ——

"RDETRS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=RDETRS"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"RDETRS" -> "PFECAT"
package SOLVETRA TransSolvePackage

--- TransSolvePackage.input ---

)set break resume
)sys rm -f TransSolvePackage.output
)spool TransSolvePackage.output
)set message test on
)set message auto off
)clear all

-- S 1 of 4
solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)
--R
--R (1)
--R [phi= 2atan(\%phi0) - theta, phi= 2atan(\%phi1) - theta,
--R
--R phi =
--R 2
--R *
--R atan
--R ROOT
--R
--R 4 theta 4 4 2 theta 2
--R - 3v tan(-----) + (- 6v + 24g l v )tan(-----)
--R 2 2
--R +
--R 4 2 2 2
--R - 3v + 24g l v - 48g l
--R *
--R 2
--R \%phi1
--R +
--R 4 theta 4 4 2 theta 2
--R - 2v tan(-----) + (- 4v + 16g l v )tan(-----)
--R 2 2
--R +
--R 4 2 2 2
--R - 2v + 16g l v - 32g l
--R *
--R \%phi0
--R +
--R 2 theta 3 2 2 2 theta
--R - 16g l v tan(-----) + (- 16g l v + 64g l )tan(-----)
--R 2 2
--R *
--R \%phi1
--R +
\[-R 4 \theta 4 \quad 4 \quad 2 \quad \theta 2\]
\[-R \quad 3v \tan(\theta) + (-6v + 24g_1 v)\tan(\theta)\]
\[-R \quad 2\]
\[-R +\]
\[-R 4 \quad 2 \quad 2 \quad 2\]
\[-R \quad -3v + 24g_1 v - 48g_1\]
\[-R \quad 2\]
\[-R *\]
\[-R 2\]
\[-R \phi 0\]
\[-R +\]
\[-R 2 \quad \theta 3\]
\[-R \quad -16g_1 v \tan(\theta)\]
\[-R \quad 2\]
\[-R +\]
\[-R 2 \quad 2 \quad 2 \quad \theta\]
\[-R \quad (-16g_1 v + 64g_1)\tan(\theta)\]
\[-R \quad 2\]
\[-R *\]
\[-R \phi 0\]
\[-R +\]
\[-R 4 \quad 2 \quad \theta 4 \quad 4 \quad \theta 2 \quad 4\]
\[-R \quad (8v + 16g_1 v)\tan(\theta) + 16v \tan(\theta) + 8v\]
\[-R \quad 2 \quad 2\]
\[-R +\]
\[-R \quad 2 \quad 2\]
\[-R \quad -16g_1 v - 64g_1\]
\[-R +\]
\[-R 2 \quad \theta 2 \quad 2\]
\[-R \quad (-v \tan(\theta) - v + 4g_1)\phi 1\]
\[-R \quad 2\]
\[-R +\]
\[-R 2 \quad \theta 2 \quad 2\]
\[-R \quad (-v \tan(\theta) - v + 4g_1)\phi 0 - 8g_1 \tan(\theta)\]
\[-R \quad 2 \quad 2\]
\[-R /\]
\[-R \quad 2 \quad \theta 2 \quad 2\]
\[-R \quad 2v \tan(\theta) + 2v - 8g_1\]
\[-R \quad 2\]
\[-R +\]
\[-R \quad - \theta\]
\[-R ,\]
\[-R \phi =\]
\[-R -\]
\[-R 2\]
\[-R *\]
\[-R \atan\]
\[-R \\ROOT\]
\[-R 4 \theta 4 \quad 4 \quad 2 \quad \theta 2\]
\[-R \quad -3v \tan(\theta) + (-6v + 24g_1 v)\tan(\theta)\]
\[\begin{align*}
\phi_1 &= 2 \\
&+ \frac{4}{2} \theta_4 - 2v \tan(\theta_4) \\
&+ \frac{4}{2} \theta_2 \theta_4 - 2v \tan(\theta_4) \\
&+ \frac{4}{2} \theta_2 \theta_4 - 2v \tan(\theta_4) \\
&+ \frac{4}{2} \theta_2 \theta_4 - 2v \tan(\theta_4) \\
&+ \frac{4}{2} \theta_2 \theta_4 - 2v \tan(\theta_4) \\
&+ \frac{4}{2} \theta_2 \theta_4 - 2v \tan(\theta_4)
\end{align*}\]
\[
\begin{align*}
\text{--R} & \quad + \quad 4 \quad 2 \quad \theta \quad 4 \quad 4 \quad \theta \quad 2 \quad 4 \\
\text{--R} & \quad (8v + 16g + v\ tan(------) + 16v\ tan(------) + 8v \\
\text{--R} & \quad 2 \quad 2 \\
\text{--R} & \quad + \\
\text{--R} & \quad 2 \quad 2 \quad 2 \\
\text{--R} & \quad - 16g + v - 64g + l \\
\text{--R} & \quad + \\
\text{--R} & \quad 2 \quad \theta \quad 2 \quad 2 \\
\text{--R} & \quad (v\ tan(------) + v - 4g + l)\%phi1 \\
\text{--R} & \quad 2 \\
\text{--R} & \quad + \\
\text{--R} & \quad 2 \quad \theta \quad 2 \quad 2 \\
\text{--R} & \quad \theta \\
\text{--R} & \quad (v\ tan(------) + v - 4g + l)\%phi0 + 8g + l\ tan(------) \\
\text{--R} & \quad 2 \quad 2 \\
\text{--R} & \quad / \\
\text{--R} & \quad 2 \quad \theta \quad 2 \quad 2 \\
\text{--R} & \quad 2v\ tan(------) + 2v - 8g + l \\
\text{--R} & \quad 2 \\
\text{--R} & \quad + \\
\text{--R} & \quad - \ theta \\
\text{--R} & \quad \} \\
\text{Type: List(Equation(Expression(Integer)))}
\end{align*}
\]

--E 1

--S 2 of 4
definingPolynomial %phi0
--R
--R
--R (2)
--R \quad 4 \quad 2 \quad 2 \quad 2 \quad \theta \quad 2 \\
--R \quad ((%phi0 - 2%phi0 + 1)v + (- 4%phi0 - 4)g + l)\ tan(------) \\
--R \quad 2 \\
--R + \\
--R \quad 3 \quad \theta \quad 4 \quad 2 \quad 2 \\
--R \quad (8%phi0 + 8%phi0)g + l\ tan(------) + (%phi0 - 2%phi0 + 1)v \\
--R \quad 2 \\
--R + \\
--R \quad 4 \quad 2 \\
--R \quad (- 4%phi0 - 4%phi0 + l)\ g \\
--R / \\
--R \quad 2 \quad \theta \quad 2 \quad 2 \\
--R \quad v\ tan(------) + v - 4g + l \\
--R \quad 2 \\
\text{Type: Expression(Integer)}
--E 2

--S 3 of 4
definingPolynomial %phi1
--R
--R
--R (3)
--R (\theta^2 + 2 \theta \tan(-\theta) + v - 4g l)\phi_0 / 2
--R +
--R (\theta^2 + 2 \theta \tan(-\theta) + \phi_1 \tan(-\theta) - 4g l)\phi_0 / 2
--R +
--R (2 \theta \tan(-\theta) + \phi_1 \tan(-\theta) - 4g l)\phi_0 / 2

--R ----------------------------- Operations ---------------------------------
--R solve : Expression(R) -> List(Equation(Expression(R)))
--R solve : Equation(Expression(R)) -> List(Equation(Expression(R)))
--R solve : (Equation(Expression(R)),Symbol) -> List(Equation(Expression(R)))
--R solve : (Expression(R),Symbol) -> List(Equation(Expression(R)))
--R solve : (List(Equation(Expression(R))),List(Symbol)) -> List(List(Equation(Expression(R))))

--E 4
This package tries to find solutions of equations of type Expression(R).

This means expressions involving transcendental, exponential, logarithmic
and nthRoot functions.

After trying to transform different kernels to one kernel by applying
several rules, it calls zerosOf for the SparseUnivariatePolynomial in
the remaining kernel.

For example the expression
\[ \sin(x) \cdot \cos(x) - 2 \]
will be transformed to
\[ -2 \tan(x/2)**4 - 2 \tan(x/2)**3 - 4 \tan(x/2)**2 + 2 \tan(x/2) - 2 \]
by using the function normalize and then to
\[ -2 \tan(x)**2 + \tan(x) - 2 \]
with help of subsTan. This function tries to express the given function
in terms of tan(x/2) to express in terms of tan(x).

Other examples are the expressions
\[ \sqrt{x+1} + \sqrt{x+7} + 1 \]
\[ \sqrt{\sin(x)} + 1 \]
\[ \text{solve}\left(\frac{1}{2}v^2\cos(\theta+\phi)\cos(\theta+\phi)+g*1\cos(\phi)=g*1,\phi\right) \]
definingPolynomial Xphi0
definingPolynomial Xphi1

See Also:
c )show TransSolvePackage
TransSolvePackage (SOLVETRA)

Exports:
solve

— package SOLVETRA TransSolvePackage —

)abbrev package SOLVETRA TransSolvePackage
++ Author: W. Wiwianka, Martin Rubey
++ Date Created: Summer 1991
++ Date Last Changed: 9/91
++ Description:
++ This package tries to find solutions of equations of type Expression(R).
++ This means expressions involving transcendental, exponential, logarithmic
++ and n-thRoot functions.
++ After trying to transform different kernels to one kernel by applying
++ several rules, it calls zerosOf for the SparseUnivariatePolynomial in
++ the remaining kernel.
++ For example the expression \(\sin(x)\cdot\cos(x) - 2\) will be transformed to
++ \(-2\tan(x/2)^4 - 2\tan(x/2)^3 - 4\tan(x/2)^2 + 2\tan(x/2) - 2\)
++ by using the function normalize and then to
++ \(-2\tan(x)^2 + \tan(x) - 2\)
++ with help of subsTan. This function tries to express the given function
++ in terms of \(\tan(x/2)\) to express in terms of \(\tan(x)\).
++ Other examples are the expressions \(\sqrt{x+1} + \sqrt{x+7} + 1\) or
++ \(\sqrt{\sin(x)} + 1\).

TransSolvePackage(R) : Exports == Implementation where
R : Join(OrderedSet, EuclideanDomain, RetractableTo Integer,
      LinearlyExplicitRingOver Integer, CharacteristicZero)

I  ==> Integer
NNI ==> NonNegativeInteger
RE ==> Expression R
EQ ==> Equation
S  ==> Symbol
V  ==> Variable
L ==> List
K ==> Kernel RE
SUP ==> SparseUnivariatePolynomial
C ==> Complex
F ==> Float
INT ==> Interval
SMP ==> SparseMultivariatePolynomial

Exports == with

solve : RE -> L EQ RE
++ solve(expr) finds the solutions of the equation expr = 0
++ where expr is a function of type Expression(R)
++ with respect to the unique symbol x appearing in eq.
solve : EQ RE -> L EQ RE
++ solve(eq) finds the solutions of the equation eq
++ where eq is an equation of functions of type Expression(R)
++ with respect to the unique symbol x appearing in eq.
solve : ( EQ RE , S ) -> L EQ RE
++ solve(eq,x) finds the solutions of the equation eq
++ where eq is an equation of functions of type Expression(R)
++ with respect to the symbol x.
solve : ( RE , S ) -> L EQ RE
++ solve(expr,x) finds the solutions of the equation expr = 0
++ with respect to the symbol x where expr is a function
++ of type Expression(R).
++
++X solve(1/2*v*v*cos(theta+phi)*cos(theta+phi)+g*l*cos(phi)=g*l,phi)
++X definingPolynomial %phi0
++X definingPolynomial %phi1

solve : (L EQ RE, L S) -> L L EQ RE
++ solve(leqs, lvar) returns a list of solutions to the list of
++ equations leqs with respect to the list of symbols lvar.
-- solve : (L EQ RE, L Kernel RE) -> L L EQ RE
-- ++ solve(leqs, lker) returns a list of solutions to the list
-- ++ of equations leqs with respect to the list of kernels lker.

Implementation == add
import ACF
import HomogeneousAggregate(R)
import AlgebraicManipulations(R, RE)
import TranscendentalManipulations(R, RE)
import TrigonometricManipulations(R, RE)
import ElementaryFunctionStructurePackage(R, RE)
import SparseUnivariatePolynomial(R)
import LinearSystemMatrixPackage(RE,Vector RE,Vector RE,Matrix RE)
import TransSolvePackageService(R)
import MultivariateFactorize(K, IndexedExponents K, R, SMP(R, K))
---- Local Function Declarations ----

solveInner : (RE, S) -> L EQ RE
tryToTrans : ( RE, S) -> RE
eliminateKernRoot: (RE , K) -> RE
eliminateRoot: (RE , S) -> RE

combineLog : ( RE , S ) -> RE
testLog : ( RE , S ) -> Boolean
splitExpr : ( RE ) -> L RE
buildnexpr : ( RE , S ) -> L RE
logsumtolog : RE -> RE
logexpp : ( RE , RE ) -> RE

testRootk : ( RE, S) -> Boolean
testkernel : ( RE , S ) -> Boolean
funcinv : ( RE , RE ) -> Union(RE,"failed")
testTrig : ( RE, S ) -> Boolean
testHTrig : ( RE, S ) -> Boolean
tableXkernels : ( RE , S ) -> L RE
subsTan : ( RE , S ) -> RE

-- exported functions

solve(oside: RE) : L EQ RE ==
  zero? oside => error "equation is always satisfied"
  lv := variables oside
  empty? lv => error "inconsistent equation"
  #lv>1 => error "too many variables"
  solve(oside,lv.first)

solve(equ:EQ RE) : L EQ RE ==
  solve(lhs(equ)-rhs(equ))

solve(equ:EQ RE, x:S) : L EQ RE ==
  oneside:=lhs(equ)-rhs(equ)
  solve(oneside,x)

  if R has QuotientFieldCategory(Integer) then
    retractIfCan(rhs sol)@Union(Integer,"failed") case "failed" => true
  else
    retractIfCan(rhs sol)@Union(Fraction Integer,"failed") case "failed" => true
  zero? eval(lside,sol) => true
false

solve(lside: RE, x:S) : L EQ RE ==
[sol for sol in solveInner(lside,x) | testZero?(lside,sol)]

solveInner(lside: RE, x:S) : L EQ RE ==
lside:=eliminateRoot(lside,x)
ausgabe1:=tableXkernels(lside,x)

X:=new()@Symbol
Y:=new()@Symbol::RE
(#ausgabe1) = 1 =>
bigX:= (first ausgabe1)::RE
eq1:=eval(lside,bigX=(X::RE))
-- Type : Expression R
f:=univariate(eq1,first kernels (X::RE))
-- Type : Fraction SparseUnivariatePolynomial Expression R
lfatt:= factors factorPolynomial numer f
lr:L RE := "append" /[zerosOf(fatt.factor,x) for fatt in lfatt]
-- Type : List Expression R
r1:=[]::L RE
for i in 1..#lr repeat
  finv := funcinv(bigX,lr(i))
  if finv case RE then r1:=cons(finv::RE,r1)
bigX_back:=funcinv(bigX,bigX)::RE
if not testkernel(bigX_back,x) then
  if bigX = bigX_back then return []::L EQ RE
  return
  "append"/[solve(bigX_back-ri, x) for ri in r1]
newlist:=[]::L RE
for i in 1..#r1 repeat
  elR := eliminateRoot((numer(bigX_back - r1(i))::RE ),x)
  f:=univariate(elR, kernel(x))
  -- Type : Fraction SparseUnivariatePolynomial Expression R
  lfatt:= factors factorPolynomial numer f
  secondsol:="append" /[zerosOf(ff.factor,x) for ff in lfatt]
  for j in 1..#secondsol repeat
    newlist:=cons((x::RE)=rootSimp( secondsol(j) ),newlist)
newlist

newlside:=tryToTrans(lside,x) ::RE
listofkernels:=tableXkernels(newlside,x)
(#listofkernels) = 1 => solve(newlside,x)
lfacts := factors factor(numer lside)
#lfacts > 1 =>
sols : L EQ RE := []
for frec in lfacts repeat
  sols := append(solve(frec.factor :: RE, x), sols)
sols
return []::L EQ RE
-- local functions

-- This function was suggested by Manuel Bronstein as a simpler
-- alternative to normalize.
simplifyingLog(f:RE):RE ==
    (u:=isExpt(f,"exp"::Symbol)) case Record(var:Kernel RE,exponent:Integer) =>
        rec := u::Record(var:Kernel RE,exponent:Integer)
        rec.exponent * first argument(rec.var)
        log f


testkernel(var1:RE,y:S) : Boolean ==
    var1:=eliminateRoot(var1,y)
    listvar1:=tableXkernels(var1,y)
    if (#listvar1 = 1) and ((listvar1(1) = (y::RE))@Boolean ) then
        true
    else if #listvar1 = 0 then true
    else false


solveRetract(lexpr:L RE, lvar:L S):Union(L L EQ RE, "failed") ==
    nlexpr : L Fraction Polynomial R := []
    for expr in lexpr repeat
        rf:Union(Fraction Polynomial R, "failed") := retractIfCan(expr)$RE
        rf case "failed" => return "failed"
        nlexpr := cons(rf, nlexpr)
    radicalSolve(nlexpr, lvar)$RadicalSolvePackage(R)


tryToTrans(lside: RE, x:S) : RE ==
    if testTrig(lside,x) or testHTrig(lside,x) then
        convLside:=( simplify(lside) )::RE
        resultLside:=convLside
        listConvLside:=tableXkernels(convLside,x)
        if (#listConvLside) > 1 then
            NormConvLside:=normalize(convLside,x)
            NormConvLside:=( NormConvLside ):: RE
            resultLside:=subsTan(NormConvLside, x)
        else if testLog(lside,x) then
            numlside:=numer(lside)::RE
            resultLside:=combineLog(numlside,x)
        else
            NormConvLside:=normalize(lside,x)
            NormConvLside:=( NormConvLside ):: RE
            resultLside:=NormConvLside
            listConvLside:=tableXkernels(NormConvLside,x)
            if (#listConvLside) > 1 then
                cnormConvLside:=complexNormalize(lside,x)
                cnormConvLside:=cnormConvLside::RE
                resultLside:=cnormConvLside
listcnorm := tableXkernels(cnormConvLside, x)
if (#listcnorm) > 1 then
  if testLog(cnormConvLside, x) then
    numlside := numer(cnormConvLside)::RE
    resultLside := combineLog(numlside, x)
resultLside

subsTan(exprvar::RE, y::S) := RE ==
Z := new()@Symbol
listofkern := tableXkernels(exprvar, y)
varkern := (first listofkern)::RE
Y := (numer first argument first (kernels(varkern)))::RE
test := Boolean := varkern = tan(((Y::RE)/(2::RE))::RE)
if not((#listofkern=1) and test) then
  return exprvar
fZ := eval(exprvar, varkern = (Z::RE))
fN := (numer fZ)::RE
f := univariate(fN, first kernels(Z::RE))
secondfun := (-2*(Y::RE)/((Y::RE)**2-1))::RE
g := univariate(secondfun, first kernels(y::RE))
H := (new()@Symbol)::RE
newH := univariate(H, first kernels(Z::RE))
result := decomposeFunc(f, g, newH)
if not(result = f) then
  result1 := result(H::RE)
  resultnew := eval(result1, H = ((tan((Y::RE))::RE)))
else return exprvar

eliminateKernRoot(var::RE, varkern::K) := RE ==
x := new()@Symbol
var1 := eval(var, (varkern::RE) = (X::RE))
var2 := numer univariate(var1, first kernels(X::RE))
var3 := monomial(1, (retract(second argument varkern)@I)::NNI@SUP RE_
  - monomial(first argument varkern, 0::NNI)@SUP RE
resultvar := resultant(var2, var3)

eliminateRoot(var::RE, y::S) := RE ==
var1 := var
while testRootk(var1, y) repeat
  varlistk1 := tableXkernels(var1, y)
  for i in varlistk1 repeat
    if is?(i, "nthRoot"::S) then
      var1 := eliminateKernRoot(var1, first kernels(i::RE))
    var1

logsumtolog(var::RE) := RE ==
  (listofexpr := isPlus(var)) case "failed" => var
listofexpr := listofexpr :: L RE
listforgcd := [] :: L R
for i in listofexpr repeat
  expcoeff := leadingCoefficient(numer(i))
  listforgcd := cons(expcoeff, listforgcd)
gcdcoeff := gcd(listforgcd) :: RE
newexpr := 0
for i in listofexpr repeat
  exprlist := splitExpr(i :: RE)
  newexpr := newexpr + logexpp(exprlist.2, exprlist.1 / gcdcoeff)
kernvar := kernels(newexpr)
var2 := 1 :: RE
for i in kernvar repeat
  var2 := var2 * (first argument i)
gcdcoeff * log(var2)

testLog(expr :: RE, Z :: S) := Boolean ==
testList := [log] :: L S
kernexpr := tableXkernels(expr, Z)
if #kernexpr = 0 then
  return false
for i in kernexpr repeat
  if not member?(name(first kernels(i)), testList) or
    not testkernel((first argument first kernels(i)), Z) then
    return false
true

splitExpr(expr :: RE) := L RE ==
lcoeff := leadingCoefficient((numer expr))
exprwcoeff := expr
listexpr := isTimes(exprwcoeff)
if listexpr case "failed" then
  [1 :: RE, expr]
else
  listexpr := remove_!(lcoeff :: RE, listexpr)
  cons(lcoeff :: RE, listexpr)
buildnexpr(expr :: RE, Z :: S) := L RE ==
nlist := splitExpr(expr)
n2list := remove_!(nlist.1, nlist)
ancoeff := 1
ansmant := 0
for i in n2list repeat
  if freeOf?(i :: RE, Z) then
    ancoeff := (i :: RE) * ancoeff
  else
    ancoeff := (i :: RE)
  [ancoeff, ansmant * nlist.1]
logexpp(expr1:RE, expr2:RE) : RE ==
log((first argument first kernels(expr1)**expr2)

combineLog(expr:RE,Y:S) : RE ==
exptable:Table(RE,RE):=table()
(isPlus(expr)) case "failed" => expr
ans:RE:=0
while expr ~= 0 repeat
loopexpr:RE:=leadingMonomial(numer(expr))::RE
if testLog(loopexpr,Y) and (#tableXkernels(loopexpr,Y)=1) then
expr:=buildnexpr(loopexpr,Y)
exptable.(exprr.1):= exprtable.(exprr.1) + exprr.2
else
ans:=ans+loopexpr
expr:=(reductum(numer expr))::RE
ansexpr:=ansexpr + logsumtolog(exprtable.i) * (i::RE)
ansexpr:=ansexpr + ans

testRootk(varlistk:RE,y:S) : Boolean ==
testList:=[nthRoot]::L S
kernelofeqnvar:=tableXkernels(varlistk,y)
if #kernelofeqnvar = 0 then
return false
for i in kernelofeqnvar repeat
if member?(name(first kernels(i)),testList) then
return true
false

tableXkernels(evar:RE,Z:S) : L RE ==
kOfvar:=kernels(evar)
listkOfvar:=[]:L RE
for i in kOfvar repeat
if not freeOf?(i::RE,Z) then
listkOfvar:=cons(i::RE,listkOfvar)
listkOfvar

testTrig(eqnvar:RE,Z:S) : Boolean ==
testList:=[sin, cos, tan, cot, sec, csc]:L S
kernelofeqnvar:=tableXkernels(eqnvar,Z)
if #kernelofeqnvar = 0 then
return false
for i in kernelofeqnvar repeat
if not member?(name(first kernels(i)),testList) or
not testkernel((first argument first kernels(i)),Z) then
return false
true

testHTrig(eqnvar:RE,Z:S) : Boolean ==
    testList:=[sinh, cosh, tanh, coth, sech, csch]:L S
    kernelofeqnvar:=tableXkernels(eqnvar,Z)
    if #kernelofeqnvar = 0 then
        return false
    for i in kernelofeqnvar repeat
        if not member?(name(first kernels(i)),testList) or _
            not testkernel( (first argument first kernels(i)) ,Z) then
            return false
    true

-- Auxiliary local function for use in funcinv.
makeInterval(l:R):C INT F ==
    if R has complex and R has ConvertibleTo(C F) then
        map(interval$INT(F),convert(l)$R)$ComplexFunctions2(F,INT F)
    else
        error "This should never happen"

funcinv(k:RE,l:RE) : Union(RE,"failed") ==
    is?(k, "sin"::Symbol) => asin(l)
    is?(k, "cos"::Symbol) => acos(l)
    is?(k, "tan"::Symbol) => atan(l)
    is?(k, "cot"::Symbol) => acot(l)
    is?(k, "sec"::Symbol) =>
        l = 0 => "failed"
        asec(l)
    is?(k, "csc"::Symbol) =>
        l = 0 => "failed"
        acsc(l)
    is?(k, "sinh"::Symbol) => asinh(l)
    is?(k, "cosh"::Symbol) => acosh(l)
    is?(k, "tanh"::Symbol) => tanh(l)
    is?(k, "coth"::Symbol) => coth(l)
    is?(k, "sech"::Symbol) => asech(l)
    is?(k, "csch"::Symbol) => acsch(l)
    is?(k, "atan"::Symbol) =>
        l = 0 => "failed"
        cot(l)
    is?(k, "acot"::Symbol) =>
        l = 0 => "failed"
        csc(l)
    is?(k, "asin"::Symbol) => sin(l)
    is?(k, "acos"::Symbol) => cos(l)
    is?(k, "asec"::Symbol) => sec(l)
    is?(k, "acsc"::Symbol) =>
        l = 0 => "failed"
        csc(l)
    is?(k, "asinh"::Symbol) => sinh(l)
    is?(k, "acosh"::Symbol) => cosh(l)
is?(k, "atanh": Symbol) => tanh(l)

is?(k, "acoth": Symbol) =>
  l = 0 => "failed"
  coth(l)

is?(k, "asech": Symbol) => sech(l)

is?(k, "acsch": Symbol) =>
  l = 0 => "failed"
  csch(l)

is?(k, "exp": Symbol) =>
  l = 0 => "failed"
  simplifyingLog l

is?(k, "log": Symbol) =>
  \text{if } R \text{ has complex and } R \text{ has ConvertibleTo(C F) then}
  \text{We will check to see if the imaginary part lies in } [-\pi, \pi)
  \text{ze : Expression C INT F}
  \text{ze := map(makeInterval,l)$ExpressionFunctions2(R,C INT F)}
  \text{z : Union(C INT F,"failed") := retractIfCan ze}
  \text{z case "failed" => exp l}
  \text{im := imag z}
  \text{fpi : Float := pi()}
  \text{(-fpi < inf(im)) and (sup(im) <= fpi) => exp l}
  "failed"

else -- R not Complex or something which doesn't map to Complex Floats
  \text{exp l}

is?(k, "%power": Symbol) =>
  (t:=normalize(l)) = 0 => "failed"
  log t

import SystemSolvePackage(RE)

ker2Poly(k:Kernel RE, lvar:L S): Polynomial RE ==
  member?(nm:=name k, lvar) => nm :: Polynomial RE
  k :: RE :: Polynomial RE

smp2Poly(pol:SMP(R,Kernel RE), lvar:L S): Polynomial RE ==
  map(x +-> ker2Poly(x, lvar),
      y +-> y::RE::Polynomial RE, pol)$PolynomialCategoryLifting( IndexedExponents Kernel RE, Kernel RE, R, SMP(R, Kernel RE), Polynomial RE)

makeFracPoly(expr:RE, lvar:L S): Fraction Polynomial RE ==
  smp2Poly(numer expr, lvar) / smp2Poly(denom expr, lvar)

makeREpol(pol: Polynomial RE): RE ==
  lvar := variables pol
  lval : List RE := [v::RE for v in lvar]
  ground eval(pol,lvar,lval)

makeRE(frac: Fraction Polynomial RE): RE ==
The input

\[
\text{solve} (\sinh(z) = \cosh(z), z)
\]
generates the error (reported as bug \# 102):

>> Error detected within library code:
   No identity element for reduce of empty list using operation append

— package SOLVETRA TransSolvePackage —
-- lker2 := \[v::Kernel(RE) for v in lvar\]
-- lval2 := \[k::RE for k in lker\]
-- [[map(eval(#1,lker2,lval2), neq) for neq in sol] for sol in ans]

---

-- SOLVETRA.dotabb --

"SOLVETRA" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVETRA"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SOLVETRA" -> "FS"

---

package SOLVESER TransSolvePackageService

--- TransSolvePackageService.input ---

)set break resume
)sys rm -f TransSolvePackageService.output
)spool TransSolvePackageService.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TransSolvePackageService
--R
--R TransSolvePackageService(R: Join(IntegralDomain,OrderedSet)) is a package constructor
--R Abbreviation for TransSolvePackageService is SOLVESER
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for SOLVESER
--R
--R----------------------------------- Operations -----------------------------------
--R decomposeFunc : (Fraction(SparseUnivariatePolynomial(Expression(R))),Fraction(SparseUnivariatePolynomial(Expression(R)))) -> Fraction(SparseUnivariatePolynomial(Expression(R)))
--R unvectorise : (Vector(Expression(R)),Fraction(SparseUnivariatePolynomial(Expression(R)))) -> Fraction(SparseUnivariatePolynomial(Expression(R))),Integer
--E 1

)spool
)lisp (bye)

---
This package finds the function func3 where func1 and func2 are given and func1 = func3(func2). If there is no solution then function func1 will be returned.

An example would be

\[
\text{func1} := 8\times X^3 + 32\times X^2 - 14\times X \quad \text{::EXPR INT}
\]

and

\[
\text{func2} := 2\times X \quad \text{::EXPR INT}
\]

convert them via univariate to FRAC SUP EXPR INT and then the solution is

\[
\text{func3} := X^3 + X^2 - X
\]

of type FRAC SUP EXPR INT

See Also:
- )show TransSolvePackageService
— package SOLVESER TransSolvePackageService —

)abbrev package SOLVESER TransSolvePackageService
++ Author: W. Wiwianka
++ Date Created: Summer 1991
++ Date Last Changed: 9/91
++ Description:
++ This package finds the function func3 where func1 and func2
++ are given and func1 = func3(func2). If there is no solution then
++ function func1 will be returned.
++ An example would be \spad{func1:= 8*X**3+32*X**2-14*X ::EXPR INT} and
++ \spad{func2:=2*X ::EXPR INT} convert them via univariate
++ to FRAC SUP EXPR INT and then the solution is \spad{func3:=X**3+X**2-X}
++ of type FRAC SUP EXPR INT

TransSolvePackageService(R) :Exports == Implementation where
R : Join(IntegralDomain, OrderedSet)
RE ==> Expression R
EQ ==> Equation
S ==> Symbol
V ==> Variable
L ==> List
SUP ==> SparseUnivariatePolynomial
ACF ==> AlgebraicallyClosedField()

Exports == with

  decomposeFunc : ( Fraction SUP RE , Fraction SUP RE, Fraction SUP RE ) -> Fraction SUP RE
++ decomposeFunc(func1, func2, newvar) returns a function func3 where
++ func1 = func3(func2) and expresses it in the new variable newvar.
++ If there is no solution then func1 will be returned.

  unvectorise : ( Vector RE , Fraction SUP RE , Integer ) -> Fraction SUP RE
++ unvectorise(vect, var, n) returns
++ \spad{vect(1) + vect(2)*var + ... + vect(n+1)*var**(n)} where
++ vect is the vector of the coefficients of the polynomial, var
++ the new variable and n the degree.

Implementation == add
import ACF
import TranscendentalManipulations(R, RE)
import ElementaryFunctionStructurePackage(R, RE)
import SparseUnivariatePolynomial(R)
import LinearSystemMatrixPackage(RE,Vector RE,Vector RE,Matrix RE)
import HomogeneousAggregate(R)

        ---- Local Function Declarations ----
subsSolve : (SUP RE, NonNegativeInteger, SUP RE, SUP RE, Integer, Fraction SUP RE) -> Union(SUP RE, "failed")

-- exported functions

unvectorise(vect: Vector RE, var: Fraction SUP RE, n: Integer) : Fraction SUP RE ==
    Z := new()@Symbol
    polyvar := Fraction SUP RE := 0
    for i in 1..((n+1)::Integer) repeat
        vecti := univariate(vect( i ), first kernels(Z::RE))
        polyvar := polyvar + (vecti) *( var)**((n-i+1)::NonNegativeInteger)
    polyvar

    X := new()@Symbol
    f1 := numer(exprf)
    f2 := denom(exprf)
    g1 := numer(exprg)
    g2 := denom(exprg)
    degF := max(degree(numer(exprf)), degree(denom(exprf)))
    degG := max(degree(g1), degree(g2))
    newF1, newF2 := Union(SUP RE, "failed")
    N := degF exquo degG
    if not (N case "failed") then
        m := N::Integer
        newF1 := subsSolve(f1, degF, g1, g2, m, newH)
        if f2 = 1 then
            newF2 := 1 :: SUP RE
        else newF2 := subsSolve(f2, degF, g1, g2, m, newH)
        if (not (newF1 case "failed") and (not (newF2 case "failed"))) then
            newF := newF1/newF2
        else return exprf
    else return exprf

-- local functions

for i in 0..M repeat
    coeffmat := new((DegF+1),1,0)@Matrix RE
    for i in 0..M repeat
        coeffmat := horizConcat(coeffmat, (vectorise((G1**(M-i)::NonNegativeInteger))*G2**i), (DegF+1))
    vec := vectorise(F, DegF+1)
    coeffma := subMatrix(coeffmat, 1, (DegF+1), 2, (M+2))
    solvar := solve(coeffma, vec)
    if not (solvar.particular case "failed") then
solvevarlist:=(solvar.particular)::Vector RE
resul:= numer(unvectorise(solvevarlist,( HH ),M))
resul
else return "failed"

---

SOLVESER.dotabb ---

"SOLVESER" [color="#FF4488",href="bookvol10.4.pdf#nameddest=SOLVESER"]
"FS" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FS"]
"SOLVESER" -> "FS"

---

package TRIMAT TriangularMatrixOperations

--- TriangularMatrixOperations.input ---

)set break resume
)sys rm -f TriangularMatrixOperations.output
)spool TriangularMatrixOperations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TriangularMatrixOperations
--R
--R TriangularMatrixOperations(R: IntegralDomain,Row: FiniteLinearAggregate(R),Col: FiniteLinearAggregate(R)) is a package constructor
--R Abbreviation for TriangularMatrixOperations is TRIMAT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TRIMAT
--R
--R-------------------------------------------------- Operations -------------------------------
--R LowTriBddDenomInv : (M,R) -> M         UpTriBddDenomInv : (M,R) -> M
--R
--E 1

)spool
)lisp (bye)

---
This package provides functions that compute "fraction-free" inverses of upper and lower triangular matrices over a integral domain. By "fraction-free inverses" we mean the following: given a matrix $B$ with entries in $R$ and an element $d$ of $R$ such that $d \cdot \text{inv}(B)$ also has entries in $R$, we return $d \cdot \text{inv}(B)$. Thus, it is not necessary to pass to the quotient field in any of our computations.

See Also:
- )show TriangularMatrixOperations
Given a matrix $B$ with entries in $R$ and an element $d$ of $R$ such that $d \cdot \text{inv}(B)$ also has entries in $R$, we return $d \cdot \text{inv}(B)$. Thus, it is not necessary to pass to the quotient field in any of our computations.

TriangularMatrixOperations($R, \text{Row}, \text{Col}, M$): Exports == Implementation where

$R : \text{IntegralDomain}$

$\text{Row} : \text{FiniteLinearAggregate} \, R$

$\text{Col} : \text{FiniteLinearAggregate} \, R$

$M : \text{MatrixCategory}(R, \text{Row}, \text{Col})$

Exports ==> with

$\text{UpTriBddDenomInv} : (M, R) \to M$

++ $\text{UpTriBddDenomInv}(B, d)$ returns $M$, where

++ $B$ is a non-singular upper triangular matrix and $d$ is an element of $R$ such that $M = d \cdot \text{inv}(B)$ has entries in $R$.

$\text{LowTriBddDenomInv} : (M, R) \to M$

++ $\text{LowTriBddDenomInv}(B, d)$ returns $M$, where

++ $B$ is a non-singular lower triangular matrix and $d$ is an element of $R$ such that $M = d \cdot \text{inv}(B)$ has entries in $R$.

Implementation ==> add

$\text{UpTriBddDenomInv}(A, \text{denom}) = $

$AI := \text{zero}(\text{nrows} \, A, \text{nrows} \, A) \, M$

$\text{offset} := \text{minColIndex} \, AI - \text{minRowIndex} \, AI$

for $i$ in $\text{minRowIndex} \, AI$ .. $\text{maxRowIndex} \, AI$

for $j$ in $\text{minColIndex} \, AI$ .. $\text{maxColIndex} \, AI$

$qsetelt_!(AI,i,j,(\text{denom exquo} \, qelt(A,i,j)) :: R)$

for $i$ in $\text{minRowIndex} \, AI$ .. $\text{maxRowIndex} \, AI$

for $j$ in $\text{offset} + i + 1$ .. $\text{maxColIndex} \, AI$

$qsetelt_!(AI,i,j, - (\text{+} \text{sum} \, \text{qelt}(AI,i,k) \cdot \text{qelt}(A,k-\text{offset},j) \text{ for } k \text{ in } i-\text{offset}..(j-1))))$

$\text{exquo} \, qelt(A, j-\text{offset}, j) :: R))$

$AI$

$\text{LowTriBddDenomInv}(A, \text{denom}) = $

$AI := \text{zero}(\text{nrows} \, A, \text{nrows} \, A) \, M$

$\text{offset} := \text{minColIndex} \, AI - \text{minRowIndex} \, AI$

for $i$ in $\text{minRowIndex} \, AI$ .. $\text{maxRowIndex} \, AI$

for $j$ in $\text{minColIndex} \, AI$ .. $\text{maxColIndex} \, AI$

$qsetelt_!(AI,i,j,(\text{denom exquo} \, qelt(A,i,j)) :: R)$

for $i$ in $\text{minRowIndex} \, AI$ .. $\text{maxRowIndex} \, AI$

for $j$ in $i-\text{offset} + 1$ .. $\text{maxColIndex} \, AI$

$qsetelt_!(AI,j,i, - (\text{+} \text{sum} \, \text{qelt}(AI,j,k+\text{offset}) \cdot \text{qelt}(AI,k,i) \text{ for } k \text{ in } i-\text{offset}..(j-1))))$

$\text{exquo} \, qelt(A, j, j+\text{offset}) :: R))$

$AI$
package TRIGMNIP TrigonometricManipulations

--- TrigonometricManipulations.input ---

)set break resume
)sys rm -f TrigonometricManipulations.output
)spool TrigonometricManipulations.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TrigonometricManipulations

--R
--R TrigonometricManipulations (R: Join(GcdDomain,OrderedSet,RetractableTo(Integer),LinearlyExplicitRingOver(Integer)),F: Join(AlgebraicallyClosedField,TranscendentalFunctionCategory,FunctionSpace(R))) is a package constructor

--R Abbreviation for TrigonometricManipulations is TRIGMNIP

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TRIGMNIP

--R

--R---------------------------------- Operations ----------------------------------

--R complexElementary : F -> F complexElementary : (F,Symbol) -> F

--R complexForm : F -> Complex(F) complexNormalize : F -> F

--R complexNormalize : (F,Symbol) -> F imag : F -> F

--R real : F -> F real? : F -> Boolean

--R trigs : F -> F

--R

--E 1

)spool
)lisp (bye)
TrigonometricManipulations (TRIGMNIP)

Exports:
complexElementary complexForm complexNormalize imag real real? trigs

— package TRIGMNIP TrigonometricManipulations —

)abbrev package TRIGMNIP TrigonometricManipulations
++ Author: Manuel Bronstein
++ Date Created: 4 April 1988
++ Date Last Updated: 14 February 1994
++ Description:
++ \spadtype{TrigonometricManipulations} provides transformations from
++ trigonometric functions to complex exponentials and logarithms, and back.

TrigonometricManipulations(R, F): Exports == Implementation where
R : Join(GcdDomain, OrderedSet, RetractableTo Integer, LinearlyExplicitRingOver Integer)
F : Join(AlgebraicallyClosedField, TranscendentalFunctionCategory, FunctionSpace R)

Z ==> Integer
SY ==> Symbol
K ==> Kernel F
FG ==> Expression Complex R

Exports ==> with
complexNormalize: F -> F
  ++ complexNormalize(f) rewrites \spad{f} using the least possible number
  ++ of complex independent kernels.
complexNormalize: (F, SY) -> F
  ++ complexNormalize(f, x) rewrites \spad{f} using the least possible
  ++ number of complex independent kernels involving \spad{x}.
complexElementary: F -> F
  ++ complexElementary(f) rewrites \spad{f} in terms of the 2 fundamental
  ++ complex transcendental elementary functions: \spad{log, exp}.
complexElementary: (F, SY) -> F
  ++ complexElementary(f, x) rewrites the kernels of \spad{f} involving
  ++ \spad{x} in terms of the 2 fundamental complex
  ++ transcendental elementary functions: \spad{log, exp}.
trigs : F -> F
  ++ trigs(f) rewrites all the complex logs and exponentials
  ++ appearing in \spad{f} in terms of trigonometric functions.
real : F -> F
  ++ real(f) returns the real part of \spad{f} where \spad{f} is a complex
  ++ function.
imag : F -> F
  ++ imag(f) returns the imaginary part of \spad{f} where \spad{f}
  ++ is a complex function.
real? : F -> Boolean
  ++ real?(f) returns \spad{true} if \spad{f = real f}.
complexForm: F -> Complex F
  ++ complexForm(f) returns \spad{[real f, imag f]}.

Implementation ==> add
import ElementaryFunctionSign(R, F)
import InnerTrigonometricManipulations(R,F,FG)
import ElementaryFunctionStructurePackage(R, F)
import ElementaryFunctionStructurePackage(Complex R, FG)

s1 := sqrt(-1::F)
ipi := pi()$F * s1
K2KG : K -> Kernel FG
kcomplex : K -> Union(F, "failed")
locexplogs : F -> FG
localexplogs : (F, F, List SY) -> FG
complexKernels: F -> Record(ker: List K, val: List F)

K2KG k == retract(tan F2FG first argument k)@Kernel(FG)
real? f == empty?(complexKernels(f).ker)
real f == real complexForm f
imag f == imag complexForm f

-- returns [[k1,...,kn], [v1,...,vn]] such that ki should be replaced by vi
complexKernels f ==
  lk:List(K) := empty()
  lv:List(F) := empty()
  for k in tower f repeat
    if (u := kcomplex k) case F then
      lk := concat(k, lk)
      lv := concat(u::F, lv)
  [lk, lv]

-- returns f if it is certain that k is not a real kernel and k = f,
-- "failed" otherwise
kcomplex k ==
  op := operator k
  is?(k, "nthRoot"::SY) =>
    arg := argument k
    even?(retract(n := second arg)@Z) and ((u := sign(first arg)) case Z)
    and (u::Z < 0) => op(s1, n / 2::F) * op(- first arg, n)
    "failed"
  is?(k, "log"::SY) and ((u := sign(a := first argument k)) case Z)
    and (u::Z < 0) => op(- a) + ipi
  "failed"

complexForm f ==
  empty?((l := complexKernels f).ker) => complex(f, 0)
  explogs2trigs locexplogs eval(f, l.ker, l.val)

loclexplogs f ==
  any?(x +-> has?(x, "rtrig"),
    operators(g := realElementary f))$List(BasicOperator) =>
    localexplogs(f, g, variables g)
  F2FG g
complexNormalize(f, x) ==
  any?(y +-> has?(operator y, "rtrig"),
    [k for k in tower(g := realElementary(f, x))
      | member?(x, variables(k::F))]$List(K))$List(K) =>
    F2FG(rischNormalize(locallexplogs(f, g, [x]), x).func)
  rischNormalize(g, x).func

complexNormalize f ==
  l := variables(g := realElementary f)
any?(x +-> has?(x, "rtrig"), operators g)$List(BasicOperator) =>
  h := localexplogs(f, g, l)
  for x in l repeat h := rischNormalize(h, x).func
  FG2F h
  for x in l repeat g := rischNormalize(g, x).func
  g

complexElementary(f, x) ==
  any?(y +-> has?(operator y, "rtrig"),
      [k for k in tower(g := realElementary(f, x))
           | member?(x, variables(k::F))]$List(K))$List(K) =>
  FG2F localexplogs(f, g, [x])
  g

complexElementary f ==
  any?(x +-> has?(x, "rtrig"),
       operators(g := realElementary f))$List(BasicOperator) =>
  FG2F localexplogs(f, g, variables g)
  g

localexplogs(f, g, lx) ==
  trigs2explogs(F2FG g, [K2KG k for k in tower f
     | is?(k, "tan"::SY) or is?(k, "cot"::SY)], lx)

trigs f ==
  real? f => f
  g := explogs2trigs F2FG f
  real g + s1 * imag g

package TUBETOOL TubePlotTools
--- TubePlotTools.input ---

)set break resume
)sys rm -f TubePlotTools.output
)spool TubePlotTools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TubePlotTools
--R
--R TubePlotTools is a package constructor
--R Abbreviation for TubePlotTools is TUBETOOL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TUBETOOL
--R
--R------------------------------- Operations --------------------------------
--R ?*? : (DoubleFloat,Point(DoubleFloat)) -> Point(DoubleFloat)
--R ?+? : (Point(DoubleFloat),Point(DoubleFloat)) -> Point(DoubleFloat)
--R ?-? : (Point(DoubleFloat),Point(DoubleFloat)) -> Point(DoubleFloat)
--R cosSinInfo : Integer -> List(List(DoubleFloat))
--R cross : (Point(DoubleFloat),Point(DoubleFloat)) -> Point(DoubleFloat)
--R dot : (Point(DoubleFloat),Point(DoubleFloat)) -> DoubleFloat
--R loopPoints : (Point(DoubleFloat),Point(DoubleFloat),Point(DoubleFloat),DoubleFloat,List(List(DoubleFloat))) -> List(Point(DoubleFloat))
--R point : (DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat) -> Point(DoubleFloat)
--R
--E 1

)spool
)lisp (bye)

---

--- TubePlotTools.help ---

====================================================================
TubePlotTools examples
====================================================================

Tools for constructing tubes around 3-dimensional parametric curves.

See Also:
  o )show TubePlotTools
TubePlotTools (TUBETOOL)

Exports:
- cosSinInfo
- cross
- dot
- loopPoints
- point
- unitVector
- ?*?
- ?+?
- ?-?

--- package TUBETOOL TubePlotTools ---

)abbrev package TUBETOOL TubePlotTools
++ Author: Clifton J. Williamson
++ Date Created: Bastille Day 1989
++ Date Last Updated: 5 June 1990
++ Description:
++ Tools for constructing tubes around 3-dimensional parametric curves.

TubePlotTools(): Exports == Implementation where
  I ==> Integer
  SF ==> DoubleFloat
  L ==> List
  Pt ==> Point SF

Exports ==> with
  point: (SF,SF,SF,SF) -> Pt
  ++ point(x1,x2,x3,c) creates and returns a point from the three
  ++ specified coordinates \spad{x1}, \spad{x2}, \spad{x3}, and also
  ++ a fourth coordinate, c, which is generally used to specify the
  ++ color of the point.
  "*" : (SF,Pt) -> Pt
  ++ s * p returns a point whose coordinates are the scalar multiple
  ++ of the point p by the scalar s, preserving the color, or fourth
  ++ coordinate, of p.
  "+" : (Pt,Pt) -> Pt
  ++ p + q computes and returns a point whose coordinates are the sums
  ++ of the coordinates of the two points \spad{p} and \spad{q}, using
  ++ the color, or fourth coordinate, of the first point \spad{p}
  ++ as the color also of the point \spad{q}.
  "-" : (Pt,Pt) -> Pt
CHAPTER 21. CHAPTER T

++ $p - q$ computes and returns a point whose coordinates are the
++ differences of the coordinates of two points \spad{p} and \spad{q},
++ using the color, or fourth coordinate, of the first point \spad{p}
++ as the color also of the point \spad{q}.

dot : (Pt,Pt) -> SF
++ \{dot(p,q)\} computes the dot product of the two points \spad{p}
++ and \spad{q} using only the first three coordinates, and returns
++ the resulting \spadtype{DoubleFloat}.

cross : (Pt,Pt) -> Pt
++ \{cross(p,q)\} computes the cross product of the two points \spad{p}
++ and \spad{q} using only the first three coordinates, and keeping
++ the color of the first point \spad{p}. The result is returned as a point.

unitVector: Pt -> Pt
++ \{unitVector(p)\} creates the unit vector of the point \spad{p} and returns
++ the result as a point. Note that \spad{unitVector(p) = p/|p|}.

cosSinInfo: I -> L L SF
++ \{cosSinInfo(n)\} returns the list of lists of values for \spad{n}, in the form
++ \{\spad{[[cos(n-1) a,sin(n-1) a],...,[cos 2 a,sin 2 a],[cos a,sin a]]}\}
++ where \spad{a = 2 pi/n}. Note that \spad{n} should be greater than 2.

loopPoints: (Pt,Pt,Pt,SF,L L SF) -> L Pt
++ \{loopPoints(p,n,b,r,lls)\} creates and returns a list of points
++ which form the loop with radius \spad{r}, around the center point
++ indicated by the point \spad{p}, with the principal normal vector of
++ the space curve at point \spad{p} given by the point\{vector\} \spad{n}, and the
++ binormal vector given by the point\{vector\} \spad{b}, and a list of lists,
++ \spad{lls}, which is the \spadfun{cosSinInfo} of the number of points
++ defining the loop.

Implementation ==> add

import PointPackage(SF)

point(x,y,z,c) == point(l : L SF := [x,y,z,c])

g getColor: Pt -> SF
getColor pt == (\maxIndex pt > 3 => color pt; 0)

g getColor2: (Pt,Pt) -> SF
getColor2(p0,p1) ==
  \maxIndex p0 > 3 => color p0
  \maxIndex p1 > 3 => color p1
  0

a * p ==
l : L SF := [a * xCoord p,a * yCoord p,a * zCoord p,getColor p]
point l

p0 + p1 ==
l : L SF := [xCoord p0 + xCoord p1,yCoord p0 + yCoord p1,_
zCoord p0 + zCoord p1,getColor2(p0,p1)]
point l
p0 - p1 ==  
  l : L SF := [xCoord p0 - xCoord p1, yCoord p0 - yCoord p1, _  
     zCoord p0 - zCoord p1, getColor2(p0, p1)]  
  point l

dot(p0, p1) ==  
  (xCoord p0 * xCoord p1) + (yCoord p0 * yCoord p1) + _  
     (zCoord p0 * zCoord p1)

cross(p0, p1) ==  
  x0 := xCoord p0; y0 := yCoord p0; z0 := zCoord p0;  
  x1 := xCoord p1; y1 := yCoord p1; z1 := zCoord p1;  
  l : L SF := [y0 * z1 - y1 * z0, z0 * x1 - z1 * x0, _  
     x0 * y1 - x1 * y0, getColor2(p0, p1)]  
  point l

unitVector p == (inv sqrt dot(p, p)) * p

cosSinInfo n ==  
  ans : L L SF := nil()  
  theta : SF := 2 * pi() / n  
  for i in 1..(n-1) repeat  
    angle := i * theta  
    ans := concat([cos angle, sin angle], ans)  
  ans

loopPoints(ctr, pNorm, bNorm, rad, cosSin) ==  
  ans : L Pt := nil()  
  while not null cosSin repeat  
    cossin := first cosSin; cos := first cossin; sin := second cossin  
    ans := cons(ctr + rad * (cos * pNorm + sin * bNorm), ans)  
    cossin := rest cossin  
    pt := ctr + rad * pNorm  
    concat(pt, concat(ans, pt))

— TUBETOOL.dotabb —

"TUBETOOL" [color="#FF4488", href="bookvol10.4.pdf#nameddest=TUBETOOL"]  
"FRAC" [color="#88FF44", href="bookvol10.3.pdf#nameddest=FRAC"]  
"TUBETOOL" -> "FRAC"
package CLIP TwoDimensionalPlotClipping

--- TwoDimensionalPlotClipping.input ---

)set break resume
/sys rm -f TwoDimensionalPlotClipping.output
/spool TwoDimensionalPlotClipping.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TwoDimensionalPlotClipping
--R
--R TwoDimensionalPlotClipping is a package constructor
--R Abbreviation for TwoDimensionalPlotClipping is CLIP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for CLIP
--R
--R------------------------------- Operations --------------------------------
--R clip : Plot -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R clip : (Plot,Fraction(Integer),Fraction(Integer)) -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R clip : List(Point(DoubleFloat)) -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R clip : List(List(Point(DoubleFloat))) -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R clipParametric : Plot -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R clipParametric : (Plot,Fraction(Integer),Fraction(Integer)) -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R clipWithRanges : (List(List(Point(DoubleFloat))),DoubleFloat,DoubleFloat,DoubleFloat,DoubleFloat) -> Record(brans: List(List(Point(DoubleFloat))),xValues: Segment(DoubleFloat),yValues: Segment(DoubleFloat))
--R
--E 1

)spool
)lisp (bye)

--- TwoDimensionalPlotClipping.help ---

TwoDimensionalPlotClipping examples

Automatic clipping for 2-dimensional plots. The purpose of this package is to provide reasonable plots of functions with singularities.

See Also:
  o )show TwoDimensionalPlotClipping
TwoDimensionalPlotClipping (CLIP)

Exports:
clip  clipParametric  clipWithRanges

--- package CLIP TwoDimensionalPlotClipping ---

)abbrev package CLIP TwoDimensionalPlotClipping
++ Author: Clifton J. Williamson
++ Date Created: 22 December 1989
++ Date Last Updated: 10 July 1990
++ Description:
++ Automatic clipping for 2-dimensional plots
++ The purpose of this package is to provide reasonable plots of
++ functions with singularities.

TwoDimensionalPlotClipping(): Exports == Implementation where
B == Boolean
L == List
SEG == Segment
RN == Fraction Integer
SF == DoubleFloat
Pt == Point DoubleFloat
PLOT == Plot
CLIPPED == Record(brans: L L Pt, xValues: SEG SF, yValues: SEG SF)

Exports == with
clip: PLOT -> CLIPPED
++ clip(p) performs two-dimensional clipping on a plot, p, from
++ the domain \spad{Plot} for the graph of one variable,
++ \spad{\text{spad}(y = f(x))}; the default parameters \spad{1/4} for the fraction
++ and \spad{5/1} for the scale are used in the \spad{\text{spadfun}(clip)} function.
clip: (PLOT,RW,RN) -> CLIPPED
++ clip(p,frac,sc) performs two-dimensional clipping on a plot, p,
++ from the domain \spadtype{Plot} for the graph of one variable
++ \( y = f(x) \); the fraction parameter is specified by \spad{frac}
++ and the scale parameter is specified by \spad{sc} for use in the
++ \spadfun{clip} function.

clipParametric: PLOT -> CLIPPED
++ clipParametric(p) performs two-dimensional clipping on a plot,
++ p, from the domain \spadtype{Plot} for the parametric curve
++ \( x = f(t), \ y = g(t) \); the default parameters \spad{1/2}
++ for the fraction and \spad{5/1} for the scale are used in the
++ \fakeAxiomFun{iClipParametric} subroutine, which is called by this
++ function.

clipParametric: (PLOT,RN,RN) -> CLIPPED
++ clipParametric(p,frac,sc) performs two-dimensional clipping on a
++ plot, p, from the domain \spadtype{Plot} for the parametric curve
++ \( x = f(t), \ y = g(t) \); the fraction parameter is
++ specified by \spad{frac} and the scale parameter is specified
++ by \spad{sc} for use in the \fakeAxiomFun{iClipParametric} subroutine,
++ which is called by this function.

clipWithRanges: (L L Pt,SF,SF,SF,SF) -> CLIPPED
++ clipWithRanges(pointLists,xMin,xMax,yMin,yMax) performs clipping
++ on a list of lists of points, \spad{pointLists}. Clipping is
++ done within the specified ranges of \spad{xMin}, \spad{xMax} and
++ \spad{yMin}, \spad{yMax}. This function is used internally by
++ the \fakeAxiomFun{iClipParametric} subroutine in this package.

clip: L Pt -> CLIPPED
++ clip(l) performs two-dimensional clipping on a curve l, which is
++ a list of points; the default parameters \spad{1/2} for
++ the fraction and \spad{5/1} for the scale are used in the
++ \fakeAxiomFun{iClipParametric} subroutine, which is called by this
++ function.

clip: L L Pt -> CLIPPED
++ clip(l1) performs two-dimensional clipping on a list of lists
++ of points, \spad{ll}; the default parameters \spad{1/2} for
++ the fraction and \spad{5/1} for the scale are used in the
++ \fakeAxiomFun{iClipParametric} subroutine, which is called by this
++ function.

Implementation ==> add
import PointPackage(DoubleFloat)
import ListFunctions2(Point DoubleFloat,DoubleFloat)

point:(SF,SF) -> Pt
intersectWithHorizLine:(SF,SF,SF,SF,SF) -> Pt
intersectWithVertLine:(SF,SF,SF,SF,SF) -> Pt
intersectWithBdry:(SF,SF,SF,SF,Pt,Pt) -> Pt
discardAndSplit: (L Pt,Pt -> B,SF,SF,SF,SF) -> L L Pt
norm: Pt -> SF
iClipParametric: (L L Pt,RN,RN) -> CLIPPED
findPt: L L Pt -> Union(Pt,"failed")
Fnan?: SF ->Boolean
Pnan?:Pt ->Boolean

Fnan? x == x~=x
Pnan? p == any?(Fnan?,p)

iClipParametric(pointLists,fraction,scale) ==
  -- error checks and special cases
  (fraction < 0) or (fraction > 1) =>
  error "clipDraw: fraction should be between 0 and 1"
empty? pointLists => [nil(),segment(0,0),segment(0,0)]
  -- put all points together , sort them according to norm
  sortedList := sort((x:Pt,y:Pt):Boolean +-> norm(x) < norm(y),
    select((z:Pt):Boolean +-> not Pnan? z,concat pointLists))
empty? sortedList => [nil(),segment(0,0),segment(0,0)]
  n := # sortedList
  num := numer fraction
  den := denom fraction
  clipNum := (n * num) quo den
  lastN := n - 1 - clipNum
  firstPt := first sortedList
  xMin : SF := xCoord firstPt
  xMax : SF := xCoord firstPt
  yMin : SF := yCoord firstPt
  yMax : SF := yCoord firstPt
  -- calculate min/max for the first (1-fraction)*N points
  -- this contracts the range
  -- this unnecessarily clips monotonic functions (step-function, x~(high power),etc.)
  for k in 0..lastN for pt in rest sortedList repeat
    xMin := min(xMin,xCoord pt)
    xMax := max(xMax,xCoord pt)
    yMin := min(yMin,yCoord pt)
    yMax := max(yMax,yCoord pt)
    xDiff := xMax - xMin; yDiff := yMax - yMin
    xDiff = 0 =>
      yDiff = 0 =>
        [pointLists,segment(xMin-1,xMax+1),segment(yMin-1,yMax+1)]
        [pointLists,segment(xMin-1,xMax+1),segment(yMin,yMax)]
      yDiff = 0 =>
        [pointLists,segment(xMin,xMax),segment(yMin-1,yMax+1)]
    numm := numer scale; denn := denom scale
    -- now expand the range by scale
    xMin := xMin - (numm :: SF) * xDiff / (denn :: SF)
    xMax := xMax + (numm :: SF) * xDiff / (denn :: SF)
    yMin := yMin - (numm :: SF) * yDiff / (denn :: SF)
    yMax := yMax + (numm :: SF) * yDiff / (denn :: SF)
    -- clip with the calculated range
    newclip:=clipWithRanges(pointLists,xMin,xMax,yMin,yMax)
    -- if we split the lists use the new clip
# (newclip.brans) > # pointLists => newclip

-- calculate extents
xs : L SF := map (xCoord, sortedList)
ys : L SF := map (yCoord, sortedList)
xMin : SF := reduce (min, xs)
yMin : SF := reduce (min, ys)
xMax : SF := reduce (max, xs)
yMax : SF := reduce (max, ys)
xseg : SEG SF := xMin..xMax
yseg : SEG SF := yMin..yMax

-- return original
[pointLists, xseg, yseg]@CLIPPED

point(xx, yy) == point(l : L SF := [xx, yy])

intersectWithHorizLine(x1, y1, x2, y2, yy) ==
  x1 = x2 => point(x1, yy)
  point(x1 + (x2 - x1)*(yy - y1)/(y2 - y1), yy)

intersectWithVertLine(x1, y1, x2, y2, xx) ==
  y1 = y2 => point(xx, y1)
  point(xx, y1 + (y2 - y1)*(xx - x1)/(x2 - x1))

intersectWithBdry(xMin, xMax, yMin, yMax, pt1, pt2) ==
  -- pt1 is in rectangle, pt2 is not
  x1 := xCoord pt1; y1 := yCoord pt1
  x2 := xCoord pt2; y2 := yCoord pt2
  if y2 > yMax then
    pt2 := intersectWithHorizLine(x1, y1, x2, y2, yMax)
    x2 := xCoord pt2; y2 := yCoord pt2
  if y2 < yMin then
    pt2 := intersectWithHorizLine(x1, y1, x2, y2, yMin)
    x2 := xCoord pt2; y2 := yCoord pt2
  if x2 > xMax then
    pt2 := intersectWithVertLine(x1, y1, x2, y2, xMax)
    x2 := xCoord pt2; y2 := yCoord pt2
  if x2 < xMin then
    pt2 := intersectWithVertLine(x1, y1, x2, y2, xMin)
    pt2

discardAndSplit(pointList, pred, xMin, xMax, yMin, yMax) ==
  ans : L L Pt := nil()
  list : L Pt := nil()
  lastPt? : B := false
  lastPt : Pt := point(0, 0)
  while not empty? pointList repeat
    pt := first pointList
pointList := rest pointList
pred(pt) =>
  if (empty? list) and lastPt? then
    bdryPt := intersectWithBdry(xMin,xMax,yMin,yMax,pt,lastPt)
    -- print bracket [ coerce bdryPt ,coerce pt ]
    --list := cons(bdryPt,list)
    list := cons(pt,list)
  if not empty? list then
    bdryPt := intersectWithBdry(xMin,xMax,yMin,yMax,first list,pt)
    -- print bracket [ coerce bdryPt ,coerce first list ]
    --list := cons(bdryPt,list)
    ans := cons(list,ans)
  lastPt := pt
  lastPt? := true
  list := nil()
empty? list => ans
reverse_! cons(reverse_! list,ans)

clip(plot,fraction,scale) ==
  -- sayBrightly([" clip: ":OutForm]$List(OutForm))$Lisp
  (fraction < 0) or (fraction > 1/2) =>
    error "clipDraw: fraction should be between 0 and 1/2"
  xVals := xRange plot
empty?(pointLists := listBranches plot) =>
  [nil(),xVals,segment(0,0)]
more?(pointLists := listBranches plot,1) =>
  error "clipDraw: plot has more than one branch"
empty?(pointList := first pointLists) =>
  [nil(),xVals,segment(0,0)]
  sortedList := sort((x,y)+->yCoord(x) < yCoord(y),pointList)
  n := # sortedList; num := numer fraction; den := denom fraction
  clipNum := (n * num) quo den
  -- throw out points with large and small y-coordinates
  yMin := yCoord(sortedList.clipNum)
yMax := yCoord(sortedList.(n - 1 - clipNum))
  if Fnan? yMin then yMin := 0
  if Fnan? yMax then yMax := 0
  (yDiff := yMax - yMin) = 0 =>
    [pointLists,xRange plot,segment(yMin - 1,yMax + 1)]
  numm := numer scale; denn := denom scale
  xMin := lo xVals; xMax := hi xVals
  yMin := yMin - (numm :: SF) * yDiff / (denn :: SF)
yMax := yMax + (numm :: SF) * yDiff / (denn :: SF)
  lists := discardAndSplit(pointList,_,
    x +-> (yCoord(x) < yMax) and (yCoord(x) > yMin),
    xMin,xMax,yMin,yMax)
yMin := yCoord(sortedList.clipNum)
yMax := yCoord(sortedList.(n - 1 - clipNum))
  if Fnan? yMin then yMin := 0
  if Fnan? yMax then yMax := 0
for list in lists repeat
    for pt in list repeat
        if not Fnan?(yCoord pt) then
            yMin := min(yMin,yCoord pt)
            yMax := max(yMax,yCoord pt)
        [lists,xVals,segment(yMin,yMax)]

clip(plot: PLOT) == clip(plot, 1/4, 5/1)

norm(pt) ==
    x := xCoord(pt); y := yCoord(pt)
    if Fnan? x then
        if Fnan? y then
            r: SF := 0
        else
            r: SF := y**2
    else
        if Fnan? y then
            r: SF := x**2
        else
            r: SF := x**2 + y**2
    r

findPt lists ==
    for list in lists repeat
        not empty? list =>
            for p in list repeat
                not Pnan? p => return p
        "failed"

clipWithRanges(pointLists, xMin, xMax, yMin, yMax) ==
    lists := L L Pt := nil()
    for pointList in pointLists repeat
        lists := concat(lists, discardAndSplit(pointList, (x:Pt):Boolean +-> (xCoord(x) <= xMax) and (xCoord(x) >= xMin) and (yCoord(x) <= yMax) and (yCoord(x) >= yMin), xMin, xMax, yMin, yMax))
    (pt := findPt lists) case "failed" =>
        [nil(), segment(0,0), segment(0,0)]
    firstPt := pt :: Pt
    xMin : SF := xCoord firstPt; xMax : SF := xCoord firstPt
    yMin : SF := yCoord firstPt; yMax : SF := yCoord firstPt
    for list in lists repeat
        for pt in list repeat
            if not Pnan? pt then
                xMin := min(xMin,xCoord pt)
                xMax := max(xMax,xCoord pt)
                yMin := min(yMin,yCoord pt)
                yMax := max(yMax,yCoord pt)
        [lists, segment(xMin,xMax), segment(yMin,yMax)]
clipParametric(plot,fraction,scale) ==
   iClipParametric(listBranches plot,fraction,scale)

clipParametric plot == clipParametric(plot,1/2,5/1)

clip(l: L Pt) == iClipParametric(list 1,1/2,5/1)
clip(l: L L Pt) == iClipParametric(1,1/2,5/1)

---

--- CLIP.dotabb ---

"CLIP" [color="#FF4488",href="bookvol10.4.pdf#nameddest=CLIP"]
"PTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PTCAT"]
"CLIP" -> "PTCAT"

---

package TWOFACT TwoFactorize

--- TwoFactorize.input ---

)set break resume
)sys rm -f TwoFactorize.output
)spool TwoFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show TwoFactorize
--R
--R TwoFactorize(F: FiniteFieldCategory) is a package constructor
--R Abbreviation for TwoFactorize is TWOFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for TWOFACT
--R
--R------------------------------------------------------------ Operations --------------------------------
--R generalSqFr : SparseUnivariatePolynomial(SparseUnivariatePolynomial(F)) -> Factored(SparseUnivariatePolynomial(F))
--R generalTwoFactor : SparseUnivariatePolynomial(SparseUnivariatePolynomial(F)) -> Factored(SparseUnivariatePolynomial(F))
--R twoFactor : (SparseUnivariatePolynomial(SparseUnivariatePolynomial(F)),Integer) -> Factored(SparseUnivariatePolynomial(F))
--R
--E 1
TwoFactorize (TWOFACT)

Exports:
  generalSqFr  generalTwoFactor  twoFactor

— package TWOFACT TwoFactorize —

)abbrev package TWOFACT TwoFactorize
++ Authors : P.Gianni, J.H.Davenport
++ Date Created : May 1990
++ Date Last Updated: March 1992
++ Description:
++ A basic package for the factorization of bivariate polynomials
++ over a finite field.
++ The functions here represent the base step for the multivariate factorizer.

TwoFactorize(F) : C == T
where
  F      : FiniteFieldCategory
  SUP    ==> SparseUnivariatePolynomial
  R      ==> SUP F
  P      ==> SUP R
  UPCF2  ==> UnivariatePolynomialCategoryFunctions2

C == with
  generalTwoFactor   : (P) -> Factored P
  generalSqFr        : (P) -> Factored P
  twoFactor          : (P,Integer) -> Factored P

T == add
  PI == PositiveInteger
  NNI == NonNegativeInteger
import CommuteUnivariatePolynomialCategory(F,R,P)

---- Local Functions ----
computeDegree : (P,Integer,Integer) -> PI
exchangeVars : P -> P
exchangeVarTerm: (R, NNI) -> P
pthRoot : (R, NNI, NNI) -> R

-- compute the degree of the extension to reduce the polynomial to a
-- univariate one
computeDegree(m : P, mx:Integer, q:Integer): PI ==
  my:=degree m
  n1:Integer:=length(10*mx*my)
  n2:Integer:=length(q)-1
  n:=(n1 quo n2)+1
  n::PI
  -- n=1 => 1$PositiveInteger
-- (nextPrime(max(n,min(mx,my)))$IntegerPrimesPackage(Integer))::PI

exchangeVars(p : P) : P ==
p = 0 => 0
exchangeVarTerm(leadingCoefficient p, degree p) +
exchangeVars(reductum p)

exchangeVarTerm(c:R, e:NNI) : P ==
c = 0 => 0
monomial(monomial(leadingCoefficient c, e)$R, degree c)$P +
exchangeVarTerm(reductum c, e)

tmp:=divideExponents(map((x:F):F+->(x::F)**PthRootPow,poly),p)
tmp case "failed" => error "consistency error in TwoFactor"
tmp

fUnion ==> Union("nil", "sqfr", "irred", "prime")
FF ==> Record(flg:fUnion, fctr:P, xpnt:Integer)

generalSqFr(m:P): Factored P ==
m = 0 => 0
degree m = 0 =>
l:=squareFree(leadingCoefficient m)
makeFR(unit(l)::P,[[u.flg,u.fctr::P,u.xpnt] for u in factorList l])
cont := content m
m := (m exquo cont)::P
sqfrm := squareFree m
pfaclist : List FF := empty()
unitPart := unit sqfrm
for u in factorList sqfrm repeat
  u.flg = "nil" =>
    uexp:NNI:=(u.xpnt):NNI
    nfacs:=squareFree(exchangeVars u.fctr)
    for v in factorList nfacs repeat
      pfaclist:=cons([v.flg, exchangeVars v.fctr, v.xpnt*uexp],
        pfaclist)
      unitPart := unit(nfacs)**uexp * unitPart
      pfaclist := cons(u,pfaclist)
  cont ^= 1 =>
    sqp := squareFree cont
    contlist:=[[w.flg,(w.fctr)::P,w.xpnt] for w in factorList sqp]
    pfaclist:= append(contlist, pfaclist)
    makeFR(unit(sqp)*unitPart, pfaclist)
    makeFR(unitPart, pfaclist)

generalTwoFactor(m:P): Factored P ==
m = 0 => 0
degree m = 0 =>
l:=factor(leadingCoefficient m)$DistinctDegreeFactorize(F,R)
makeFR(unit(l)::P,[[u.flg,u.fctr::P,u.xpnt] for u in factorList l])
ll:=List FF
ll:=[[]
unitPart:P
cont:=content m
if degree(cont)>0 then
m1:=m exquo cont
m1 case "failed" => error "content doesn't divide"
m:=m1
contfact:=factor(cont)$DistinctDegreeFactorize(F,R)
unitPart:=(unit contfact)::P
ll:=[[w.flg,(w.fctr)::P,w.xpnt] for w in factorList contfact]
else
unitPart:=cont::P
sqfrm:=squareFree m
for u in factors sqfrm repeat
expo:=u.exponent
if expo < 0 then error "negative exponent in a factorisation"
expon:NonNegativeInteger:=expo::NonNegativeInteger
fac:=u.factor
degree fac = 1 => ll:=["irred",fac,expon],ll]
differentiate fac = 0 =>
-- the polynomial is inseparable w.r.t. its main variable
map(differentiate,fac) = 0 =>
p:=characteristic$F
PthRootPow:=(size$F exquo p)::NonNegativeInteger
m1:=divideExponents(map(x+->pthRoot(x,p,PthRootPow),fac),p)
m1 case "failed" => error "consistency error in TwoFactor"
res:=generalTwoFactor m1
unitPart:=unitPart*unit(res)**((p*expon)::NNI)
ll:=
[:[[v.flg,v.fctr,expon*p*v.xpnt] for v in factorList res],ll]
m2:=generalTwoFactor swap fac
unitPart:=unitPart*unit(m2)**(expon::NNI)
ll:=
[:[[v.flg,swap v.fctr,expon*v.xpnt] for v in factorList m2],ll]
ydeg:"max"/[degree w for w in coefficients fac]
twoF:=twoFactor(fac,ydeg)
unitPart:=unitPart*unit(twoF)**expon
ll:=[[[v.flg,v.fctr,expon*v.xpnt] for v in factorList twoF],
:ll]
makeFR(unitPart,11)

-- factorization of a primitive square-free bivariate polynomial --
twoFactor(m:P,dx:Integer):Factored P ==
-- choose the degree for the extension
n:PI:=computeDegree(m,dx,size()$F)
-- extend the field
-- find the substitution for x
look:Boolean:=true
dm:=degree m
try:Integer:=min(5,size()$F)
i:Integer:=0
lcm := leadingCoefficient m
umv : R
while look and i < try repeat
   vval := random()$F
   i:=i+1
   zero? elt(lcm, vval) => "next value"
   umv := map(x +-> elt(x,vval), m)$UPCF2(R, P, F, R)
   degree(gcd(umv,differentiate umv))"=0 => "next val"
   n := 1
   look := false
extField:=FiniteFieldExtension(F,n)
SUEx:=SUP extField
TP:=SparseUnivariatePolynomial SUEX
mm:TP:=0
m1:=m
while m1="=0 repeat
   mm:=mm+monomial(map(coerce,leadingCoefficient m1)$UPCF2(F,R,
extField,SUEX),degree m1)
   m1:=reductum m1
lcm := leadingCoefficient mm
val : extField
umex : SUEX
if not look then
   val := vval :: extField
   umex := map(coerce, umv)$UPCF2(F, R, extField, SUEX)
while look repeat
   val:=random()$extField
   i:=i+1
   elt(lcm, val)=0 => "next value"
   umex := map(x +-> elt(x,val), mm)$UPCF2(SUEX, TP, extField, SUEX)
   degree(gcd(umex,differentiate umex))"=0 => "next val"
   look:=false
prime:SUEx:=monomial(1,1)-monomial(val,0)
fumex:=factor(umex)$DistinctDegreeFactorize(extField,SUEX)
lfact1:=factors fumex

#lfact1=1 => primeFactor(m,1)
lfact : List TP :=
   [map(coerce,lf.factor)$UPCF2(extField,SUEX,SUEX,TP)
   for lf in lfact1]
lfact:=cons(map(coerce,unit fumex)$UPCF2(extField,SUEX,SUEX,TP),
   lfact)
import GeneralHenselPackage(SUEX,TP)
dx1:PI:=(dx+1)::PI
lfacth:=completeHensel(mm,lfact,prime,dx1)
lfactk: List P :=[]
Normp := NormRetractPackage(F, extField, SUEx, TP, n)

while not empty? lfacth repeat
  ff := first lfacth
  lfacth := rest lfacth
  if (c:=leadingCoefficient leadingCoefficient ff) ^=1 then
    ff:=((inv c)::SUEx)* ff
  not ((ffu:= retractIfCan(ff)$Normp) case "failed") =>
    lfactk := cons(ffu::P, lfactk)
  normfacs := normFactors(ff)$Normp
  lfacth := [g for g in lfacth | not member?(g, normfacs)]
  ffn := */normfacs
  lfactk:=cons(retractIfCan(ffn)$Normp :: P, lfactk)
  */[primeFactor(ff1,1) for ff1 in lfactk]
Chapter 22

Chapter U

package UNIFACT UnivariateFactorize

— UnivariateFactorize.input —

)set break resume
)sys rm -f UnivariateFactorize.output
)spool UnivariateFactorize.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateFactorize
--R
--R UnivariateFactorize(ZP: UnivariatePolynomialCategory(Integer)) is a package constructor
--R Abbreviation for UnivariateFactorize is UNIFACT
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UNIFACT
--R
--R------------------------------- Operations --------------------------------
--R factor : ZP -> Factored(ZP)
--R factorSquareFree : ZP -> Factored(ZP)
--R henselFact : (ZP,Boolean) -> Record(contp: Integer,factors: List(Record(irr: ZP,pow: Integer)))
--R
--E 1

)spool
)lisp (bye)

______

4085
--- UnivariateFactorize.help ---

UnivariateFactorize examples

Package for the factorization of univariate polynomials with integer coefficients. The factorization is done by "lifting" (HENSEL) the factorization over a finite field.

See Also:
* )show UnivariateFactorize

---

UnivariateFactorize (UNIFACT)

Exports:
factor factorSquareFree henselFact

--- package UNIFACT UnivariateFactorize ---

)abbrev package UNIFACT UnivariateFactorize
++ Author: Patrizia Gianni
++ Date Created: ???
++ Date Last Updated: December 1993
++ Description:
++ Package for the factorization of univariate polynomials with integer coefficients. The factorization is done by "lifting" (HENSEL) the factorization over a finite field.

UnivariateFactorize(ZP) : public == private where
Z ==> Integer
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
SUPZ ==> SparseUnivariatePolynomial Z

ZP : UnivariatePolynomialCategory Z

FR ==> Factored ZP
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FFE ==> Record(flg:fUnion, fctr:ZP, xpnt:Z)
ParFact ==> Record(irr: ZP,pow: Z)
FinalFact ==> Record(contp: Z,factors:List(ParFact))

public == with
factor : ZP -> FR
++ factor(m) returns the factorization of m
factorSquareFree : ZP -> FR
++ factorSquareFree(m) returns the factorization of m square free
++ polynomial
henselFact : (ZP,Boolean) -> FinalFact
++ henselFact(m,flag) returns the factorization of m,
++ FinalFact is a Record s.t. FinalFact.contp=content m,
++ FinalFact.factors=List of irreducible factors
++ of m with exponent , if flag =true the polynomial is
++ assumed square free.

private == add
--- local functions ---

henselFact : ZP -> List(ZP)
quadratic : ZP -> List(ZP)
remp : (Z, PI) -> Z
negShiftz : (Z, PI) -> Z
negShiftp : (ZP,PI) -> ZP
bound : ZP -> PI
choose : ZP -> FirstStep
eisenstein : ZP -> Boolean
isPowerOf2 : Z -> Boolean
subMinusX : SUPZ -> ZP
sqroot : Z -> Z

--- declarations ---
CYC ==> CyclotomicPolynomialPackage()
DDRecord ==> Record(factor: ZP,degree: Z)
DDList ==> List DDRecord
FirstStep ==> Record(prime:PI,factors:DDList)
ContPrim ==> Record(cont: Z,prim: ZP)

import GeneralHenselPackage(Z,ZP)
import ModularDistinctDegreeFactorizer ZP

factor(m: ZP) ==
  flist := henselFact(m,false)
  ctp:=unitNormal flist.contp
  makeFR((ctp.unit)::ZP,cons(["nil",ctp.canonical::ZP,1$Z]$FFE,
    [["prime",u.irr,u.pow]$FFE for u in flist.factors]))

factorSquareFree(m: ZP) ==
  flist := henselFact(m,true)
  ctp:=unitNormal flist.contp
  makeFR((ctp.unit)::ZP,cons(["nil",ctp.canonical::ZP,1$Z]$FFE,
    [["prime",u.irr,u.pow]$FFE for u in flist.factors]))

-- Integer square root: returns 0 if t is non-positive
sqroot(t: Z): Z ==
  t <= 0 => 0
  s:=Integer:=t::Integer
  s:=approxSqrt(s)$IntegerRoots(Integer)
  t:=s::Z
  t

-- Eisenstein criterion: returns true if polynomial is irreducible. Result of false in inconclusive.
eisenstein(m : ZP): Boolean ==
  -- calculate the content of the terms after the first
  c := content reductum m
  c = 0 => false
  c = 1 => false
  -- factor the content
  -- if there is a prime in the factorization that does not divide
  -- the leading term and appears to multiplicity 1, and the square
  -- of this does not divide the last coef, return true.
  -- Otherwise return false.
  lead := leadingCoefficient m
  trail := lead
  m := reductum m
  while m ^= 0 repeat
    trail := leadingCoefficient m
    m:= reductum m
    fc := factor(c) :: Factored(Z)
    for r in factors fc repeat
      if (r.exponent = 1) and (0 ^= (lead rem r.factor)) and
      (0 ^= (trail rem (r.factor ** 2))) then return true
    false

negShiftz(n: Z,Modulus:PI): Z ==
  if n < 0 then n := n+Modulus
n > (Modulus quo 2) => n-Modulus

negShift?(pp: ZP, Modulus: PI): ZP ==
map(x +-> negShiftz(x, Modulus), pp)

-- Choose the bound for the coefficients of factors
bound(m: ZP): PI ==
nm, nmq2, lcm, bin0, bin1: NNI
cbound, j: PI
k: NNI
lcm := abs(leadingCoefficient m):: NNI
nm := (degree m):: NNI

if nmq2^=1 then nm := (nmq2-1): NNI
else nm := nmq2
bin0 := nm
cbound := (bin0*norm+lcm):: PI
for i in 2..(nm-1):: NNI repeat
bin1 := bin0
bin0 := (bin0*(nm+1-i):: NNI) quo i
j := (bin0*norm+bin1*lcm):: PI
if cbound<j then cbound := j

(2*cbound*lcm):: PI -- adjusted by lcm to prepare for exquo in ghensel

remp(t: Z, q: PI): Z == ((t := t rem q)<0 => t+q ; t)

numFactors(ddlist: DDLList): Z ==
ans: Z := 0
for dd in ddlist repeat
(d := degree(dd.factor)) = 0 => nil
ans := ans + ((d pretend Z) exquo dd.degree):: Z
ans

-- select the prime, try up to 4 primes,
-- choose the one yielding the fewest factors, but stopping if
-- fewer than 9 factors
choose(m: ZP): FirstStep ==
qSave: PI := 1
ddSave: DDLList := []
numberOfFactors: Z := 0
lcm := leadingCoefficient m
k: Z := 1
ddRep := 5
disc: ZP := 0
q: PI := 2
while k<ddRep repeat
-- q must be a new prime number at each iteration
q := nextPrime(q)$ IntegerPrimesPackage(Z) pretend PI
(rr:=lcm rem q) = 0$Z => "next prime"
disc:=gcd(m,differentiate m,q)
(degree disc)^=0 => "next prime"
k := k+1
newdd := ddFact(m,q)
((n := numFactors(newdd)) < 9) =>
  ddSave := newdd
  qSave := q
  k := 5
(numberOfFactors = 0) or (n < numberOfFactors) =>
  ddSave := newdd
  qSave := q
  numberOfFactors := n
[qSave,ddSave]$FirstStep

-- Find the factors of m, primitive, square-free, with lc positive
-- and mindeg m = 0
henselfact1(m: ZP):List(ZP) ==
  zero? degree m =>$\$
  (m = 1) => [m]
  selected := choose(m)
  (numFactors(selected.factors) = 1$Z) => [m]
  q := selected.prime
  fl := separateFactors(selected.factors,q)
  --choose the bound
  cbound := bound(m)
  completeHensel(m,fl,q,cbound)

-- check for possible degree reduction
-- could use polynomial decomposition ?
henselfact(m: ZP):List ZP ==
  deggcd:=degree m
  mm:= m
  while not zero? mm repeat (deggcd:=gcd(deggcd, degree mm); mm:=reductum mm)
  deggcd>1 and deggcd<degree m =>$\$
    faclist := henselfact1(divideExponents(m, deggcd)::ZP)
    "append"/[henselfact1 multiplyExponents(mm, deggcd) for mm in faclist]
  henselfact1 m

quadratic(m: ZP):List(ZP) ==
  d,d2: Z
  d := coefficient(m,1)**2-4*coefficient(m,0)*coefficient(m,2)
  d2 := sqroot(d)
  (d-d2**2)^=0 => [m]
  alpha: Z := coefficient(m,1)+d2
  beta: Z := 2*coefficient(m,2)
  d := gcd(alpha,beta)
  if d ^=1 then
alpha := alpha quo d  
beta := beta quo d  
m0: ZP := monomial(beta,1)+monomial(alpha,0)  
cons(m0,[(m exquo m0):: ZP])

isPowerOf2(n : Z): Boolean ==  
n = 1 => true  
qr : Record(quotient: Z, remainder: Z) := divide(n,2)  
qr.remainder = 1 => false  
isPowerOf2 qr.quotient

subMinusX(supPol : SUPZ): ZP ==  
minusX : SUPZ := monomial(-1,1)$SUPZ  
(elt(supPol,minusX)$SUPZ) : ZP

-- Factorize the polynomial m, test=true if m is known to be  
-- square-free, false otherwise.  
-- FinalFact.contp=content m, FinalFact.factors=List of irreducible  
-- factors with exponent .  
henselFact(m: ZP,test:Boolean):FinalFact ==  
factorlist : List(ParFact) := []  
c : Z  
-- make m primitive  
c := content m  
m := (m exquo c)::ZP  
-- make the lc m positive  
if leadingCoefficient m < 0 then  
c := -c  
m := -m  
-- is x**d factor of m?  
if (d := minimumDegree m) >0 then  
m := (monicDivide(m,monomial(1,d))).quotient  
factorlist := [[monomial(1,1),d]$ParFact]  
d := degree m  
-- is m constant?  
d=0 => [c,factorlist]$FinalFact  
-- is m linear?  
d=1 => [c,cons([[m,1]$ParFact,factorlist]]$FinalFact  
-- does m satisfy Eisenstein’s criterion?  
eisenstein m => [c,cons([[m,1]$ParFact,factorlist]]$FinalFact  
lcPol : ZP := leadingCoefficient(m) :: ZP
-- is m cyclotomic \((x^n - 1)\)?
- \(\text{lcPol} = \text{reductum}(m)\) => -- if true, both will = 1
  for fac in
    (cyclotomicDecomposition(degree m)$CYC : List ZP) repeat
      factorlist := cons([[fac,1]$ParFact,factorlist])$FinalFact

-- is m odd cyclotomic \((x^{2n+1} + 1)\)?
odd?(d) and (lcPol = reductum(m)) =>
  for sfac in cyclotomicDecomposition(degree m)$CYC repeat
    fac:=subMinusX sfac
    if leadingCoefficient fac < 0 then fac := -fac
    factorlist := cons([[fac,1]$ParFact,factorlist])$FinalFact

-- is the poly of the form \(x^n + 1\) with \(n\) a power of 2?
-- if so, then irreducible
isPowerOf2(d) and (lcPol = reductum(m)) =>
  factorlist := cons([[m,1]$ParFact,factorlist])$FinalFact

-- is m quadratic?
d=2 =>
  lfq:List(ZP) := quadratic m
  #lfq=1 => [c,cons([lfq.first,1]$ParFact,factorlist])$FinalFact
  (lf0,lf1) := (lfq.first,second lfq)
  if lf0=lf1 then factorlist := cons([[lf0,2]$ParFact,factorlist])$FinalFact
  else factorlist := append([[v,1]$ParFact for v in lfq],factorlist)$FinalFact

-- m is square-free
test =>
  fln := henselfact(m)
  [c,append(factorlist,[[pf,1]$ParFact for pf in fln]])$FinalFact

-- find the square-free decomposition of m
irrfact := squareFree(m)
llf := factors irrfact

-- factorize the square-free primitive terms
for ll in llf repeat
  d1 := ll.exponent
  pol := ll.factor
  degree pol=1 => factorlist := cons([pol,d1]$ParFact,factorlist)
  degree pol=2 =>
    fln := quadratic(pol)
    factorlist := append([[pf,d1]$ParFact for pf in fln],factorlist)
    fln := henselfact(pol)
    factorlist := append([[pf,d1]$ParFact for pf in fln],factorlist)
  [c,factorlist]$FinalFact
package UFPS1 UnivariateFormalPowerSeriesFunctions

— UnivariateFormalPowerSeriesFunctions.input —

)set break resume
)sys rm -f UnivariateFormalPowerSeriesFunctions.output
)spool UnivariateFormalPowerSeriesFunctions.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateFormalPowerSeriesFunctions
--R
--R UnivariateFormalPowerSeriesFunctions(Coef: Ring) is a package constructor
--R Abbreviation for UnivariateFormalPowerSeriesFunctions is UFPS1
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UFPS1
--R
--R-------------------------------------- Operations --------------------------------------
--R hadamard : (UnivariateFormalPowerSeries(Coef),UnivariateFormalPowerSeries(Coef)) -> UnivariateFormalPowerSeries(Coef)
--R
--E 1

)spool
)lisp (bye)
UnivariateFormalPowerSeriesFunctions examples
====================================================================

This package has no description

See Also:
  o )show UnivariateFormalPowerSeriesFunctions

——

UnivariateFormalPowerSeriesFunctions (UFPS1)

Exports:
  hadamard

—— package UFPS1 UnivariateFormalPowerSeriesFunctions ——

)abbrev package UFPS1 UnivariateFormalPowerSeriesFunctions
++ Description:
++ This package has no description

UnivariateFormalPowerSeriesFunctions(Coef: Ring): Exports == Implementation
where

  UFPS ==> UnivariateFormalPowerSeries Coef

Exports == with

  hadamard: (UFPS, UFPS) -> UFPS

Implementation == add

  hadamard(f, g) ==
series map((z1:Coef,z2:Coef):Coef +-> z1*z2, coefficients f, coefficients g) $StreamFunctions3(Coef, Coef, Coef)

---

UFPS1.dotabb

"UFPS1" [color="FF4488",href="bookvol10.4.pdf#nameddest=UFPS1"]
"UFPS" [color="88FF44",href="bookvol10.3.pdf#nameddest=UFPS"]
"UFPS1" -> "UFPS"

---

package ULS2 UnivariateLaurentSeriesFunctions2

--- UnivariateLaurentSeriesFunctions2.input ---

)set break resume
)sys rm -f UnivariateLaurentSeriesFunctions2.output
)spool UnivariateLaurentSeriesFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateLaurentSeriesFunctions2
--R UnivariateLaurentSeriesFunctions2(Coef1: Ring,Coef2: Ring,vari: Symbol, var2: Symbol, cen1: Coef1, cen2: Coef2) is a package constructor
--R Abbreviation for UnivariateLaurentSeriesFunctions2 is ULS2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ULS2
--R
--R-------------------------------- Operations --------------------------------
--R map : ((Coef1 -> Coef2),UnivariateLaurentSeries(Coef1,vari,cen1)) -> UnivariateLaurentSeries(Coef2,vari,cen2)
--R
--E 1

)spool
)lisp (bye)

---

--- UnivariateLaurentSeriesFunctions2.help ---
UnivariateLaurentSeriesFunctions2 examples

Mapping package for univariate Laurent series. This package allows one to apply a function to the coefficients of a univariate Laurent series.

See Also:
- )show UnivariateLaurentSeriesFunctions2

Exports:
map

--- package ULS2 UnivariateLaurentSeriesFunctions2 ---

)abbrev package ULS2 UnivariateLaurentSeriesFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 5 March 1990
++ Date Last Updated: 5 March 1990
++ Description:
++ Mapping package for univariate Laurent series
++ This package allows one to apply a function to the coefficients of
++ a univariate Laurent series.

UnivariateLaurentSeriesFunctions2(Coef1,Coef2,var1,var2,cen1,cen2):
Exports == Implementation where
Coef1 : Ring
Coef2 : Ring
var1: Symbol
var2: Symbol
cen1: Coef1
cen2: Coef2
ULS1 ==> UnivariateLaurentSeries(Coef1, var1, cen1)
ULS2 ==> UnivariateLaurentSeries(Coef2, var2, cen2)
UTS1 ==> UnivariateTaylorSeries(Coef1, var1, cen1)
UTS2 ==> UnivariateTaylorSeries(Coef2, var2, cen2)
UTSF2 ==> UnivariateTaylorSeriesFunctions2(Coef1, Coef2, UTS1, UTS2)

Exports ==> with
   map: (Coef1 -> Coef2, ULS1) -> ULS2
   ++ \spad{map(f,g(x))} applies the map f to the coefficients of the Laurent
   ++ series \spad{f(g(x))}.

Implementation ==> add

   map(f,ups) == laurent(degree ups, map(f, taylorRep ups)$UTSF2)

___

ULS2.dotabb

"ULS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ULS2"]
"MODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"ULS2" -> "LMODULE"
"ULS2" -> "SGROUP"

___

package UPOLYC2 UnivariatePolynomialCategoryFunctions2

---

UnivariatePolynomialCategoryFunctions2.input ---

)set break resume
)sys rm -f UnivariatePolynomialCategoryFunctions2.output
)spool UnivariatePolynomialCategoryFunctions2.output
)set message test on
)set message auto off
)clear all
UnivariatePolynomialCategoryFunctions2 (UPOLYC2)
Exports:

map

— package UPOLYC2 UnivariatePolynomialCategoryFunctions2 —

)abbrev package UPOLYC2 UnivariatePolynomialCategoryFunctions2
++ Description:
++ Mapping from polynomials over R to polynomials over S
++ given a map from R to S assumed to send zero to zero.

UnivariatePolynomialCategoryFunctions2(R,PR,S,PS): Exports == Impl where
  R, S: Ring
  PR : UnivariatePolynomialCategory R
  PS : UnivariatePolynomialCategory S

Exports ==> with
  map: (R -> S, PR) -> PS
  ++ map(f, p) takes a function f from R to S,
  ++ and applies it to each (non-zero) coefficient of a polynomial p
  ++ over R, getting a new polynomial over S.
  ++ Note that since the map is not applied to zero elements, it may map
  ++ zero to zero.

Impl ==> add
  map(f, p) ==
    ans:PS := 0
    while p ^= 0 repeat
      ans := ans + monomial(f leadingCoefficient p, degree p)
      p := reductum p
    ans

— UPOLYC2.dotabb —

"UPOLYC2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPOLYC2"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPOLYC2" -> "PFECAT"

package UPCDEN UnivariatePolynomialCommonDenominator
--- UnivariatePolynomialCommonDenominator.input ---

)set break resume
)sys rm -f UnivariatePolynomialCommonDenominator.output
)spool UnivariatePolynomialCommonDenominator.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialCommonDenominator
--R
--R UnivariatePolynomialCommonDenominator(R: IntegralDomain,Q: QuotientFieldCategory(R),UP: UnivariatePolynomialCategory(Q)) is a package constructor
--R Abbreviation for UnivariatePolynomialCommonDenominator is UPCDEN
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UPCDEN
--R
--R---------------------------------------- Operations ----------------------------------------
--R clearDenominator : UP -> UP  commonDenominator : UP -> R
--R splitDenominator : UP -> Record(num: UP,den: R)
--R
--E 1

)spool
)lisp (bye)

---

--- UnivariatePolynomialCommonDenominator.help ---

====================================================================
UnivariatePolynomialCommonDenominator examples
====================================================================

UnivariatePolynomialCommonDenominator provides functions to compute the common denominator of the coefficients of univariate polynomials over the quotient field of a gcd domain.

See Also:
• )show UnivariatePolynomialCommonDenominator

---
UnivariatePolynomialCommonDenominator (UPCDEN)

Exports:
  clearDenominator commonDenominator splitDenominator

-- package UPCDEN UnivariatePolynomialCommonDenominator --

)abbrev package UPCDEN UnivariatePolynomialCommonDenominator
++ Author: Manuel Bronstein
++ Date Created: 2 May 1988
++ Date Last Updated: 22 Feb 1990
++ Description:
  ++ UnivariatePolynomialCommonDenominator provides
  ++ functions to compute the common denominator of the coefficients of
  ++ univariate polynomials over the quotient field of a gcd domain.

UnivariatePolynomialCommonDenominator(R, Q, UP): Exports == Impl where
  R : IntegralDomain
  Q : QuotientFieldCategory R
  UP: UnivariatePolynomialCategory Q

Exports ==> with
  commonDenominator: UP -> R
      ++ commonDenominator(q) returns a common denominator d for
      ++ the coefficients of q.
  clearDenominator : UP -> UP
      ++ clearDenominator(q) returns p such that \spad{q = p/d} where d is
      ++ a common denominator for the coefficients of q.
  splitDenominator : UP -> Record(num: UP, den: R)
      ++ splitDenominator(q) returns \spad{[p, d]} such that
      ++ \spad{q = p/d} and d
      ++ is a common denominator for the coefficients of q.

Impl ==> add
  import CommonDenominator(R, Q, List Q)

  commonDenominator p == commonDenominator coefficients p
clearDenominator p ==
  d := commonDenominator p
  map(x +-> numer(d*x)::Q, p)

splitDenominator p ==
  d := commonDenominator p
  [map(x +-> numer(d*x)::Q, p), d]

— UPCDEN.dotabb —

"UPCDEN" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPCDEN"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPCDEN" -> "PFECAT"

package UPDECOMP UnivariatePolynomialDecompositionPackage

— UnivariatePolynomialDecompositionPackage.input —

)set break resume
)sys rm -f UnivariatePolynomialDecompositionPackage.output
)spool UnivariatePolynomialDecompositionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialDecompositionPackage
--R
--R UnivariatePolynomialDecompositionPackage(R: Join(IntegralDomain,CharacteristicZero),UP: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for UnivariatePolynomialDecompositionPackage is UPDECOMP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UPDECOMP
--R
--R--------------------------------------------- Operations ---------------------------------------------
--R leftFactorIfCan : (UP,UP) -> Union(UP,"failed")
--R monicCompleteDecompose : UP -> List(UP)
--R monicDecomposeIfCan : UP -> Union(Record(left: UP,right: UP),"failed")
UnivariatePolynomialDecompositionPackage (UPDECOMP)

Exports:
  leftFactorIfCan    monicCompleteDecompose  monicDecomposeIfCan
  monicRightFactorIfCan rightFactorIfCan

--- package UPDECOMP UnivariatePolynomialDecompositionPackage ---
UnivariatePolynomialDecompositionPackage(R, UP): Exports == Implementation where

R : Join(IntegralDomain, CharacteristicZero)
UP : UnivariatePolynomialCategory(R)
N --> NonNegativeInteger
LR --> Record(left: UP, right: UP)
QR --> Record(quotient: UP, remainder: UP)

Exports ==> with

monicRightFactorIfCan: (UP, N) -> Union(UP, "failed")
  ++ monicRightFactorIfCan(f, d) returns a candidate to be the
  ++ monic right factor (h in f = g o h) of degree d of a
  ++ functional decomposition of the polynomial f or
  ++ \verb|\spad{"failed"}| if no such candidate.
rightFactorIfCan: (UP, N, R) -> Union(UP, "failed")
  ++ rightFactorIfCan(f, d, c) returns a candidate to be the
  ++ right factor (h in f = g o h) of degree d with leading
  ++ coefficient c of a functional decomposition of the
  ++ polynomial f or \verb|\spad{"failed"}| if no such candidate.
leftFactorIfCan: (UP, UP) -> Union(UP, "failed")
  ++ leftFactorIfCan(f, h) returns the left factor (g in f = g o h)
  ++ of the functional decomposition of the polynomial f with
  ++ given h or \verb|\spad{"failed"}| if g does not exist.
monicDecomposeIfCan: UP \rightarrow \text{Union}(\text{LR}, \text{"failed"})
++ monicDecomposeIfCan(f) returns a functional decomposition
++ of the monic polynomial f of "failed" if it has not found any.
monicCompleteDecompose: UP \rightarrow \text{List} \text{UP}
++ monicCompleteDecompose(f) returns a list of factors of f for
++ the functional decomposition ([f_1, \ldots, f_n] means
++ f = f_1 \circ \ldots \circ f_n).

Implementation ==> add

rightFactorIfCan(p, dq, lcq) ==
dp := \text{degree } p
zero? lcq =>
  \text{error "rightFactorIfCan: leading coefficient may not be zero"}
(zero? dp) or (zero? dq) => "failed"
nc := dp \text{ exquo } dq
nc case "failed" => "failed"
n := nc::N
s := \text{subtractIfCan}(dq,1)::N
lcp := \text{leadingCoefficient } p
q: UP := \text{monomial}(lcq,dq)
k: N
for k in 1..s repeat
  c: R := 0
  i: N
  for i in 0..\text{subtractIfCan}(k,1)::N repeat
    c := c+\text{coefficient}(q,\text{subtractIfCan}(dq,i)::N)*
    \text{coefficient}(p,\text{subtractIfCan}(dp+i,k)::N)
  cquo := c \text{ exquo } ((k*n)::R*lcp)
cquo case "failed" => return "failed"
q := q+\text{monomial}(cquo::R,\text{subtractIfCan}(dq,k)::N)
q

monicRightFactorIfCan(p, dq) == rightFactorIfCan(p, dq, 1$R)

import \text{UnivariatePolynomialDivisionPackage}(R, UP)

leftFactorIfCan(f, h) ==
g: UP := 0
zero? \text{degree } h => "failed"
for i in 0.. while not zero? f repeat
  qrf := divideIfCan(f, h)
  qrf case "failed" => return "failed"
  qr := qrf :: QR
  r := qr.\text{remainder}
  not ground? r => return "failed"
  g := g+\text{monomial}(\text{ground}(r), i)
f := qr.\text{quotient}
g
monicDecomposeIfCan f ==
  df := degree f
  zero? df => "failed"
  for dh in 2..subtractIfCan(df,1)::N | zero?(df rem dh) repeat
    h := monicRightFactorIfCan(f,dh)
    h case UP =>
      g := leftFactorIfCan(f,h::UP)
      g case UP => return [g::UP,h::UP]
    "failed"

monicCompleteDecompose f ==
  cf := monicDecomposeIfCan f
  cf case "failed" => [ f ]
  lr := cf :: LR
  append(monicCompleteDecompose lr.left,[lr.right])

package UPDIVP UnivariatePolynomialDivisionPackage

— UnivariatePolynomialDivisionPackage.input —

)set break resume
)sys rm -f UnivariatePolynomialDivisionPackage.output
)spool UnivariatePolynomialDivisionPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialDivisionPackage
--R
--R UnivariatePolynomialDivisionPackage(R: IntegralDomain,UP: UnivariatePolynomialCategory(R))
--R Abbreviation for UnivariatePolynomialDivisionPackage is UPDIVP
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UPDIVP
--R
--R----------------------------------- Operations ----------------------------------
--R divideIfCan : (UP,UP) -> Union(Record(quotient: UP,remainder: UP),"failed")
--R
---E 1

)spool
)lisp (bye)

UnivariatePolynomialDivisionPackage (UPDIVP)

Exports:
divideIfCan

--- package UPDIVP UnivariatePolynomialDivisionPackage ---
UnivariatePolynomialDivisionPackage(R,UP): Exports == Implementation where
R : IntegralDomain
UP : UnivariatePolynomialCategory(R)
N ==> NonNegativeInteger
QR ==> Record(quotient: UP, remainder: UP)

Exports ==> with
divideIfCan: (UP,UP) -> Union(QR,"failed")
   ++ divideIfCan(f,g) returns quotient and remainder of the
   ++ division of f by g or "failed" if it has not succeeded.

Implementation ==> add

divideIfCan(p1:UP,p2:UP):Union(QR,"failed") ==
   zero? p2 => error "divideIfCan: division by zero"
   one? (lc := leadingCoefficient p2) => monicDivide(p1,p2)
   ((lc := leadingCoefficient p2) = 1) => monicDivide(p1,p2)
   q: UP := 0
   while not ((e := subtractIfCan(degree(p1),degree(p2))) case "failed") repeat
      c := leadingCoefficient(p1) exquo lc
      c case "failed" => return "failed"
      ee := e::N
      q := q+monomial(c::R,ee)
      p1 := p1-c*mapExponents(x +-> x+ee, p2)
   [q,p1]
package UP2 UnivariatePolynomialFunctions2

— UnivariatePolynomialFunctions2.input —

)set break resume
)sys rm -f UnivariatePolynomialFunctions2.output
)spool UnivariatePolynomialFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialFunctions2
--R
--R UnivariatePolynomialFunctions2(x: Symbol,R: Ring,y: Symbol,S: Ring) is a package constructor
--R Abbreviation for UnivariatePolynomialFunctions2 is UP2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UP2
--R
--R------------------------------- Operations --------------------------------
--R map : ((R -> S),UnivariatePolynomial(x,R)) -> UnivariatePolynomial(y,S)
--R
--E 1

)spool
)lisp (bye)

— UnivariatePolynomialFunctions2.help —

====================================================================
UnivariatePolynomialFunctions2 examples
====================================================================

This package lifts a mapping from coefficient rings R to S to a mapping from UnivariatePolynomial(x,R) to UnivariatePolynomial(y,S).

Note that the mapping is assumed to send zero to zero, since it will only be applied to the non-zero coefficients of the polynomial.

See Also:
o )show UnivariatePolynomialFunctions2
UnivariatePolynomialFunctions2 (UP2)

Exports:
map

--- package UP2 UnivariatePolynomialFunctions2 ---

)abbrev package UP2 UnivariatePolynomialFunctions2
++ Description:
++ This package lifts a mapping from coefficient rings R to S to
++ a mapping from \spadtype{UnivariatePolynomial}(x,R) to
++ \spadtype{UnivariatePolynomial}(y,S). Note that the mapping is assumed
++ to send zero to zero, since it will only be applied to the non-zero
++ coefficients of the polynomial.

UnivariatePolynomialFunctions2(x:Symbol, R:Ring, y:Symbol, S:Ring): with
  map: (R -> S, UnivariatePolynomial(x,R)) -> UnivariatePolynomial(y,S)
  ++ map(func, poly) creates a new polynomial by applying func to
  ++ every non-zero coefficient of the polynomial poly.
== add
map(f, p) == map(f, p)$UnivariatePolynomialCategoryFunctions2(R,
  UnivariatePolynomial(x, R), S, UnivariatePolynomial(y, S))

---

UP2.dotabb

"UP2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UP2"]
"LMODULE" [color="#4488FF",href="bookvol10.2.pdf#nameddest=LMODULE"]
"SGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SGROUP"]
"UP2" -> "LMODULE"
"UP2" -> "SGROUP"
package UPMP UnivariatePolynomialMultiplicationPackage

--- UnivariatePolynomialMultiplicationPackage.input ---

)set break resume
)sys rm -f UnivariatePolynomialMultiplicationPackage.output
)spool UnivariatePolynomialMultiplicationPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialMultiplicationPackage
--R
--R UnivariatePolynomialMultiplicationPackage(R: Ring,U: UnivariatePolynomialCategory(R)) is a package constructor
--R Abbreviation for UnivariatePolynomialMultiplicationPackage is UPMP
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UPMP
--R
--R----------------------------------- Operations -----------------------------------
--R karatsubaOnce : (U,U) -> U noKaratsuba : (U,U) -> U
--R karatsuba : (U,U,NonNegativeInteger,NonNegativeInteger) -> U
--R
--E 1

)spool
)lisp (bye)

---

--- UnivariatePolynomialMultiplicationPackage.help ---

====================================================================
UnivariatePolynomialMultiplicationPackage examples
====================================================================

This package implements Karatsuba’s trick for multiplying (large) univariate polynomials. It could be improved with a version doing the work on place and also with a special case for squares. We’ve done this in Basicmath, but we believe that this out of the scope of Axiom.

See Also:
o )show UnivariatePolynomialMultiplicationPackage

---
UnivariatePolynomialMultiplicationPackage (UPMP)

UnivariatePolynomialMultiplicationPackage(R: Ring, U: UnivariatePolynomialCategory(R)): C == T

where

HL ==> Record(quotient:U, remainder:U)

C == with

noKaratsuba: (U, U) -> U
++ \spad{noKaratsuba(a,b)} returns \spad{a*b} without
++ using Karatsuba’s trick at all.
karatsubaOnce: (U, U) -> U
++ \spad{karatsubaOnce(a,b)} returns \spad{a*b} by applying
++ Karatsuba’s trick once. The other multiplications
++ are performed by calling \spad{*} from \spad{U}.
karatsuba: (U, U, NonNegativeInteger, NonNegativeInteger) -> U;
++ \spad{karatsuba(a,b,1,k)} returns \spad{a*b} by applying
++ Karatsuba’s trick provided that both \spad{a} and \spad{b}
++ have at least \spad{1} terms and \spad{k > 0} holds
++ and by calling \spad{noKaratsuba} otherwise. The other
++ multiplications are performed by recursive calls with

Exports:
karatsuba karatsubaOnce noKaratsuba
++ the same third argument and \spad{k-1} as fourth argument.

T == add
noKaratsuba(a,b) ==
    zero? a => a
    zero? b => b
    zero?(degree(a)) => leadingCoefficient(a) * b
    zero?(degree(b)) => a * leadingCoefficient(b)
lu: List(U) := reverse monomials(a)
res: U := 0;
for u in lu repeat
    res := pomopo!(res, leadingCoefficient(u), degree(u), b)
res
karatsubaOnce(a:U,b:U): U ==
da := minimumDegree(a)
db := minimumDegree(b)
if not zero? da then a := shiftRight(a,da)
if not zero? db then b := shiftRight(b,db)
d := da + db
n: NonNegativeInteger := min(degree(a),degree(b)) quo 2
rec: HL := karatsubaDivide(a, n)
ha := rec.quotient
la := rec.remainder
rec := karatsubaDivide(b, n)
hb := rec.quotient
lb := rec.remainder
w: U := (ha - la) * (lb - hb)
u: U := la * lb
v: U := ha * hb
w := w + (u + v)
w := shiftLeft(w,n) + u
zero? d => shiftLeft(v,2*n) + w
shiftLeft(v,2*n + d) + shiftLeft(w,d)
karatsuba(a:U,b:U,l:NonNegativeInteger,k:NonNegativeInteger): U ==
    zero? k => noKaratsuba(a,b)
degree(a) < l => noKaratsuba(a,b)
degree(b) < l => noKaratsuba(a,b)
nNumberOfMonomials(a) < l => noKaratsuba(a,b)
nNumberOfMonomials(b) < l => noKaratsuba(a,b)
da := minimumDegree(a)
db := minimumDegree(b)
if not zero? da then a := shiftRight(a,da)
if not zero? db then b := shiftRight(b,db)
d := da + db
n: NonNegativeInteger := min(degree(a),degree(b)) quo 2
k := subtractIfCan(k,1)::NonNegativeInteger
rec: HL := karatsubaDivide(a, n)
ha := rec.quotient
la := rec.remainder
rec := karatsubaDivide(b, n)
hb := rec.quotient
1b := rec.remainder
w: U := karatsuba(ha - la, 1b - hb, 1, k)
ul: U := karatsuba(la, lb, 1, k)
v: U := karatsuba(ha, hb, 1, k)
w := w + (u + v)
w := shiftLeft(w,n) + u
zero? d => shiftLeft(v,2*n) + w
shiftLeft(v,2*n + d) + shiftLeft(w,d)

---

<table>
<thead>
<tr>
<th>UPMP.dotabb</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;UPMP&quot; [color=&quot;#FF4488&quot;,href=&quot;bookvol10.4.pdf#nameddest=UPMP&quot;]</td>
</tr>
<tr>
<td>&quot;PFECAT&quot; [color=&quot;#4488FF&quot;,href=&quot;bookvol10.2.pdf#nameddest=PFECAT&quot;]</td>
</tr>
<tr>
<td>&quot;UPMP&quot; -&gt; &quot;PFECAT&quot;</td>
</tr>
</tbody>
</table>

package UPSQFREE UnivariatePolynomialSquareFree

--- UnivariatePolynomialSquareFree.input ---

)set break resume
)sys rm -f UnivariatePolynomialSquareFree.output
)spool UnivariatePolynomialSquareFree.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePolynomialSquareFree
--R
--R UnivariatePolynomialSquareFree(RC: IntegralDomain,P)where
--R P: Join(UnivariatePolynomialCategory(RC),IntegralDomain)with
--R gcd : (%,%) -> % is a package constructor
--R Abbreviation for UnivariatePolynomialSquareFree is UPSQFREE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UPSQFREE
--R
--R----------------------------- Operations -----------------------------
--R squareFree : P -> Factored(P) squareFreePart : P -> P
--R BumInSepFFE : Record(flg: Union("nil","sqfr","irred","prime"),fctr: P,xpnt: Integer) -> Rec...
UnivariatePolynomialSquareFree (UPSQFREE)

Exports:
- squareFree
- squareFreePart
- BumInSepFFE
package UPSQFREE UnivariatePolynomialSquareFree

)abbrev package UPSQFREE UnivariatePolynomialSquareFree
++ Author: Dave Barton, Barry Trager
++ Description:
++ This package provides for square-free decomposition of
++ univariate polynomials over arbitrary rings, i.e.
++ a partial factorization such that each factor is a product
++ of irreducibles with multiplicity one and the factors are
++ pairwise relatively prime. If the ring
++ has characteristic zero, the result is guaranteed to satisfy
++ this condition. If the ring is an infinite ring of
++ finite characteristic, then it may not be possible to decide when
++ polynomials contain factors which are pth powers. In this
++ case, the flag associated with that polynomial is set to "nil"
++ (meaning that that polynomials are not guaranteed to be square-free).

UnivariatePolynomialSquareFree(RC:IntegralDomain,P):C == T

where
fUnion ==> Union("nil", "sqfr", "irred", "prime")
FF ==> Record(flg:fUnion, fctr:P, xpnt:Integer)
P:Join(UnivariatePolynomialCategory(RC),IntegralDomain) with
  gcd: (%,%) -> %
  ++ gcd(p,q) computes the greatest-common-divisor of p and q.
C == with
  squareFree: P -> Factored(P)
  ++ squareFree(p) computes the square-free factorization of the
  ++ univariate polynomial p. Each factor has no repeated roots, and the
  ++ factors are pairwise relatively prime.
  squareFreePart: P -> P
  ++ squareFreePart(p) returns a polynomial which has the same
  ++ irreducible factors as the univariate polynomial p, but each
  ++ factor has multiplicity one.
  BumInSepFFE: FF -> FF
  ++ BumInSepFFE(f) is a local function, exported only because
  ++ it has multiple conditional definitions.
T == add
  if RC has CharacteristicZero then
    squareFreePart(p:P) == (p exquo gcd(p, differentiate p))::P
  else
    squareFreePart(p:P) ==
      unit(s := squareFree(p)$$%) * */[f.factor for f in factors s]
  if RC has FiniteFieldCategory then
    BumInSepFFE(ffe:FF) ==
      ["sqfr", map(charthRoot,ffe.fctr), characteristic$P*ffe.xpnt]
else if RC has CharacteristicNonZero then
  BumInSepFFE(ffe:FF) ==
  np := multiplyExponents(ffe.fctr,characteristic$P:NonNegativeInteger)
  (nthrp := charthRoot(np)) case "failed" =>
    ["nil", np, ffe.xpnt]
  ["sqfr", nthrp, characteristic$P*ffe.xpnt]
else
  BumInSepFFE(ffe:FF) ==
  ["nil",
   multiplyExponents(ffe.fctr,characteristic$P:NonNegativeInteger),
   ffe.xpnt]
if RC has CharacteristicZero then
  squareFree(p:P) == --Yun's algorithm - see SYMSAC '76, p.27
    --Note ci primitive is, so GCD's don't need to %do contents.
    --Change gcd to return cofctrs also?
    ci:=p; di:=differentiate(p); pi:=gcd(ci,di)
    degree(pi)=0 =>
      (u,c,a):=unitNormal(p)
      makeFR(u,["sqfr",c,1])
    i:NonNegativeInteger:=0; lffe:List FF:=[
    lcp := leadingCoefficient p
    while degree(ci)^=0 repeat
      ci:=(ci exquo pi)::P
      di:=(di exquo pi)::P - differentiate(ci)
      pi:=gcd(ci,di)
      i:=i+1
      degree(pi) > 0 =>
        lcp:=(lcp exquo (leadingCoefficient(pi)**i))::RC
        lffe:=[["sqfr",pi,i],:lffe]
        makeFR(lcp::P,lffe)
    else
      squareFree(p:P) == --Musser's algorithm - see SYMSAC '76, p.27
        --p MUST BE PRIMITIVE, Any characteristic.
        --Note ci primitive, so GCD's don't need to %do contents.
        --Change gcd to return cofctrs also?
        ci := gcd(p,differentiate(p))
        degree(ci)=0 =>
          (u,c,a):=unitNormal(p)
          makeFR(u,["sqfr",c,1])
        di := (p exquo ci)::P
        i:NonNegativeInteger:=0; lffe:List FF:=[
        dunit : P := 1
        while degree(di)^=0 repeat
          diprev := di
          di := gcd(ci,di)
          ci:=(ci exquo di)::P
i:=i+1
degree(diprev) = degree(di) =>
  lc := (leadingCoefficient(diprev) exquo leadingCoefficient(di))::RC
  dunit := lc**i * dunit
  pi:=(diprev exquo di)::P
  lffe:=["sqfr",pi,i],:lffe]
  dunit := dunit * di ** (i+1)
degree(ci)=0 => makeFR(dunit*ci,lffe)
redSqfr:=squareFree(divideExponents(ci,characteristic$P)::P)
lsnil:= [BumInSepFFE(ffe) for ffe in factorList redSqfr]
lffe:=append(lsnl,lffe)
makeFR(dunit*(unit redSqfr),lffe)

———

— UPSQFREE.dotabb —

"UPSQFREE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPSQFREE"]
"PFECAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PFECAT"]
"UPSQFREE" -> "PFECAT"

———

package UPXS2 UnivariatePuiseuxSeriesFunctions2

— UnivariatePuiseuxSeriesFunctions2.input —

)set break resume
)sys rm -f UnivariatePuiseuxSeriesFunctions2.output
)spool UnivariatePuiseuxSeriesFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariatePuiseuxSeriesFunctions2
--R
--R UnivariatePuiseuxSeriesFunctions2(Coef1: Ring,Coef2: Ring,var1: Symbol,var2: Symbol,cen1: Coef1,cen2: Coef2)
--R Abbreviation for UnivariatePuiseuxSeriesFunctions2 is UPXS2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UPXS2
--R
--R----------------------------------- Operations -----------------------------------
--R map : ((Coef1 -> Coef2),UnivariatePuiseuxSeries(Coef1,var1,cen1)) -> UnivariatePuiseuxSeries
This package allows one to apply a function to the coefficients of a univariate Puiseux series.

See Also:
- )show UnivariatePuiseuxSeriesFunctions2

Exports:
- map
++ Date Last Updated: 5 April 1991
++ Description:
++ Mapping package for univariate Puiseux series.
++ This package allows one to apply a function to the coefficients of
++ a univariate Puiseux series.

UnivariatePuiseuxSeriesFunctions2(Coef1,Coef2,var1,var2,cen1,cen2):_  
Exports == Implementation where
  Coef1 : Ring
  Coef2 : Ring
  var1: Symbol
  var2: Symbol
  cen1: Coef1
  cen2: Coef2
  UPS1 ==> UnivariatePuiseuxSeries(Coef1, var1, cen1)
  UPS2 ==> UnivariatePuiseuxSeries(Coef2, var2, cen2)
  ULSP2 ==> UnivariateLaurentSeriesFunctions2(Coef1, Coef2, var1, var2, cen1, cen2)

Exports ==> with
  map: (Coef1 -> Coef2,UPS1) -> UPS2
    + \spad{map(f,g(x))} applies the map f to the coefficients of the
    + Puiseux series \spad{g(x)}.

Implementation ==> add

  map(f,ups) == puiseux(rationalPower ups, map(f, laurentRep ups)$ULSP2)


— UPXS2.dotabb —

"UPXS2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UPXS2"]
"PID" [color="#4488FF",href="bookvol10.2.pdf#nameddest=PID"]
"OAGROUP" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OAGROUP"]
"UPXS2" -> "PID"
"UPXS2" -> "OAGROUP"

package OREPTCO UnivariateSkewPolynomialCategory-Ops

— UnivariateSkewPolynomialCategoryOps.input —
package orepcto univariate skewpolynomialcategoryops

)set break resume
)sy s rm -f UnivariateSkewPolynomialCategoryOps.output
)spool UnivariateSkewPolynomialCategoryOps.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateSkewPolynomialCategoryOps

--R UnivariateSkewPolynomialCategoryOps(R: Ring,C: UnivariateSkewPolynomialCategory(R)) is a package constructor
--R Abbreviation for UnivariateSkewPolynomialCategoryOps is OREPCTO
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for OREPCTO
--R
--R-------------------------------- Operations --------------------------------
--R apply : (C,R,R,Automorphism(R),(R -> R)) -> R
--R leftDivide : (C,C,Automorphism(R)) -> Record(quotient: C,remainder: C) if R has FIELD
--R monicLeftDivide : (C,C,Automorphism(R)) -> Record(quotient: C,remainder: C) if R has INTDOM
--R monicRightDivide : (C,C,Automorphism(R)) -> Record(quotient: C,remainder: C) if R has INTDOM
--R rightDivide : (C,C,Automorphism(R)) -> Record(quotient: C,remainder: C) if R has FIELD
--R times : (C,C,Automorphism(R),(R -> R)) -> C
--R
--E 1

)spool
)lisp (bye)

——

— UnivariateSkewPolynomialCategoryOps.help —

====================================================================
UnivariateSkewPolynomialCategoryOps examples
====================================================================

UnivariateSkewPolynomialCategoryOps provides products and divisions of univariate skew polynomials.

See Also:
  o )show UnivariateSkewPolynomialCategoryOps

——
UnivariateSkewPolynomialCategoryOps (OREPCTO)

Exports:

apply  leftDivide  monicLeftDivide
monicRightDivide  rightDivide  times

— package OREPCTO UnivariateSkewPolynomialCategoryOps —

)abbrev package OREPCTO UnivariateSkewPolynomialCategoryOps
++ Author: Manuel Bronstein
++ Date Created: 1 February 1994
++ Date Last Updated: 1 February 1994
++ Description:
++ \spad{UnivariateSkewPolynomialCategoryOps} provides products and
++ divisions of univariate skew polynomials.
-- Putting those operations here rather than defaults in OREPCAT allows
-- OREPCAT to be defined independently of sigma and delta.
-- MB 2/94

UnivariateSkewPolynomialCategoryOps(R, C):Exports == Implementation where
R: Ring
C: UnivariateSkewPolynomialCategory R

N ==> NonNegativeInteger
MOR ==> Automorphism R
QUOREM ==> Record(quotient: C, remainder: C)

Exports == with

  times: (C, C, MOR, R -> R) -> C
  ++ times(p, q, sigma, delta) returns \spad{p * q}.
  ++ \spad{\sigma} and \spad{\delta} are the maps to use.
  apply: (C, R, R, MOR, R -> R) -> R
  ++ apply(p, c, m, sigma, delta) returns \spad{p(m)} where the action
  ++ is given by \spad{x m = c sigma(m) + delta(m)}.
if R has IntegralDomain then
  monicLeftDivide: (C, C, MOR) -> QUOREM
  ++ monicLeftDivide(a, b, sigma) returns the pair \spad{[q,r]}
+ such that \( a = b \cdot q + r \) and the degree of \( r \) is
+ less than the degree of \( b \).
+ \( b \) must be monic.
+ This process is called 'left division'.
+ \( \sigma \) is the morphism to use.

monicRightDivide: (C, C, MOR) -> QUOREM
+ monicRightDivide(a, b, sigma) returns the pair \( \{q,r\} \) such
+ that \( a = q \cdot b + r \) and the degree of \( r \) is
+ less than the degree of \( b \).
+ \( b \) must be monic.
+ This process is called 'right division'.
+ \( \sigma \) is the morphism to use.

if R has Field then
leftDivide: (C, C, MOR) -> QUOREM
+ leftDivide(a, b, sigma) returns the pair \( \{q,r\} \) such
+ that \( a = b \cdot q + r \) and the degree of \( r \) is
+ less than the degree of \( b \).
+ \( b \) must be monic.
+ This process is called 'left division'.
+ \( \sigma \) is the morphism to use.

rightDivide: (C, C, MOR) -> QUOREM
+ rightDivide(a, b, sigma) returns the pair \( \{q,r\} \) such
+ that \( a = q \cdot b + r \) and the degree of \( r \) is
+ less than the degree of \( b \).
+ \( b \) must be monic.
+ This process is called 'right division'.
+ \( \sigma \) is the morphism to use.

Implementation ==> add

termPoly: (R, N, C, MOR, R -> R) -> C
localLeftDivide : (C, C, MOR, R) -> QUOREM
localRightDivide: (C, C, MOR, R) -> QUOREM

times(x, y, sigma, delta) ==
  zero? y => 0
  z:C := 0
  while x ^= 0 repeat
    z := z + termPoly(leadingCoefficient x, degree x, y, sigma, delta)
    x := reductum x
  z

termPoly(a, n, y, sigma, delta) ==
  zero? y => 0
  (u := subtractIfCan(n, 1)) case "failed" => a * y
  n1 := u::N
  z:C := 0
  while y ^= 0 repeat
    m := degree y
    b := leadingCoefficient y
    z := z + termPoly(a, n1, monomial(sigma b, m + 1), sigma, delta)
    + termPoly(a, n1, monomial(delta b, m), sigma, delta)
    y := reductum y
apply(p, c, x, sigma, delta) ==
  w:R := 0
  xn:R := x
  for i in 0..degree p repeat
    w := w + coefficient(p, i) * xn
    xn := c * sigma xn + delta xn
  return w

-- localLeftDivide(a, b, sigma, b1) ==
-- localLeftDivide(a, b, sigma, b1) ==
  zero? b => error "leftDivide: division by 0"  
  zero? a or
  (n := subtractIfCan(degree(a), (m := degree b))) case "failed" =>
    [0, a]
  q := monomial((sigma**(-m)) (b1 * leadingCoefficient a), n::N)
  qr := localLeftDivide(a - b * q, b, sigma, b1)
  [q + qr.quotient, qr.remainder]

-- localRightDivide(a, b, sigma, b1) ==
-- localRightDivide(a, b, sigma, b1) ==
  zero? b => error "rightDivide: division by 0"  
  zero? a or
  (n := subtractIfCan(degree(a), (m := degree b))) case "failed" =>
    [0, a]
  q := monomial(leadingCoefficient(a) * (sigma**n) b1, n::N)
  qr := localRightDivide(a - q * b, b, sigma, b1)
  [q + qr.quotient, qr.remainder]

if R has IntegralDomain then
  monicLeftDivide(a, b, sigma) ==
    unit?(u := leadingCoefficient b) =>
    localLeftDivide(a, b, sigma, recip(u)::R)
    error "monicLeftDivide: divisor is not monic"

  monicRightDivide(a, b, sigma) ==
    unit?(u := leadingCoefficient b) =>
    localRightDivide(a, b, sigma, recip(u)::R)
    error "monicRightDivide: divisor is not monic"

if R has Field then
  leftDivide(a, b, sigma) ==
    localLeftDivide(a, b, sigma, inv leadingCoefficient b)

  rightDivide(a, b, sigma) ==
    localRightDivide(a, b, sigma, inv leadingCoefficient b)
package UTS2 UnivariateTaylorSeriesFunctions2

---

UnivariateTaylorSeriesFunctions2.input

)set break resume
)sys rm -f UnivariateTaylorSeriesFunctions2.output
)spool UnivariateTaylorSeriesFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateTaylorSeriesFunctions2

--R
--R UnivariateTaylorSeriesFunctions2(Coef1: Ring, Coef2: Ring, UTS1: UnivariateTaylorSeriesCategory(Coef1), UTS2: UnivariateTaylorSeriesCategory(Coef2)) is a package constructor
--R Abbreviation for UnivariateTaylorSeriesFunctions2 is UTS2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UTS2
--R
--R----------------------------- Operations -----------------------------
--R map : ((Coef1 -> Coef2), UTS1) -> UTS2
--E 1

)spool
)lisp (bye)

---

UnivariateTaylorSeriesFunctions2.help

====================================================================
UnivariateTaylorSeriesFunctions2 examples
====================================================================
CHAPTER 22. CHAPTER U

Mapping package for univariate Taylor series. This package allows one to apply a function to the coefficients of a univariate Taylor series.

See Also:
  o )show UnivariateTaylorSeriesFunctions2

---

UnivariateTaylorSeriesFunctions2 (UTS2)

Exports:
map

--- package UTS2 UnivariateTaylorSeriesFunctions2 ---

)abbrev package UTS2 UnivariateTaylorSeriesFunctions2
++ Author: Clifton J. Williamson
++ Date Created: 9 February 1990
++ Date Last Updated: 9 February 1990
++ Description:
++ Mapping package for univariate Taylor series.
++ This package allows one to apply a function to the coefficients of
++ a univariate Taylor series.

UnivariateTaylorSeriesFunctions2(Coef1,Coef2,UTS1,UTS2):_Exports == Implementation where
Coef1 : Ring
Coef2 : Ring
UTS1 : UnivariateTaylorSeriesCategory Coef1
UTS2 : UnivariateTaylorSeriesCategory Coef2
ST2 ==> StreamFunctions2(Coef1,Coef2)
Exports ==> with
map: (Coef1 -> Coef2,UTS1) -> UTS2
++\spad{map(f,g(x))} applies the map f to the coefficients of
++ the Taylor series \spad{g(x)}.

Implementation ==> add
map(f,uts) == series map(f,coefficients uts)$ST2

package UTSODE UnivariateTaylorSeriesODESolver

UnivariateTaylorSeriesODESolver.input

)set break resume
)sys rm -f UnivariateTaylorSeriesODESolver.output
)spool UnivariateTaylorSeriesODESolver.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UnivariateTaylorSeriesODESolver
--R
--R UnivariateTaylorSeriesODESolver(Coef: Algebra(Fraction(Integer)),UTS: UnivariateTaylorSeriesCategory(Coef)) is a package constructor
--R Abbreviation for UnivariateTaylorSeriesODESolver is UTSODE
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UTSODE
--R
--R-------------------------------------------------------------------------------- Operations  --------------------------------
--R fixedPointExquo : (UTS,UTS) -> UTS  ode1 : ((UTS -> UTS),Coef) -> UTS
--R mpsode : (List(Coef),List((List(UTS) -> UTS)) -> List(UTS)
--R ode : ((List(UTS) -> UTS),List(Coef)) -> UTS
--R ode2 : (((UTS,UTS) -> UTS),Coef,Coef) -> UTS
--R stFunc1 : (UTS -> UTS) -> (Stream(Coef) -> Stream(Coef))
This package provides Taylor series solutions to regular linear or non-linear ordinary differential equations of arbitrary order.

See Also:
- )show UnivariateTaylorSeriesODESolver

---

UnivariateTaylorSeriesODESolver (UTSODE)

Exports:
- fixExp
- fixode
- fixode1
- fixode2
- stFunc1
- stFunc2
- stFuncN

--- package UTSODE UnivariateTaylorSeriesODESolver ---

)abbrev package UTSODE UnivariateTaylorSeriesODESolver
++ Author: Stephen Watt (revised by Clifton J. Williamson)
++ Date Created: February 1988
++ Date Last Updated: 30 September 1993
++ Description:
++ Taylor series solutions of explicit ODE's.
++ This package provides Taylor series solutions to regular
++ linear or non-linear ordinary differential equations of
++ arbitrary order.

UnivariateTaylorSeriesODESolver(Coef, UTS):=
Exports == Implementation where
Coef : Algebra Fraction Integer
UTS : UnivariateTaylorSeriesCategory Coef
L ==> List
L2 ==> ListFunctions2
FN ==> (L UTS) -> UTS
ST ==> Stream Coef
YS ==> Y$ParadoxicalCombinatorsForStreams(Coef)
STT ==> StreamTaylorSeriesOperations(Coef)

Exports ==> with
  stFunc1: (UTS -> UTS) -> (ST -> ST)
  ++ stFunc1(f) is a local function exported due to compiler problem.
  ++ This function is of no interest to the top-level user.
  stFunc2: ((UTS, UTS) -> UTS) -> ((ST, ST) -> ST)
  ++ stFunc2(f) is a local function exported due to compiler problem.
  ++ This function is of no interest to the top-level user.
  stFuncN: FN -> ((L ST) -> ST)
  ++ stFuncN(f) is a local function xported due to compiler problem.
  ++ This function is of no interest to the top-level user.
fixedPointExquo: (UTS, UTS) -> UTS
  ++ fixedPointExquo(f, g) computes the exact quotient of \( f \) and 
  ++ \( g \) using a fixed point computation.
ode1: ((UTS -> UTS), Coef) -> UTS
  ++ ode1(f, c) is the solution to \( y' = f(y) \)
  ++ such that \( y(a) = c \).
ode2: ((UTS, UTS) -> UTS, Coef, Coef) -> UTS
  ++ ode2(f, c0, c1) is the solution to \( y'' = f(y, y') \), \( y(a) = c0 \)
  ++ and \( y'(a) = c1 \).
ode: (FN, List Coef) -> UTS
  ++ ode2(f, c) is the solution to \( y(n) = f(y, y', ..., y(n-1)) \) such that
  ++ \( y(i)(a) = c(i) \) for \( i \) in 1..n.
mpsode: (L Coef, L FN) -> L UTS
  ++ mpsode(r, f) solves the system of differential equations
  ++ \( \text{spad}[dy[i]/dx = f[i] \{x, y[i], y[2], ..., y[n]\}], \)
  ++ \( \text{spad}[y[i](a) = r[i]] \) for \( i \) in 1..n.

Implementation ==> add

  stFunc1 f == s -> coefficients f series(s)
\[
\text{stFunc2 } f \equiv (s1, s2) \rightarrow \text{coefficients } f(\text{series}(s1), \text{series}(s2))
\]
\[
\text{stFuncN } f \equiv \lambda s \rightarrow \text{coefficients } f(\text{map}((\text{series}), s))\text{MapFunctions}(\text{ST}, \text{UTS})
\]
\[
\text{import StreamTaylorSeriesOperations(Coef)}
\]
\[
\text{divloope}(\text{Coef}, \text{ST}, \text{Coef}, \text{ST}, \text{ST}) \rightarrow \text{ST}
\]
\[
\text{divloope}(hx, tx, hy, ty, c) \equiv \text{delay}(\text{concat}(hx, hy, hy*(tx-(ty*c))))
\]
\[
\text{divloop}(\text{Coef}, \text{ST}, \text{Coef}, \text{ST}) \rightarrow \text{ST}
\]
\[
\text{divloop}(hx, tx, hy, ty) \equiv \text{YS}(s \rightarrow \text{divloope}(hx, tx, hy, ty, s))
\]
\[
\text{sdiv}(\text{ST}, \text{ST}) \rightarrow \text{ST}
\]
\[
\text{sdiv}(x, y) \equiv \text{delay}
\]
\[
\text{empty? } x \Rightarrow \text{empty()}
\]
\[
\text{empty? } y \Rightarrow \text{error "stream division by zero"}
\]
\[
hx := \text{frst } x; \quad tx := \text{rst } x
\]
\[
hy := \text{frst } y; \quad ty := \text{rst } y
\]
\[
\text{zero? } hy \Rightarrow
\]
\[
\text{zero? } hx \Rightarrow \text{sdiv}(tx, ty)
\]
\[
\text{error "stream division by zero"}
\]
\[
rhy := \text{recip} hy
\]
\[
rhy \text{ case "failed" } \Rightarrow \text{error "stream division: no reciprocal"}
\]
\[
\text{divloop}(hx, tx, rhy::\text{Coef}, ty)
\]
\[
\text{fixedPointExquo}(f, g) \equiv \text{series sdiv(\text{coefficients } f, \text{coefficients } g)}
\]

-- first order
\[
\text{ode1re}: (\text{ST} \rightarrow \text{ST}, \text{Coef}, \text{ST}) \rightarrow \text{ST}
\]
\[
\text{ode1re}(f, c, y) \equiv \text{lazyIntegrate}(c, f y)\text{STT}
\]
\[
\text{iOde1: } ((\text{ST} \rightarrow \text{ST}), \text{Coef}) \rightarrow \text{ST}
\]
\[
\text{iOde1}(f, c) \equiv \text{YS}(s \rightarrow \text{ode1re}(f, c, s))
\]
\[
\text{ode1}(f, c) \equiv \text{series iOde1(stFunc1 f, c)}
\]

-- second order
\[
\text{ode2re}: ((\text{ST, ST}) \rightarrow \text{ST}, \text{Coef}, \text{Coef}, \text{ST}) \rightarrow \text{ST}
\]
\[
\text{ode2re}(f, c0, c1, y) = \\
\text{yi} := \text{lazyIntegrate}(c1, f(y, \text{deriv}(y)\text{STT}))\text{STT}
\text{lazyIntegrate}(c0, yi)\text{STT}
\]
\[
\text{iOde2: } ((\text{ST, ST}) \rightarrow \text{ST}, \text{Coef}, \text{Coef}) \rightarrow \text{ST}
\]
\[
\text{iOde2}(f, c0, c1) \equiv \text{YS}(s \rightarrow \text{ode2re}(f, c0, c1, s))
\]
\[
\text{ode2}(f, c0, c1) \equiv \text{series iOde2(stFunc2 f, c0, c1)}
\]

-- nth order
\[
\text{odeNre}: (\text{List ST} \rightarrow \text{ST}, \text{List Coef}, \text{List ST}) \rightarrow \text{List ST}
\]
\[
\text{odeNre}(f, cl, yl) = \\
\]
--- yl is \([y, y', \ldots, y^{<n>}]\)
--- integrate \([y', \ldots, y^{<n>}]\) to get \([y, \ldots, y^{<n-1>}]\)
\[ yil := \text{lazyIntegrate}(c, y)^{\text{STT}} \text{ for } c \text{ in } cl \text{ for } y \text{ in } \text{rest} \ yl \]
--- use \(y^{<n>} = f(y, \ldots, y^{<n-1>}]\)
\[
\text{concat}(yil, [f \ yil])
\]

\[
i\text{Ode}: ((\text{L ST}) \rightarrow \text{ST}, \text{List Coef}) \rightarrow \text{ST}
i\text{Ode}(f, cl) \equiv \text{first YS}(ls \rightarrow \text{odeNre}(f, cl, ls), #cl + 1)
\]

\[
o\text{de}(f, cl) \equiv \text{series } i\text{Ode}(\text{stFunctN f, cl})
\]

\[
s\text{imulre}: (\text{L Coef}, \text{L ((L ST) \rightarrow ST), L ST}) \rightarrow \text{L ST}
s\text{imulre}(\text{cst}, \text{lfs}, c) \equiv
\[
\begin{align*}
&\text{lazyIntegrate}(\text{csti}, \text{lsfi} \text{ concat(monom}(1,1)^{\text{STT}}, c))_c \\
&\text{for } \text{csti in } \text{cst} \text{ for } \text{lsfi in } \text{lfs}
\end{align*}
\]
\[
i\text{psode}: (\text{L Coef}, \text{L ((L ST) \rightarrow ST)}) \rightarrow \text{L ST}
i\text{psode}(\text{cst}, \text{lsts}) \equiv \text{YS}(ls \rightarrow \text{simulre}(\text{cs}, \text{lsts}, ls), #c)
\]

\[
\text{mpsode}(\text{cs}, \text{lsts}) \equiv
\]

\[
\begin{align*}
&\text{stSol := i\text{psode}(\text{cs}, \text{map(\text{stFunctN}, \text{lsts})^L}(\text{FN}, (\text{L ST}) \rightarrow \text{ST})))} \\
&\text{stSol := i\text{psode}(\text{cs}, [\text{stFunctN(list) for } \text{lst in } \text{lsts}])} \\
&\text{map(series, stSol)^L}(\text{ST}, \text{UTS})
\end{align*}
\]

---

--- UTSODE.dotabb ---

"UTSODE" [color="#FF4488", href="bookvol10.4.pdf#nameddest=UTSODE"]
"UTSCAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UTSODE" \rightarrow "UTSCAT"

---

package UNISEG2 UniversalSegmentFunctions2

--- UniversalSegmentFunctions2.input ---

)set break resume
)sys rm -f UniversalSegmentFunctions2.output
)spool UniversalSegmentFunctions2.output
)set message test on
)set message auto off
)clear all

-- S 1 of 1
\texttt{UniversalSegmentFunctions2}\help

\begin{quote}
\textbf{UniversalSegmentFunctions2 examples}
\end{quote}

This package provides operations for mapping functions onto segments.

\textbf{See Also:}
\begin{quote}
\texttt{\texttt{UniversalSegmentFunctions2}}
\end{quote}
Exports:
map

--- package UNISEG2 UniversalSegmentFunctions2 ---

)abbrev package UNISEG2 UniversalSegmentFunctions2
++ Date Last Updated: June 4, 1991
++ Description:
++ This package provides operations for mapping functions onto segments.

UniversalSegmentFunctions2(R:Type, S:Type): with
  map: (R -> S, UniversalSegment R) -> UniversalSegment S
    ++ map(f,seg) returns the new segment obtained by applying
    ++ f to the endpoints of seg.

  if R has OrderedRing then
    map: (R -> S, UniversalSegment R) -> Stream S
      ++ map(f,s) expands the segment s, applying \spad{f} to each value.

== add
map(f:R -> S, u:UniversalSegment R):UniversalSegment S ==
  s := f lo u
  hasHi u => segment(s, f hi u)
  segment s

  if R has OrderedRing then
    map(f:R -> S, u:UniversalSegment R): Stream S ==
      map(f, expand u)$StreamFunctions2(R, S)

---

--- UNISEG2.dotabb ---

"UNISEG2" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UNISEG2"]
"OCAMON" [color="#4488FF",href="bookvol10.2.pdf#nameddest=OCAMON"]
"UNISEG2" -> "OCAMON"

---

package UDPO UserDefinedPartialOrdering

--- UserDefinedPartialOrdering.input ---
UserDefinedPartialOrdering is a package constructor
Abbreviation for UserDefinedPartialOrdering is UDPO
This constructor is not exposed in this frame.
Issue )edit bookvol10.4.pamphlet to see algebra source code for UDPO

--R
--R--------------------------------------------------------------- Operations --------------------------------
--R setOrder : List(S) -> Void
--R userOrdered? : () -> Boolean
--R getOrder : () -> Record(low: List(S), high: List(S))
--R largest : (List(S), ((S, S) -> Boolean)) -> S
--R largest : List(S) -> S if S has ORDSET
--R less? : (S, S) -> Union(Boolean, "failed")
--R more? : (S, S) -> Boolean if S has ORDSET

--E 1

UserDefinedPartialOrdering examples

Provides functions to force a partial ordering on any set.

See Also:
  o )show UserDefinedPartialOrdering
UserDefinedPartialOrdering (UDPO)

Exports:
setOrder userOrdered? getOrder largest less? more?

— package UDPO UserDefinedPartialOrdering —

)abbrev package UDPO UserDefinedPartialOrdering
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ Provides functions to force a partial ordering on any set.

UserDefinedPartialOrdering(S:SetCategory): with
  setOrder : List S -> Void
    ++ setOrder([a1,...,an]) defines a partial ordering on S given by:
    ++ (1) \spad{a1 < a2 < ... < an}.
    ++ (2) \spad{b < a_i} for \spad{i = 1..n} and b not among the a_i's.
    ++ (3) undefined on \spad{\{b, c\}} if neither is among the a_i's.
  setOrder : (List S, List S) -> Void
    ++ setOrder([b1,...,bm], [a1,...,an]) defines a partial 
    ++ ordering on S given by:
    ++ (1) \spad{b1 < b2 < ... < bm < a1 < a2 < ... < an}.
    ++ (2) \spad{bj < c < ai} for c not among the a_i's and bj's.
    ++ (3) undefined on \spad{\{c,d\}} if neither is among the a_i's,bj's.
  getOrder : () -> Record(low: List S, high: List S)
    ++ getOrder() returns \spad{\{[b1,...,bm], [a1,...,an]\}} such that the 
    ++ partial ordering on S was given by
    ++ \spad{setOrder([b1,...,bm],[a1,...,an])}.
  less? : (S, S) -> Union(Boolean, "failed")
    ++ less?(a, b) compares \spad{a} and b in the partial ordering induced by
    ++ setOrder.
  less? : (S, S, (S, S) -> Boolean) -> Boolean
    ++ less?(a, b, fn) compares \spad{a} and b in the partial ordering induced
    ++ by setOrder, and returns \spad{fn(a, b)} if \spad{a}
    ++ and b are not comparable
++ in that ordering.

largest : (List S, (S, S) -> Boolean) -> S
++ largest(l, fn) returns the largest element of l where the partial
++ ordering induced by setOrder is completed into a total one by fn.

userOrdered?: () -> Boolean
++ userOrdered?() tests if the partial ordering induced by
++ setOrder is not empty.

if S has OrderedSet then
largest: List S -> S
++ largest l returns the largest element of l where the partial
++ ordering induced by setOrder is completed into a total one by
++ the ordering on S.

more? : (S, S) -> Boolean
++ more?(a, b) compares a and b in the partial ordering induced
++ by setOrder, and uses the ordering on S if a and b are not
++ comparable in the partial ordering.

== add
llow :Reference List S := ref nil()
lhigh:Reference List S := ref nil()

userOrdered?() == not(empty? deref llow) or not(empty? deref lhigh)
getOrder() == [deref llow, deref lhigh]
setOrder l == setOrder(nil(), l)

setOrder(l, h) ==
  setref(llow, removeDuplicates l)
  setref(lhigh, removeDuplicates h)
  void

less?(a, b, f) ==
  (u := less?(a, b)) case "failed" => f(a, b)
u::Boolean

largest(x, f) ==
  empty? x => error "largest: empty list"
  empty? rest x => first x
  a := largest(rest x, f)
  less?(first x, a, f) => a
  first x

less?(a, b) ==
  for x in deref llow repeat
    x = a => return(a ^= b)
    x = b => return false
  aa := bb := false$Boolean
  for x in deref lhigh repeat
    if x = a then
      bb => return false
    aa := true
if $x = b$ then
  $aa$ => return($a ^= b$)
  $bb := true$
$aa$ => false
$bb$ => true
"failed"

if $S$ has OrderedSet then
  more?(a, b) == not less?(a, b, (y,z) +-> y <$S z$)
largest x == largest(x, (y,z) +-> y <$S z$)

---

UDPO.dotabb

"UDPO" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UDPO"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"UDPO" -> "FLAGG"

---

package UDVO UserDefinedVariableOrdering

--- UserDefinedVariableOrdering.input ---

)set break resume
)sys rm -f UserDefinedVariableOrdering.output
)spool UserDefinedVariableOrdering.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UserDefinedVariableOrdering
--R
--R UserDefinedVariableOrdering is a package constructor
--R Abbreviation for UserDefinedVariableOrdering is UDVO
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UDVO
--R
--R------------------------ Operations ------------------------
--R resetVariableOrder : () -> Void
--R getVariableOrder : () -> Record(high: List(Symbol),low: List(Symbol))
--R setVariableOrder : List(Symbol) -> Void
CHAPTER 22. CHAPTER U

--R setVariableOrder : (List(Symbol),List(Symbol)) -> Void
--R
--E 1

)spool
)lisp (bye)

---

--- UserDefinedVariableOrdering.help ---

UserDefinedVariableOrdering examples

This packages provides functions to allow the user to select the ordering on the variables and operators for displaying polynomials, fractions and expressions. The ordering affects the display only and not the computations.

See Also:
o )show UserDefinedVariableOrdering

---

UserDefinedVariableOrdering (UDVO)

Exports:
resetVariableOrder  getVariableOrder  setVariableOrder

--- package UDVO UserDefinedVariableOrdering ---

)abbrev package UDVO UserDefinedVariableOrdering
++ Author: Manuel Bronstein
++ Date Created: March 1990
++ Date Last Updated: 9 April 1991
++ Description:
++ This packages provides functions to allow the user to select the ordering
++ on the variables and operators for displaying polynomials,
++ fractions and expressions. The ordering affects the display
++ only and not the computations.

UserDefinedVariableOrdering(): with
  setVariableOrder : List Symbol -> Void
    ++ setVariableOrder([a1,...,an]) defines an ordering on the
    ++ variables given by `spad{a1 > a2 > ... > an > other variables}`.
  setVariableOrder : (List Symbol, List Symbol) -> Void
    ++ setVariableOrder([b1,...,bm], [a1,...,an]) defines an ordering
    ++ on the variables given by
    ++ `spad{b1 > b2 > ... > bm >} other variables \spad{> a1 > a2 > ... > an}`.
  getVariableOrder : () -> Record(high:List Symbol, low:List Symbol)
    ++ getVariableOrder() returns `\spad{[[b1,...,bm], [a1,...,an]]}` such that
    ++ the ordering on the variables was given by
    ++ `spad{setVariableOrder([b1,...,bm], [a1,...,an])}`.
  resetVariableOrder: () -> Void
    ++ resetVariableOrder() cancels any previous use of
    ++ setVariableOrder and returns to the default system ordering.

== add
import UserDefinedPartialOrdering(Symbol)

setVariableOrder l == setOrder reverse l
setVariableOrder(l1, l2) == setOrder(reverse l2, reverse l1)
resetVariableOrder() == setVariableOrder(nil(), nil())

getVariableOrder() ==
  r := getOrder()
  [reverse(r.high), reverse(r.low)]
package UTSODETL UTSodetools

— UTSodetools.input —

)set break resume
)sys rm -f UTSodetools.output
)spool UTSodetools.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show UTSodetools
--R
--R UTSodetools(F: Ring, UP: UnivariatePolynomialCategory(F), L: LinearOrdinaryDifferentialOperatorCategory(UP), UTS: UnivariateTaylorSeriesCategory(F)) is a package constructor
--R Abbreviation for UTSodetools is UTSODETL
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for UTSODETL
--R
--R------------------------------- Operations --------------------------------
--R LODO2FUN : L -> (List(UTS) -> UTS)  UP2UTS : UP -> UTS
--R RF2UTS : Fraction(UP) -> UTS if F has INTDOM
--R UTS2UP : (UTS,NonNegativeInteger) -> UP
--R
--E 1

)spool
)lisp (bye)

— UTSodetools.help —

====================================================================
UTSodetools examples
====================================================================

UTSodetools provides tools to interface with the series ODE solver when presented with linear ODEs.

See Also:
 o )show UTSodetools
UTSodetools (UTSODETL)

Exports:
LODO2FUN  RF2UTS  UTS2UP

— package UTSODETL UTSodetools —

)abbrev package UTSODETL UTSodetools
++ Author: Manuel Bronstein
++ Date Created: 31 January 1994
++ Date Last Updated: 3 February 1994
++ Description:
++ \spad{RUTSodetools} provides tools to interface with the series
++ ODE solver when presented with linear ODEs.

UTSodetools(F, UP, L, UTS): Exports == Implementation where
  F : Ring
  UP : UnivariatePolynomialCategory F
  L : LinearOrdinaryDifferentialOperatorCategory UP
  UTS: UnivariateTaylorSeriesCategory F

Exports ==> with
  UP2UTS: UP -> UTS
    ++ UP2UTS(p) converts \spad{p} to a Taylor series.
  UTS2UP: (UTS, NonNegativeInteger) -> UP
    ++ UTS2UP(s, n) converts the first \spad{n} terms of \spad{s}
    ++ to a univariate polynomial.
  LOD2FUN: L -> (List UTS -> UTS)
    ++ LOD2FUN(op) returns the function to pass to the series ODE
    ++ solver in order to solve \spad{op y = 0}.
  if F has IntegralDomain then
    RF2UTS: Fraction UP -> UTS
      ++ RF2UTS(f) converts \spad{f} to a Taylor series.

Implementation ==> add
  fun: (Vector UTS, List UTS) -> UTS
CHAPTER 22. CHAPTER U

\[
\text{UTS2UP}(s, n) \equiv \\
\begin{align*}
\text{xmc} & := \text{monomial}(1, 1)\text{UP} - \text{center}(0)\text{UP} \\
\text{xmcn} \cdot \text{UP} & := 1 \\
\text{ans} \cdot \text{UP} & := 0 \\
\text{for } i \text{ in } 0..n \text{ repeat} \\
\text{ans} & := \text{ans} + \text{coefficient}(s, i) \cdot \text{xmcn} \\
\text{xmcn} & := \text{xmc} \cdot \text{xmcn} \\
\text{ans} 
\end{align*}
\]

\[
\text{LODO2FUN} \text{ op} \equiv \\
\begin{align*}
\text{a} & := \text{recip}(\text{UP2UTS}(-\text{leadingCoefficient op}))\text{UTS} \\
\text{n} & := (\text{degree(op)} - 1)\text{NonNegativeInteger} \\
\text{v} & := [\text{a} \cdot \text{UP2UTS coefficient(op, i)} \text{for } i \text{ in } 0..n]\text{Vector(UTS)} \\
\text{r} & := (l1: \text{List UTS}) : \text{UTS} +\rightarrow \text{fun}(v, l1) \\
\text{r} \\
\text{fun}(v, l) \equiv \\
\begin{align*}
\text{ans} \cdot \text{UTS} & := 0 \\
\text{for } b \text{ in } l \text{ for } i \text{ in } 1.. \text{repeat} \text{ans} := \text{ans} + v.i \cdot b \\
\text{ans} 
\end{align*}
\]

\[
\text{if } F \text{ has IntegralDomain then} \\
\text{RF2UTS} \text{ f} \equiv \text{UP2UTS}(\text{numer f}) \ast \text{recip}(\text{UP2UTS deno f})\text{UTS}
\]

——

|UTSODETL.dotabb|

"UTSODETL" [color="#FF4488",href="bookvol10.4.pdf#nameddest=UTSODETL"]
"UTSCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=UTSCAT"]
"UTSODETL" -> "UTSCAT"

——

package POLYVEC U32VectorPolynomialOperations

—— U32VectorPolynomialOperations.input ——

)set break resume
)sys rm -f U32VectorPolynomialOperations.output
U32VectorPolynomialOperations help

U32VectorPolynomialOperations examples
This is a low-level package which implements operations on vectors treated as univariate modular polynomials. Most operations takes modulus as parameter. Modulus is machine sized prime which should be small enough to avoid overflow in intermediate calculations.

See Also:
- )show U32VectorPolynomialOperations

---

**U32VectorPolynomialOperations (POLYVEC)**

Exports:
- copyfirst
- copieslice
- degree
- differentiate
- differentiate
- divide!
- evalat
- extendedgcd
- gcd
- lcm
- mul
- mulbybinomial
- mulbybinomial
- mulbyscale
- pow
- remainder!
- resultant
- tomodpa
- truncatedmuladd
- truncatedmultiplication
- vectoraddmul
- vectorcombination

---

)abbrev package POLYVEC U32VectorPolynomialOperations
++ Description:
++ This is a low-level package which implements operations
++ on vectors treated as univariate modular polynomials. Most
++ operations takes modulus as parameter. Modulus is machine
++ sized prime which should be small enough to avoid overflow
++ in intermediate calculations.
U32VectorPolynomialOperations() : Export == Implementation where
PA =>> U32Vector
Export =>> with
  copy_first : (PA, PA, Integer) -> Void
  ++ copy_first(v1, v2, n) copies first n elements
  ++ of v2 into n first positions in v1.
copy_slice : (PA, PA, Integer, Integer) -> Void
  ++ copy_first(v1, v2, m, n) copies the slice of v2 starting
  ++ at m elements and having n elements into corresponding
  ++ positions in v1.
eval_at : (PA, Integer, Integer, Integer) -> Integer
  ++ eval_at(v, deg, pt, p) treats v as coefficients of
  ++ polynomial of degree deg and evaluates the
  ++ polynomial at point pt modulo p
  ++
  ++ X a:=new(3,1)$U32VEC
  ++ X a.1:=2
  ++ X eval_at(a,2,3,1024)
  ++ X eval_at(a,2,2,8)
  ++ X eval_at(a,2,3,10)
vector_add_mul : (PA, PA, Integer, Integer, Integer, Integer) _
  -> Void
  ++ vector_add_mul(v1, v2, m, n, c, p) sets v1(m), ..., vv1(n) to corresponding entries in v1 + c*v2
  ++ modulo p.
mul_by_binomial : (PA, Integer, Integer) -> Void
  ++ mul_by_binomial(v, pt, p) treats v a polynomial
  ++ and multiplies in place this polynomial by binomial (x + pt).
  ++ Highest coefficient of product is ignored.
mul_by_binomial : (PA, Integer, Integer, Integer) -> Void
  ++ mul_by_binomial(v, deg, pt, p) treats v as
  ++ coefficients of polynomial of degree deg and
  ++ multiplies in place this polynomial by binomial (x + pt).
  ++ Highest coefficient of product is ignored.
mul_by_scalar : (PA, Integer, Integer, Integer) -> Void
  ++ mul_by_scalar(v, deg, c, p) treats v as
  ++ coefficients of polynomial of degree deg and
  ++ multiplies in place this polynomial by scalar c
mul : (PA, PA, Integer) -> PA
  ++ Polynomial multiplication.
truncated_multiplication : (PA, PA, Integer, Integer) -> PA
  ++ truncated_multiplication(x, y, d, p) computes
  ++ x*y truncated after degree d
truncated_mul_add : (PA, PA, PA, Integer, Integer) -> Void
  ++ truncated_mul_add(x, y, z, d, p) adds to z
  ++ the product x*y truncated after degree d
pow : (PA, PositiveInteger, NonNegativeInteger, Integer) -> PA
  ++ pow(u, n, d, p) returns u^n truncated after degree d, except if
  ++ n=1, in which case u itself is returned
differentiate : (PA, Integer) -> PA
CHAPTER 22. CHAPTER U

++ Polynomial differentiation.

differentiate : (PA, NonNegativeInteger, Integer) -> PA
++ Polynomial differentiation.

divide! : (PA, PA, PA, Integer) -> Void
++ Polynomial division.

remainder! : (PA, PA, Integer) -> Void
++ Polynomial remainder

vector_combination : (PA, Integer, PA, Integer, Integer, Integer, Integer, p : Integer) -> Void
++ vector_combination(v1, c1, v2, c2, n, delta, p) replaces
++ first n + 1 entries of v1 by corresponding entries of
++ c1*v1+c2*x^delta*v2 mod p.

to_mod_pa : (SparseUnivariatePolynomial Integer, Integer) -> PA
++ to_mod_pa(s, p) reduces coefficients of polynomial
++ s modulo prime p and converts the result to vector

gcd : (PA, PA, Integer) -> PA
++ gcd(v1, v2, p) computes monic gcd of v1 and v2 modulo p.

gcd : (PrimitiveArray PA, Integer, Integer, Integer) -> PA
++ gcd(a, lo, hi, p) computes gcd of elements
++ a(lo), a(lo+1), ..., a(hi).

lcm : (PrimitiveArray PA, Integer, Integer, Integer) -> PA
++ lcm(a, lo, hi, p) computes lcm of elements
++ a(lo), a(lo+1), ..., a(hi).

degree : PA -> Integer
++ degree(v) is degree of v treated as polynomial

extended_gcd : (PA, PA, Integer) -> List(PA)
++ extended_gcd(v1, v2, p) gives [g, c1, c2] such
++ that g is \spad{gcd(v1, v2, p)}, \spad{g = c1*v1 + c2*v2}
++ and degree(c1) < max(degree(v2) - degree(g), 0) and
++ degree(c2) < max(degree(v1) - degree(g), 1)

resultant : (PA, PA, Integer) -> Integer
++ resultant(v1, v2, p) computes resultant of v1 and v2
++ modulo p.

Implementation ==> add

Qmuladdmod ==> QSMULADDMOD6432$Lisp
Qmuladd ==> QSMULADD6432$Lisp
Qmul  ==> QSMULMOD32$Lisp
Qdot2 ==> QSDOT2MOD6432$Lisp
Qrem  ==> QSMOD6432$Lisp
modInverse ==> invmod


copy_first(np : PA, op : PA, n : Integer) : Void ==
    ns := n pretend SingleInteger
    for j in 0..(ns - 1) repeat
        np(j) := op(j)

copy_slice(np : PA, op : PA, m : Integer, n : Integer) : Void ==
ms := m pretend SingleInteger
ns := n pretend SingleInteger
for j in ms..(ms + ns - 1) repeat
    np(j) := op(j)

eval_at(v: PA, deg: Integer, pt: Integer, _
    p: Integer) : Integer ==
    i : SingleInteger := deg::SingleInteger
    res : Integer := 0
    while not(i < 0) repeat
        res := Qmuladdmod(pt, res, v(i), p)
        i := i - 1
    res

to_mod_pa(s: SparseUnivariatePolynomial Integer, p: Integer) : PA ==
    zero?(s) => new(1, 0)$PA
    n0 := degree(s) pretend SingleInteger
    ncoeffs := new((n0+1) pretend NonNegativeInteger, 0)$PA
    while not(zero?(s)) repeat
        n := degree(s)
        ncoeffs(n) := positiveRemainder(leadingCoefficient(s), p)
        s := reductum(s)
    ncoeffs

vector_add_mul(v1: PA, v2: PA, m: Integer, n: Integer, _
    c: Integer, p: Integer) : Void ==
    ms := m pretend SingleInteger
    ns := n pretend SingleInteger
    for i in ms..ns repeat
        v1(i) := Qmuladdmod(c, v2(i), v1(i), p)

mul_by_binomial(v: PA, n: Integer, pt: Integer, _
    p: Integer) : Void ==
    prev_coeff : Integer := 0
    ns := n pretend SingleInteger
    for i in 0..(ns - 1) repeat
        pp := v(i)
        v(i) := Qmuladdmod(pt, pp, prev_coeff, p)
        prev_coeff := pp

mul_by_binomial(v: PA, pt: Integer, _
    p: Integer) : Void ==
    mul_by_binomial(v, #v, pt, p)

mul_by_scalar(v: PA, n: Integer, c: Integer, _
    p: Integer) : Void ==
    ns := n pretend SingleInteger
    for i in 0..ns repeat
        v(i) := Qmul(c, v(i), p)
degree(v : PA) : Integer ==
  n := #v
  for i in (n - 1)..<0 by -1 repeat
    not(v(i) = 0) => return i
  -1

vector_combination(v1 : PA, c1 : Integer,_
                  v2 : PA, c2 : Integer, _
                  n : Integer, delta : Integer, _
                  p : Integer) : Void ==
  ns := n pretend SingleInteger
  ds := delta pretend SingleInteger
  if not(c1 = 1) then
    ns + 1 < ds =>
      for i in 0..ns repeat
        v1(i) := Qmul(v1(i), c1, p)
    for i in 0..(ds - 1) repeat
      v1(i) := Qmul(v1(i), c1, p)
    for i in ds..ns repeat
      v1(i) := Qdot2(v1(i), c1, v2(i - ds), c2, p)
  else
    for i in ds..ns repeat
      v1(i) := Qmuladdmod(c2, v2(i - ds), v1(i), p)

divide!(r0 : PA, r1 : PA, res : PA, p: Integer) : Void ==
  dr0 := degree(r0) pretend SingleInteger
  dr1 := degree(r1) pretend SingleInteger
  c0 := r1(dr1)
  c0 := modInverse(c0, p)
  while not(dr0 < dr1) repeat
    delta := dr0 - dr1
    c1 := Qmul(c0, r0(dr0), p)
    res(delta) := c1
    c1 := p - c1
    r0(dr0) := 0
    dr0 := dr0 - 1
    if dr0 < 0 then break
    vector_combination(r0, 1, r1, c1, dr0, delta, p)
  while r0(dr0) = 0 repeat
    dr0 := dr0 - 1
    if dr0 < 0 then break

remainder!(r0 : PA, r1 : PA, p: Integer) : Void ==
  dr0 := degree(r0) pretend SingleInteger
  dr1 := degree(r1) pretend SingleInteger
  c0 := r1(dr1)
  c0 := modInverse(c0, p)
  while not(dr0 < dr1) repeat
    delta := dr0 - dr1
    c1 := Qmul(c0, r0(dr0), p)
c1 := p - c1
r0(dr0) := 0
dr0 := dr0 - 1
if dr0 < 0 then break
vector_combination(r0, 1, r1, c1, dr0, delta, p)
while r0(dr0) = 0 repeat
  dr0 := dr0 - 1
  if dr0 < 0 then break

gcd(x : PA, y : PA, p : Integer) : PA ==
dr0 := degree(y) pretend SingleInteger
dr1 : SingleInteger
if dr0 < 0 then
  tmpp := x
  x := y
  y := tmpp
dr1 := dr0
dr0 := degree(y) pretend SingleInteger
else
  dr1 := degree(x) pretend SingleInteger
dr0 < 0 => return new(1, 0)$PA
r0 := new((dr0 + 1) pretend NonNegativeInteger, 0)$PA
copy_first(r0, y, dr0 + 1)
dr1 < 0 =>
c := r0(dr0)
c := modInverse(c, p)
mul_by_scalar(r0, dr0, c, p)
return r0
r1 := new((dr1 + 1) pretend NonNegativeInteger, 0)$PA
copy_first(r1, x, dr1 + 1)
while 0 < dr1 repeat
  while not(dr0 < dr1) repeat
    delta := dr0 - dr1
c1 := sub_SI(p, r0(dr0))$Lisp
c0 := r1(dr1)
    if c0 ≠ 1 and delta > 30 then
      c0 := modInverse(c0, p)
mul_by_scalar(r1, dr1, c0, p)
c0 := 1
r0(dr0) := 0
dr0 := dr0 - 1
vector_combination(r0, c0, r1, c1, dr0, delta, p)
while r0(dr0) = 0 repeat
  dr0 := dr0 - 1
  if dr0 < 0 then break
tmpp := r0
tmp := dr0
r0 := r1
dr0 := dr1
r1 := tmpp
\[
\begin{align*}
\text{dr1} & := \text{tmp} \\
\text{not}(\text{dr1} < 0) & => \\
\text{r1}(0) & := 1 \\
\text{return r1} \\
\text{c} & := \text{r0}(\text{dr0}) \\
\text{c} & := \text{modInverse}(\text{c}, \text{p}) \\
\text{mul_by_scalar}(\text{r0}, \text{dr0}, \text{c}, \text{p}) \\
\text{r0}
\end{align*}
\]

\[
\text{gcd}(a : \text{PrimitiveArray PA, lo : Integer, hi: Integer, p: Integer}) \_ \\
\text{: PA} == \\
\text{res} := a(\text{lo}) \\
\text{for i in (lo + 1)\ldots hi repeat} \\
\text{res := gcd}(a(i), \text{res}, \text{p}) \\
\text{res}
\]

\[
\text{lcm2}(v1 : \text{PA, v2 : PA, p : Integer}) : \text{PA} == \\
\text{pp} := \text{gcd}(v1, v2, p) \\
\text{dv2} := \text{degree}(v2) \\
\text{dpp} := \text{degree}(\text{pp}) \\
\text{dv2} = \text{dpp} => \\
\text{v1} \\
\text{dpp} = 0 => \text{mul}(v1, v2, p) \\
\text{tmpl} := \text{new}((\text{dv2} + 1) \text{ pretend NonNegativeInteger, 0})$PA \\
\text{tmp2} := \text{new}((\text{dv2} - \text{dpp} + 1) \text{ pretend NonNegativeInteger, 0})$PA \\
\text{copy_first}(\text{tmpl}, v2, \text{dv2} + 1) \\
\text{divide!}(\text{tmpl}, \text{pp}, \text{tmp2}, \text{p}) \\
\text{mul}(v1, \text{tmp2}, \text{p})
\]

\[
\text{lcm}(a : \text{PrimitiveArray PA, lo : Integer, hi: Integer, p: Integer}) \_ \\
\text{: PA} == \\
\text{res} := a(\text{lo}) \\
\text{for i in (lo + 1)\ldots hi repeat} \\
\text{res := lcm2}(a(i), \text{res}, \text{p}) \\
\text{res}
\]

\[
\text{inner_mul} : (\text{PA, PA, PA, SingleInteger, SingleInteger, \_ \_ SingleInteger, \_ \_ Integer}) \rightarrow \text{Void}
\]

\[
\text{mul}(x : \text{PA, y : PA, p : Integer}) : \text{PA} == \\
\text{xdeg} := \text{degree}(x) \text{ pretend SingleInteger} \\
\text{ydeg} := \text{degree}(y) \text{ pretend SingleInteger} \\
\text{if xdeg > ydeg then} \\
\text{tmpp} := x \\
\text{tmp} := xdeg \\
\text{x} := y \\
\text{xdeg} := ydeg \\
\text{y} := \text{tmpp} \\
\text{ydeg} := \text{tmp} \\
\text{xcoeffs} := x
\]
ycoeffs := y
xdeg < 0 => x
xdeg = 0 and xcoeffs(0) = 1 => copy(y)
zdeg : SingleInteger := xdeg + ydeg
zdeg0 := ((zdeg + 1)::Integer) pretend NonNegativeInteger
zcoeffs := new(zdeg0, 0)$PA
inner_mul(xcoeffs, ycoeffs, zcoeffs, xdeg, ydeg, zdeg, p)

inner_mul(x, y, z, xdeg, ydeg, zdeg, p) ==
if ydeg < xdeg then
  tmpp := x
tmp := xdeg
x := y
xdeg := ydeg
y := tmpp
ydeg := tmp
xdeg :=
zdeg < xdeg => zdeg
xdeg
ydeg :=
zdeg < ydeg => zdeg
ydeg
ss : Integer
i : SingleInteger
j : SingleInteger
for i in 0..xdeg repeat
  ss := z(i)
  for j in 0..i repeat
    ss := Qmuladd(x(i - j), y(j), ss)
  z(i) := Qrem(ss, p)
for i in (xdeg+1)..ydeg repeat
  ss := z(i)
  for j in 0..xdeg repeat
    ss := Qmuladd(x(j), y(i-j), ss)
  z(i) := Qrem(ss, p)
for i in (ydeg+1)..zdeg repeat
  ss := z(i)
  for j in (i-xdeg)..ydeg repeat
    ss := Qmuladd(x(i - j), y(j), ss)
  z(i) := Qrem(ss, p)

truncated_mul_add(x, y, z, m, p) ==
xdeg := (#x - 1) pretend SingleInteger
ydeg := (#y - 1) pretend SingleInteger
inner_mul(x, y, z, xdeg, ydeg, m pretend SingleInteger, p)

truncated_multiplication(x, y, m, p) ==
xdeg := (#x - 1) pretend SingleInteger
ydeg := (#y - 1) pretend SingleInteger
z := new((m pretend SingleInteger + 1)
pretend NonNegativeInteger, 0)$PA
inner_mul(x, y, z, xdeg, ydeg, m pretend SingleInteger, p)
z

pow(x : PA, n : PositiveInteger, d: NonNegativeInteger, _
p : Integer) : PA ==
one? n => x
odd?(n)$Integer =>
  truncated_multiplication(x,
pow(truncated_multiplication(x, x, d, p),
shift(n,-1) pretend PositiveInteger,
d,
p),
d,p)
pow(truncated_multiplication(x, x, d, p),
shift(n,-1) pretend PositiveInteger,
d,p)

differentiate(x: PA, p: Integer): PA ==
d := #x - 1
if zero? d then empty()$PA
else
  r := new(d::NonNegativeInteger, 0)$PA
  for i in 0..d-1 repeat
    i1 := i+1
    r.i := Qmul(i1, x.i1, p)
  r

differentiate(x: PA, n: NonNegativeInteger, p: Integer): PA ==
zero? n => x
d := #x - 1
if d < n then empty()$PA
else
  r := new((d-n+1) pretend NonNegativeInteger, 0)$PA
  for i in n..d repeat
    j := i-n
    f := j+1
    for k in j+2..i repeat f := Qmul(f, k, p)
    r.(j pretend NonNegativeInteger) := Qmul(f, x.i, p)
  r

extended_gcd(x : PA, y : PA, p : Integer) : List(PA) ==
dr0 := degree(x) pretend SingleInteger
dr1 : SingleInteger
swapped : Boolean := false
t0 : PA
if dr0 < 0 then
\[(x, y) := (y, x)\]
\[\text{dr1} := \text{dr0}\]
\[\text{dr0} := \text{degree}(x) \text{ pretend SingleInteger}\]
\[\text{swapped} := \text{true}\]

else
\[\text{dr1} := \text{degree}(y) \text{ pretend SingleInteger}\]
\[\text{dr1} < 0 \implies \text{dr0} < 0 \implies \text{return} \begin{bmatrix} \text{new}(1, 0) \text{PA}, \text{new}(1, 0) \text{PA}, \text{new}(1, 1) \text{PA} \end{bmatrix}\]
\[\text{r0} := \text{new}((\text{dr0} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{copy_first}(\text{r0}, x, \text{dr0} + 1)\]
\[c := \text{r0}(...), c := \text{modInverse}(c, p)\]
\[\text{mul_by_scalar}(\text{r0}, \text{dr0}, c, p)\]
\[t0 := \text{new}(1, c) \text{PA}\]
\[\text{if swapped then}\]
\[\text{return} \begin{bmatrix} \text{r0}, \text{new}(1, 0) \text{PA}, t0 \end{bmatrix}\]
else
\[\text{swapped} \implies \text{return} \begin{bmatrix} \text{r0}, t0, \text{new}(1, 0) \text{PA} \end{bmatrix}\]
\[\text{dt} := (\text{dr0} > 0 \implies \text{dr0} - 1 ; 0)\]
\[\text{ds} := (\text{dr1} > 0 \implies \text{dr1} - 1 ; 0)\]
\[\text{invariant: } \text{r0} = s0*x + t0*y, \text{r1} = s1*x + t1*y\]
\[\text{r0} := \text{new}((\text{dr0} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{t0} := \text{new}((\text{dt} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{s0} := \text{new}((\text{ds} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{copy_first}(\text{r0}, x, \text{dr0} + 1)\]
\[\text{s0}(0) := 1\]
\[\text{r1} := \text{new}((\text{dr1} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{t1} := \text{new}((\text{dt} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{s1} := \text{new}((\text{ds} + 1) \text{ pretend NonNegativeInteger}, 0) \text{PA}\]
\[\text{copy_first}(\text{r1}, y, \text{dr1} + 1)\]
\[\text{t1}(0) := 1\]
while \text{dr1} > 0 repeat
while \text{dr0} > 0 repeat
\[\text{delta} := \text{dr0} - \text{dr1}\]
\[\text{c1} := \text{sub_SI}(p, \text{r0}(\text{dr0})) \text{Lisp}\]
\[\text{c0} := \text{r1}(...), \text{if c0} \neq 1 \text{ and delta} > 30 \text{ then}\]
\[\text{c0} := \text{modInverse}(c0, p)\]
\[\text{mul_by_scalar}(\text{r1}, \text{dr1}, c0, p)\]
\[\text{mul_by_scalar}(\text{t1}, \text{dt}, c0, p)\]
\[\text{mul_by_scalar}(\text{s1}, \text{ds}, c0, p)\]
\[\text{c0} := 1\]
\[\text{r0}(\text{dr0}) := 0\]
\[\text{dr0} := \text{dr0} - 1\]
\[\text{vector_combination}(\text{r0}, \text{c0}, \text{r1}, \text{c1}, \text{dr0}, \text{delta}, \text{p})\]
\[\text{vector_combination}(\text{t0}, \text{c0}, \text{t1}, \text{c1}, \text{dt}, \text{delta}, \text{p})\]
\[\text{vector_combination}(\text{s0}, \text{c0}, \text{s1}, \text{c1}, \text{ds}, \text{delta}, \text{p})\]
while \text{r0}(\text{dr0}) = 0 repeat
\begin{verbatim}

\text{dr0} := \text{dr0} - 1 \\
\text{if dr0} < 0 \text{ then break} \\
(r0, r1) := (r1, r0) \\
(dr0, dr1) := (dr1, dr0) \\
(s0, s1) := (s1, s0) \\
(t0, t1) := (t1, t0) \\
\text{if dr1} \geq 0 \Rightarrow \\
\text{c} := r1(0) \\
\text{c} := \text{modInverse}(c, p) \\
r1(0) := 1 \\
mul_by_scalar(s1, ds, c, p) \\
mul_by_scalar(t1, dt, c, p) \\
\text{return} [r1, s1, t1] \\
\text{c} := \text{r0(dr0)} \\
\text{c} := \text{modInverse}(c, p) \\
mul_by_scalar(r0, dr0, c, p) \\
mul_by_scalar(s0, ds, c, p) \\
mul_by_scalar(t0, dt, c, p) \\
[r0, s0, t0]
\end{verbatim}

\textbf{resultant}(x : PA, y : PA, p : Integer) : Integer == \\
\text{dr0} := \text{degree}(x) \text{ pretend SingleInteger} \\
\text{dr0} < 0 \Rightarrow 0 \\
\text{dr1} := \text{degree}(y) \text{ pretend SingleInteger} \\
\text{dr1} < 0 \Rightarrow 0 \\
r0 := \text{new}((\text{dr0} + 1) \text{ pretend NonNegativeInteger}, 0)$PA \\
\text{copy_first}(r0, x, \text{dr0} + 1) \\
r1 := \text{new}((\text{dr1} + 1) \text{ pretend NonNegativeInteger}, 0)$PA \\
\text{copy_first}(r1, y, \text{dr1} + 1) \\
\text{res} := 1 \\
\text{repeat} \\
\text{dr0} < \text{dr1} \Rightarrow \\
(r0, r1) := (r1, r0) \\
(dr0, dr1) := (dr1, dr0) \\
c0 := r1(dr1) \\
\text{dr1} = 0 \Rightarrow \\
\text{while} 0 < \text{dr0} \text{ repeat} \\
\text{res} := \text{Qmul}(\text{res}, c0, p) \\
\text{dr0} := \text{dr0} - 1 \\
\text{return} \text{res} \\
\text{delta} := \text{dr0} - \text{dr1} \\
c1 := \text{sub_SI}(p, r0(\text{dr0}))$Lisp \\
\text{if} c0 = 1 \text{ then} \\
c1 := \text{Qmul}(c1, \text{modInverse}(c0, p), p) \\
r0(\text{dr0}) := 0 \\
\text{dr0} := \text{dr0} - 1 \\
\text{vector_combination}(r0, 1, r1, c1, \text{dr0}, \text{delta}, p) \\
\text{res} := \text{Qmul}(\text{res}, c0, p) \\
\text{while} r0(\text{dr0}) = 0 \text{ repeat} \\
\text{dr0} := \text{dr0} - 1
\end{verbatim}
dr0 < 0 => return 0
res := Qmul(res, c0, p)
package VECTOR2 VectorFunctions2

— VectorFunctions2.input —

)set break resume
)sys rm -f VectorFunctions2.output
)spool VectorFunctions2.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show VectorFunctions2
--R
--R VectorFunctions2(A: Type,B: Type) is a package constructor
--R Abbreviation for VectorFunctions2 is VECTOR2
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for VECTOR2
--R
--R--------------------------------- Operations --------------------------------
--R map : ((A -> B), Vector(A)) -> Vector(B)
--R map : ((A -> Union(B,"failed")), Vector(A)) -> Union(Vector(B),"failed")
--R reduce : ((A,B) -> B), Vector(A), B) -> B
--R scan : (((A,B) -> B), Vector(A), B) -> Vector(B)
--R
--E 1

)spool
)lisp (bye)

———
VectorFunctions2 examples

This package provides operations which all take as arguments vectors of elements of some type \texttt{A} and functions from \texttt{A} to another of type \texttt{B}. The operations all iterate over their vector argument and either return a value of type \texttt{B} or a vector over \texttt{B}.

See Also:
o \texttt{)show VectorFunctions2}

Export:
map, reduce, scan

\begin{verbatim}
)abbrev package VECTOR2 VectorFunctions2
++ Description:
++ This package provides operations which all take as arguments
++ vectors of elements of some type \texttt{A} and functions from \texttt{A} to
++ another of type \texttt{B}. The operations all iterate over their vector argument
++ and either return a value of type \texttt{B} or a vector over \texttt{B}.

VectorFunctions2(A, B): Exports == Implementation where
A, B: Type
\end{verbatim}
VA ==> Vector A
VB ==> Vector B
O2 ==> FiniteLinearAggregateFunctions2(A, VA, B, VB)
UB ==> Union(B, "failed")

Exports ==> with
   scan ((A, B) -> B, VA, B) -> VB
   ++ scan(func, vec, ident) creates a new vector whose elements are
   ++ the result of applying reduce to the binary function func,
   ++ increasing initial subsequences of the vector vec,
   ++ and the element ident.
   reduce ((A, B) -> B, VA, B) -> B
   ++ reduce(func, vec, ident) combines the elements in vec using the
   ++ binary function func. Argument ident is returned if vec is empty.
   map : (A -> B, VA) -> VB
   ++ map(f, v) applies the function f to every element of the vector v
   ++ producing a new vector containing the values.
   map : (A -> UB, VA) -> Union(VB, "failed")
   ++ map(f, v) applies the function f to every element of the vector v
   ++ producing a new vector containing the values or \spad{"failed"}.

Implementation ==> add
   scan(f, v, b) == scan(f, v, b)$O2
   reduce(f, v, b) == reduce(f, v, b)$O2
   map(f:(A->B), v:VA):VB == map(f, v)$O2
   map(f:(A -> UB), a:VA):Union(VB, "failed") ==
      res : List B := []
      for u in entries(a) repeat
         r := f u
         r = "failed" => return "failed"
      res := [r::B,:res]
      vector reverse! res

<table>
<thead>
<tr>
<th>VECTOR2.dotabb</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;VECTOR2&quot; [color=#FF4488,href=bookvol10.4.pdf#nameddest=VECTOR2]</td>
</tr>
<tr>
<td>&quot;IVECTOR&quot; [color=#88FF44,href=bookvol10.3.pdf#nameddest=IVECTOR]</td>
</tr>
<tr>
<td>&quot;VECTOR2&quot; -&gt; &quot;IVECTO&quot;</td>
</tr>
</tbody>
</table>
package VIEWDEF ViewDefaultsPackage

— ViewDefaultsPackage.input —

)set break resume
)sys rm -f ViewDefaultsPackage.output
)spool ViewDefaultsPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ViewDefaultsPackage
--R
--R ViewDefaultsPackage is a package constructor
--R Abbreviation for ViewDefaultsPackage is VIEWDEF
--R This constructor is exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for VIEWDEF
--R
--R------------------------------- Operations --------------------------------
--R axesColorDefault : () -> Palette
--R lineColorDefault : () -> Palette
--R pointColorDefault : () -> Palette
--R unitsColorDefault : () -> Palette
--R viewDefaults : () -> Void
--R axesColorDefault : Palette -> Palette
--R lineColorDefault : Palette -> Palette
--R pointColorDefault : Palette -> Palette
--R pointSizeDefault : () -> PositiveInteger
--R pointSizeDefault : PositiveInteger -> PositiveInteger
--R tubePointsDefault : PositiveInteger -> PositiveInteger
--R tubePointsDefault : () -> PositiveInteger
--R tubeRadiusDefault : Float -> DoubleFloat
--R tubeRadiusDefault : () -> DoubleFloat
--R unitsColorDefault : Palette -> Palette
--R var1StepsDefault : () -> PositiveInteger
--R var1StepsDefault : PositiveInteger -> PositiveInteger
--R var2StepsDefault : () -> PositiveInteger
--R var2StepsDefault : PositiveInteger -> PositiveInteger
--R viewPosDefault : () -> List(NonNegativeInteger)
--R viewPosDefault : List(NonNegativeInteger) -> List(NonNegativeInteger)
--R viewSizeDefault : () -> List(PositiveInteger)
--R viewSizeDefault : List(PositiveInteger) -> List(PositiveInteger)
--R viewWriteAvailable : () -> List(String)
--R viewWriteDefault : () -> List(String)
--R viewWriteDefault : List(String) -> List(String)
--R
--E 1

)spool
ViewDefaultsPackage (VIEWDEF)

Exports:

axesColorDefault  lineColorDefault  pointColorDefault
pointSizeDefault  tubePointsDefault  tubeRadiusDefault
unitsColorDefault  var1StepsDefault  var2StepsDefault
unitsColorDefault  viewDefaults    viewPosDefault
viewSizeDefault    viewWriteAvailable viewWriteDefault

— package VIEWDEF ViewDefaultsPackage —

)abbrev package VIEWDEF ViewDefaultsPackage
++ Author: Jim Wen
++ Date Created: 15 January 1990
++ Date Last Updated:
++ Description:
++ ViewportDefaultsPackage describes default and user definable
++ values for graphics

ViewportDefaultsPackage():Exports == Implementation where
I ==> Integer
C ==> Color
PAL ==> Palette
L ==> List
S ==> String
E ==> Expression
PI ==> PositiveInteger
NNI ==> NonNegativeInteger
SF ==> DoubleFloat
B ==> Boolean

writeAvailable ==> (["PIXMAP","BITMAP","POSTSCRIPT","IMAGE"]::L S)
-- need not worry about case of letters

Exports ==> with

  pointColorDefault : () -> PAL
  ++ pointColorDefault() returns the default color of points in a 2D
  ++ viewport.

  pointColorDefault : PAL -> PAL
  ++ pointColorDefault(p) sets the default color of points in a 2D viewport
  ++ to the palette p.

  lineColorDefault : () -> PAL
  ++ lineColorDefault() returns the default color of lines connecting
  ++ points in a 2D viewport.

  lineColorDefault : PAL -> PAL
  ++ lineColorDefault(p) sets the default color of lines connecting points
  ++ in a 2D viewport to the palette p.

  axesColorDefault : () -> PAL
  ++ axesColorDefault() returns the default color of the axes in a
  ++ 2D viewport.

  axesColorDefault : PAL -> PAL
  ++ axesColorDefault(p) sets the default color of the axes in a 2D
  ++ viewport to the palette p.

  unitsColorDefault : () -> PAL
  ++ unitsColorDefault() returns the default color of the unit ticks in
  ++ a 2D viewport.

  unitsColorDefault : PAL -> PAL
++ unitsColorDefault(p) sets the default color of the unit ticks in
++ a 2D viewport to the palette p.

pointSizeDefault : () -> PI
++ pointSizeDefault() returns the default size of the points in
++ a 2D viewport.

pointSizeDefault : PI -> PI
++ pointSizeDefault(i) sets the default size of the points in a 2D
++ viewport to i.

viewPosDefault : () -> L NNI
++ viewPosDefault() returns the default x and y position of a
++ viewport window unless overridden explicitly, newly created
++ viewports will have this x and y coordinate.

viewPosDefault : L NNI -> L NNI
++ viewPosDefault([x,y]) sets the default x and y position of a
++ viewport window unless overridden explicitly, newly created
++ viewports will have the x and y coordinates x, y.

viewSizeDefault : () -> L PI
++ viewSizeDefault() returns the default viewport width and height.

viewSizeDefault : L PI -> L PI
++ viewSizeDefault([w,h]) sets the default viewport width to w and height
++ to h.

viewDefaults : () -> Void
++ viewDefaults() resets all the default graphics settings.

viewWriteDefault : () -> L S
++ viewWriteDefault() returns the list of things to write in a viewport
++ data file; a viewalone file is always generated.

viewWriteDefault : L S -> L S
++ viewWriteDefault(l) sets the default list of things to write in a
++ viewport data file to the strings in l; a viewalone file is always
++ generated.

viewWriteAvailable : () -> L S
++ viewWriteAvailable() returns a list of available methods for writing,
++ such as BITMAP, POSTSCRIPT, etc.

var1StepsDefault : () -> PI
++ var1StepsDefault() is the current setting for the number of steps to
++ take when creating a 3D mesh in the direction of the first defined
++ free variable (a free variable is considered defined when its
++ range is specified (e.g. x=0..10)).
var2StepsDefault : () -> PI
++ var2StepsDefault() is the current setting for the number of steps to
++ take when creating a 3D mesh in the direction of the first defined
++ free variable (a free variable is considered defined when its
++ range is specified (e.g. x=0..10)).

var1StepsDefault : PI -> PI
++ var1StepsDefault(i) sets the number of steps to take when creating a
++ 3D mesh in the direction of the first defined free variable to i
++ (a free variable is considered defined when its range is specified
++ (e.g. x=0..10)).

var2StepsDefault : PI -> PI
++ var2StepsDefault(i) sets the number of steps to take when creating a
++ 3D mesh in the direction of the first defined free variable to i
++ (a free variable is considered defined when its range is specified
++ (e.g. x=0..10)).

tubePointsDefault : PI -> PI
++ tubePointsDefault(i) sets the number of points to use when creating
++ the circle to be used in creating a 3D tube plot to i.

tubePointsDefault : () -> PI
++ tubePointsDefault() returns the number of points to be used when
++ creating the circle to be used in creating a 3D tube plot.

tubeRadiusDefault : Float -> SF -- current tube.spad asks for SF
++ tubeRadiusDefault(r) sets the default radius for a 3D tube plot to r.

tubeRadiusDefault : () -> SF
++ tubeRadiusDefault() returns the radius used for a 3D tube plot.

Implementation ==> add

import Color()
import Palette()
--import StringManipulations()

defaultPointColor : Reference(PAL) := ref bright red()
defaultLineColor : Reference(PAL) := ref pastel green() --bright blue()
defaultAxesColor : Reference(PAL) := ref dim red()
defaultUnitsColor : Reference(PAL) := ref dim yellow()
defaultPointSize : Reference(PI) := ref(3::PI)
defaultXPos : Reference(NNI) := ref(0::NNI)
defaultYPos : Reference(NNI) := ref(0::NNI)
defaultWidth : Reference(PI) := ref(400::PI)
defaultHeight : Reference(PI) := ref(400::PI)
defaultThingsToWrite : Reference(L S) := ref([]::L S)
defaultVar1Steps : Reference(PI) := ref(27::PI)
defaultVar2Steps : Reference(PI) := ref(27::PI)
defaultTubePoints : Reference(PI) := ref(6::PI)
defaultTubeRadius : Reference(SF) := ref(convert(0.5)@SF)
defaultClosed : Reference(B) := ref(false)

--%Viewport window dimensions specifications
viewPosDefault == [defaultXPos(),defaultYPos()]
viewPosDefault l ==
  #1 < 2 => error "viewPosDefault expects a list with two elements"
  [defaultXPos() := first l,defaultYPos() := last l]
viewSizeDefault == [defaultWidth(),defaultHeight()]
viewSizeDefault l ==
  #1 < 2 => error "viewSizeDefault expects a list with two elements"
  [defaultWidth() := first l,defaultHeight() := last l]
viewDefaults ==
defaultPointColor : Reference(PAL) := ref bright red()
defaultLineColor : Reference(PAL) := ref pastel green() --bright blue()
defaultAxesColor : Reference(PAL) := ref dim red()
defaultUnitsColor : Reference(PAL) := ref dim yellow()
defaultPointSize : Reference(PI) := ref(3::PI)
defaultXPos : Reference(NNI) := ref(0::NNI)
defaultYPos : Reference(NNI) := ref(0::NNI)
defaultWidth : Reference(PI) := ref(400::PI)
defaultHeight : Reference(PI) := ref(427::PI)

--%2D graphical output specifications
pointColorDefault == defaultPointColor()
pointColorDefault p == defaultPointColor() := p

lineColorDefault == defaultLineColor()
lineColorDefault p == defaultLineColor() := p

axesColorDefault == defaultAxesColor()
axesColorDefault p == defaultAxesColor() := p

unitsColorDefault == defaultUnitsColor()
unitsColorDefault p == defaultUnitsColor() := p

pointSizeDefault == defaultPointSize()
pointSizeDefault x == defaultPointSize() := x

--%3D specific stuff
var1StepsDefault == defaultVar1Steps()
var1StepsDefault i == defaultVar1Steps() := i

var2StepsDefault == defaultVar2Steps()
var2StepsDefault i == defaultVar2Steps() := i
tubePointsDefault == defaultTubePoints()
tubePointsDefault i == defaultTubePoints() := i

tubeRadiusDefault == defaultTubeRadius()
tubeRadiusDefault f == defaultTubeRadius() := convert(f)@SF

--%File output stuff
viewWriteAvailable == writeAvailable

viewWriteDefault == defaultThingsToWrite()

viewWriteDefault listOfThings ==
  thingsToWrite : L S := []
  for aTypeOfFile in listOfThings repeat
    if (writeTypeInt := position(upperCase aTypeOfFile, viewWriteAvailable())) < 0 then
      sayBrightly([" > ", concat(aTypeOfFile, " is not a valid file type for writing a viewport")])$Lisp
    else
      thingsToWrite := append(thingsToWrite, [aTypeOfFile])
  defaultThingsToWrite() := thingsToWrite

— VIEWDEF.dotabb —

"VIEWDEF" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VIEWDEF"]
"STRING" [color="#88FF44",href="bookvol10.3.pdf#nameddest=STRING"]
"VIEWDEF" -> "STRING"

package VIEW ViewportPackage

— ViewportPackage.input —

)set break resume
)sys rm -f ViewportPackage.output
)spool ViewportPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show ViewportPackage
--R ViewportPackage is a package constructor
--R Abbreviation for ViewportPackage is VIEW
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for VIEW
--R
--R---------------------------------- Operations ----------------------------------
--R coerce : GraphImage -> TwoDimensionalViewport
--R drawCurves : (List(List(Point(DoubleFloat))),Palette,Palette,PositiveInteger,List(DrawOption)) -> TwoDimensionalViewport
--R drawCurves : (List(List(Point(DoubleFloat))),List(DrawOption)) -> TwoDimensionalViewport
--R graphCurves : (List(List(Point(DoubleFloat))),Palette,Palette,PositiveInteger,List(DrawOption)) -> GraphImage
--R graphCurves : List(List(Point(DoubleFloat))) -> GraphImage
--R graphCurves : (List(List(Point(DoubleFloat))),List(DrawOption)) -> GraphImage
--R
--E 1

)spool
)lisp (bye)

— ViewportPackage.help —

====================================================================
ViewportPackage examples
====================================================================

ViewportPackage provides functions for creating GraphImages
and TwoDimensionalViewports from lists of lists of points.

See Also:
o )show ViewportPackage


ViewportPackage (VIEW)

Exports:
coerce drawCurves graphCurves

— package VIEW ViewportPackage —

)abbrev package VIEW ViewportPackage
++ Author: Jim Wen
++ Date Created: 30 April 1989
++ Date Last Updated: 15 June 1990
++ Description:
++ ViewportPackage provides functions for creating GraphImages
++ and TwoDimensionalViewports from lists of lists of points.

ViewportPackage():Exports == Implementation where
DROP ==> DrawOption
GRIMAGE ==> GraphImage
L ==> List
P ==> Point DoubleFloat
PAL ==> Palette
PI ==> PositiveInteger
VIEW2D ==> TwoDimensionalViewport
Exports == with

  graphCurves : (L L P,PAL,PAL,PI,L DROP) -> GRIMAGE
  ++ graphCurves([p0],[p1],...,[pn],ptColor,lineColor,ptSize,[options])
  ++ creates a \spadtype{GraphImage} from the list of lists of points, p0
  ++ through pn, using the options specified in the list \spad{options}.
  ++ The graph point color is specified by \spad{ptColor}, the graph line
  ++ color is specified by \spad{lineColor}, and the size of the points is
  ++ specified by \spad{ptSize}.

  graphCurves : (L L P,L DROP) -> GRIMAGE
  ++ graphCurves([p0],[p1],...,[pn]) creates a \spadtype{GraphImage} from
  ++ the list of lists of points indicated by p0 through pn.

  graphCurves : (L L P,L DROP) -> GRIMAGE
++ graphCurves([[p0],[p1],...,[pn]], [options]) creates a
++ \spadtype{GraphImage} from the list of lists of points, p0 through pn,
++ using the options specified in the list \spad{options}.

drawCurves : (L L P, PAL, PAL, PI, L DROP) -> VIEW2D
++ drawCurves([[p0],[p1],...,[pn]], ptColor, lineColor, ptSize, [options])
++ creates a \spadtype{TwoDimensionalViewport} from the list of lists of
++ points, p0 through pn, using the options specified in the list
++ \spad{options}. The point color is specified by \spad{ptColor}, the
++ line color is specified by \spad{lineColor}, and the point size is
++ specified by \spad{ptSize}.

drawCurves : (L L P, L DROP) -> VIEW2D
++ drawCurves([[p0],[p1],...,[pn]], [options]) creates a
++ \spadtype{TwoDimensionalViewport} from the list of lists of points,
++ p0 through pn, using the options specified in the list \spad{options};

coerce : GRIMAGE -> VIEW2D
++ coerce(gi) converts the indicated \spadtype{GraphImage}, gi, into the
++ \spadtype{TwoDimensionalViewport} form.

Implementation ==> add

import ViewDefaultsPackage
import DrawOptionFunctions0

--% Functions that return GraphImages

graphCurves(listOfListsOfPoints) ==
  graphCurves(listOfListsOfPoints, pointColorDefault(), _
               lineColorDefault(), pointSizeDefault(), nil())

graphCurves(listOfListsOfPoints, optionsList) ==
  graphCurves(listOfListsOfPoints, pointColorDefault(), _
               lineColorDefault(), pointSizeDefault(), optionsList)

graphCurves(listOfListsOfPoints, ptColor, lineColor, ptSize, optionsList) ==
  len := #listOfListsOfPoints
  listOfPointColors : L PAL := [ptColor for i in 1..len]
  listOfLineColors : L PAL := [lineColor for i in 1..len]
  listOfPointSizes : L PI := [ptSize for i in 1..len]
  makeGraphImage(listOfListsOfPoints, listOfPointColors, _
                  listOfLineColors, listOfPointSizes, optionsList)

--% Functions that return Two Dimensional Viewports

drawCurves(listOfListsOfPoints, optionsList) ==
  drawCurves(listOfListsOfPoints, pointColorDefault(), _
             lineColorDefault(), pointSizeDefault(), optionsList)

  v := viewport2D()
  options(v, optList)
\begin{verbatim}
g := graphCurves(ptLists,ptColor,lColor,ptSize,optList)
putGraph(v,g,1)
makeViewport2D v

--% Coercions

coerce(graf:GRIMAGE):VIEW2D ==
  if (key graf = 0) then makeGraphImage graf
  v := viewport2D()
  title(v,"VIEW2D")
  -- dimensions(v,viewPosDefault().1,viewPosDefault().2,viewSizeDefault().1,viewSizeDefault())
  putGraph(v,graf,1::PI)
  makeViewport2D v

---

— VIEW.dotabb —

"VIEW" [color="#FF4488",href="bookvol10.4.pdf#nameddest=VIEW"]
"FIELD" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FIELD"]
"RADCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=RADCAT"]
"FLAGG" [color="#4488FF",href="bookvol10.2.pdf#nameddest=FLAGG"]
"VIEW" -> "FIELD"
"VIEW" -> "RADCAT"
"VIEW" -> "FLAGG"

---
\end{verbatim}
package WEIER WeierstrassPreparation

--- WeierstrassPreparation.input ---

)set break resume
)sys rm -f WeierstrassPreparation.output
)spool WeierstrassPreparation.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show WeierstrassPreparation
--R
--R WeierstrassPreparation(R: Field) is a package constructor
--R Abbreviation for WeierstrassPreparation is WEIER
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for WEIER
--R
--R----------------------------------- Operations -----------------------------------
--R cfirst : NonNegativeInteger -> (Stream(Polynomial(R)) -> Stream(Polynomial(R)))
--R clikeUniv : Symbol -> (Polynomial(R) -> SparseUnivariatePolynomial(Polynomial(R)))
--R crest : NonNegativeInteger -> (Stream(Polynomial(R)) -> Stream(Polynomial(R)))
--R qqq : (NonNegativeInteger,TaylorSeries(R),Stream(TaylorSeries(R))) -> (Stream(TaylorSeries(R)) -> Stream(TaylorSeries(R)))
--R sts2stst : (Symbol,Stream(Polynomial(R))) -> Stream(Stream(Polynomial(R)))
--R weierstrass : (Symbol,TaylorSeries(R)) -> List(TaylorSeries(R))
--R
--E 1

)spool
)lisp (bye)
This package implements the Weierstrass preparation theorem for multivariate power series.

\texttt{weierstrass(v,p)} where \( v \) is a variable, and \( p \) is a \texttt{TaylorSeries(R)} in which the terms of lowest degree must include \( c*v^s \) where \( c \) is a constant, \( s>0 \), is a list of \texttt{TaylorSeries} coefficients \( A[i] \) of the equivalent polynomial

\[ A = A[0] + A[1]*v + A[2]*v^2 + \ldots + A[s-1]*v^{s-1} + v^s \]

such that \( p=A*B \), \( B \) being a \texttt{TaylorSeries} of minimum degree 0

See Also:
\texttt{cshow WeierstrassPreparation}

---

\textbf{WeierstrassPreparation (WEIER)}

\begin{center}
\includegraphics[width=0.2\textwidth]{weierstrass.png}
\end{center}

Exports:
\texttt{cfirst clikcUniv crest qqqq}
\texttt{sts2stst weierstrass}

---

\texttt{package WEIER WeierstrassPreparation}

\texttt{)abbrev package WEIER WeierstrassPreparation}
++ Author: William H. Burge
++ Date Created: Sept 1988
++ Date Last Updated: Feb 15 1992
++ Description:
++ This package implements the Weierstrass preparation theorem for multivariate power series.
++ \texttt{weierstrass(v,p)} where \(v\) is a variable, and \(p\) is a \texttt{TaylorSeries(R)} in which the terms
++ of lowest degree \(s\) must include \(c\cdot v^s\) where \(c\) is a constant, \(s > 0\),
++ is a list of \texttt{TaylorSeries} coefficients \(A[i]\) of the equivalent polynomial
++ \(A = A[0] + A[1]\cdot v + A[2]\cdot v^2 + \ldots + A[s-1]\cdot v^{s-1} + v^s\)
++ such that \(p = A \cdot B\), \(B\) being a \texttt{TaylorSeries} of minimum degree 0

\texttt{WeierstrassPreparation(R): Defn == Impl where}
\begin{verbatim}
R : Field
VarSet==>Symbol
SMP ==> Polynomial R
PS ==> InnerTaylorSeries SMP
NNI ==> NonNegativeInteger
ST ==> Stream
StS ==> Stream SMP
STPS==>StreamTaylorSeriesOperations
STTAYLOR==>StreamTaylorSeriesOperations
SUP ==> SparseUnivariatePolynomial(SMP)
ST2==>StreamFunctions2
SMPS==> TaylorSeries(R)
L==>List
null ==> empty?
likeUniv ==> univariate
coef ==> coefficient$SUP
nil ==> empty
\end{verbatim}

\texttt{Defn == with}
\begin{verbatim}
crest:(NNI->( StS-> StS))
  ++ \spad{crest n} is used internally.
cfirst:(NNI->( StS-> StS))
  ++ \spad{cfirst n} is used internally.
sts2stst:(VarSet,StS)->ST StS
  ++ \spad{sts2stst(v,s)} is used internally.
clikeUniv:VarSet->(SMP->SUP)
  ++ \spad{clikeUniv(v)} is used internally.
weierstrass:(VarSet,SMPS)->L SMPS
  ++ \spad{weierstrass(v,ts)} where \(v\) is a variable and \(ts\) is
  ++ a \texttt{TaylorSeries}, implements the Weierstrass Preparation
  ++ Theorem. The result is a list of \texttt{TaylorSeries} that
  ++ are the coefficients of the equivalent series.
qqq:(NNI,SMPS,ST SMPS)->((ST SMPS)->ST SMPS)
  ++ \spad{qqq(n,s,st)} is used internally.
\end{verbatim}
Impl ==> add
import TaylorSeries(R)
import StreamTaylorSeriesOperations SMP
import StreamTaylorSeriesOperations SMPS

map1==>map$(ST2(SMP,SUP))
map2==>map$(ST2(StS,SMP))
map3==>map$(ST2(StS,StS))
transback:ST SMPS→L SMPS
transback smps==
  if null smps
  then nil()$(L SMPS)
  else
    if null first (smps:(ST StS))
    then nil()$(L SMPS)
    else
      cons(map2(first,smps:ST StS):SMPS,
           transback(map3(rest,smps:ST StS):(ST SMPS))$(L SMPS))

clikeUniv(var)==p +-> likeUniv(p,var)
mind:(NNI,StS)→NNI
mind(n, sts)==
  if null sts
  then error "no mindegree"
  else if first sts=0
    then mind(n+1,rest sts)
  else n
mindegree (sts:StS):NNI== mind(0,sts)

streamlikeUniv:(SUP,NNI)→StS
streamlikeUniv(p:SUP,n:NNI): StS ==
  if n=0
    then cons(coef (p,0),nil()$StS)
  else cons(coef (p,n),streamlikeUniv(p,(n-1):NNI))
transpose:ST StS→ST StS
transpose(s:ST StS)==delay(
  if null s
  then nil()$(ST StS)
  else cons(map2(first,s),transpose(map3(rest,rst s))))

zp==>map$StreamFunctions3(SUP,NNI,StS)
sts2stst(var, sts)==
  zp((x,y) +-> streamlikeUniv(x,y),
      map1(clikeUniv var, sts),(integers 0):(ST NNI))
tp: (VarSet, StS) -> ST StS
tp(v, sts) == transpose sts 2stst(v, sts)
map4 == map$(ST2 (StS, StS))
maptake: (NNI, ST StS) -> ST SMPS
maptake(n, p) == map4(cfirst n, p) pretend ST SMPS
mapdrop: (NNI, ST StS) -> ST SMPS
mapdrop(n, p) == map4(crest n, p) pretend ST SMPS
YSS == Y$ParadoxicalCombinatorsForStreams(SMPS)
weier: (VarSet, StS) -> ST SMPS
weier(v, sts) ==
  a := mindegree sts
  if a = 0
    then error "has constant term"
  else
    p := tp(v, sts) pretend (ST SMPS)
    b := rest(((first p pretend StS)), a::NNI)
    c := retractIfCan first b
c case "failed" =>
  error "the coefficient of the lowest degree of the variable should be a constant"
  e := recip b
  f := if e case "failed"
   then error "no reciprocal"
   else e::StS
  q := (YSS qqq(a, f: SMPS, rest p))
  maptake(a, (p*q) pretend ST StS)
cfirst n == s +-> first(s, n)$StS
crest n == s +-> rest(s, n)$StS
qq: (NNI, SMPS, ST SMPS, ST SMPS) -> ST SMPS
qq(a, e, p, c) ==
  cons(e, (-e)*mapdrop(a, (p*c) pretend (ST StS)))
qqq(a, e, p) == s +-> qq(a, e, p, s)
wei: (VarSet, SMPS) -> ST SMPS
wei(v: VarSet, s: SMPS) == weier(v, s: StS)
weierstrass(v, smps) == transback wei(v, smps)

"WEIER" [color="#FF4488", href="bookvol10.4.pdf#nameddest=WEIER"]
"PFECAT" [color="#4488FF", href="bookvol10.2.pdf#nameddest=PFECAT"]
"WEIER" -> "PFECAT"
package WFFINTBS WildFunctionFieldIntegralBasis

— WildFunctionFieldIntegralBasis.input —

)set break resume
)sys rm -f WildFunctionFieldIntegralBasis.output
)spool WildFunctionFieldIntegralBasis.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show WildFunctionFieldIntegralBasis

--R WildFunctionFieldIntegralBasis(K: FiniteFieldCategory,R: UnivariatePolynomialCategory(K),UP: UnivariatePolynomialCategory(R),F: FramedAlgebra(R,UP)) is a package constructor

--R Abbreviation for WildFunctionFieldIntegralBasis is WFFINTBS

--R This constructor is not exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for WFFINTBS

--R

--R----------------------------------- Operations --------------------------------

--R integralBasis : () -> Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R))

--R localIntegralBasis : R -> Record(basis: Matrix(R),basisDen: R,basisInv: Matrix(R))

--R

--E 1

)spool
)lisp (bye)

— WildFunctionFieldIntegralBasis.help —

====================================================================
WildFunctionFieldIntegralBasis examples
====================================================================

In this package K is a finite field, R is a ring of univariate polynomials over K, and F is a framed algebra over R. The package provides a function to compute the integral closure of R in the quotient field of F as well as a function to compute a "local integral basis" at a specific prime.

See Also:
o )show WildFunctionFieldIntegralBasis
WildFunctionFieldIntegralBasis (WFFINTBS)

Exports:
integralBasis  localIntegralBasis

— package WFFINTBS WildFunctionFieldIntegralBasis —

\begin{verbatim}
)abbrev package WFFINTBS WildFunctionFieldIntegralBasis
++ Authors: Victor Miller, Clifton Williamson
++ Date Created: 24 July 1991
++ Date Last Updated: 20 September 1994
++ Description:
++ In this package \textit{K} is a finite field, \textit{R} is a ring of univariate
++ polynomials over \textit{K}, and \textit{F} is a framed algebra over \textit{R}. The package
++ provides a function to compute the integral closure of \textit{R} in the quotient
++ field of \textit{F} as well as a function to compute a “local integral basis”
++ at a specific prime.

\textbf{WildFunctionFieldIntegralBasis}(\textit{K, R, UP, F}): Exports == Implementation where
\textit{K} : FiniteFieldCategory
--\textit{K} : Join(Field,Finite)
\textit{R} : UnivariatePolynomialCategory \textit{K}
\textit{UP} : UnivariatePolynomialCategory \textit{R}
\textit{F} : FramedAlgebra(\textit{R, UP})

I  ==> Integer
Mat ==> Matrix \textit{R}
NNI ==> NonNegativeInteger
SAE ==> SimpleAlgebraicExtension
RResult ==> Record(basis: Mat, basisDen: \textit{R}, basisInv:Mat)
IResult ==> Record(basis: Mat, basisDen: \textit{R}, basisInv:Mat,discr: \textit{R})
MATSTOR ==> StorageEfficientMatrixOperations

Exports ==> with
  integralBasis : () -> RResult
++ the integral closure of \( R \) in the quotient field of \( F \), where
++ \( F \) is a framed algebra with \( R \)-module basis \( \{w_1, w_2, \ldots, w_n\} \).
++ If \( \text{basis} \) is the matrix \( \{ (a_{ij}, i = 1..n, j = 1..n) \} \), then
++ the \( i \)th element of the integral basis is
++ \( \text{basis}(v_i) = \left( \frac{1}{\text{basisDen}} \right) \sum a_{ij} \cdot w_j \), i.e., the
++ \( i \)th row of \( \text{basis} \) contains the coordinates of the
++ \( i \)th basis vector. Similarly, the \( i \)th row of the
++ matrix \( \text{basisInv} \) contains the coordinates of \( \text{basis}(w_i) \) with
++ respect to the basis \( \{v_1, \ldots, v_n\} \) if \( \text{basisInv} \) is the
++ matrix \( \left( (b_{ij}, i = 1..n, j = 1..n) \right) \), then
++ \( \text{basis}(w_i) = \sum b_{ij} \cdot v_j \), i.e., the
++ \( i \)th row of \( \text{basisInv} \) contains the coordinates of \( \text{basis}(w_i) \) with
++ respect to the basis \( \{v_1, \ldots, v_n\} \) if \( \text{basisInv} \) is the
++ matrix \( \left( (b_{ij}, i = 1..n, j = 1..n) \right) \), then
++ \( \text{basis}(w_i) = \sum b_{ij} \cdot v_j \).

**localIntegralBasis :** \( R \to \text{RResult} \)
++ \( \text{integralBasis}(p) \) returns a record
++ \( \{ \text{basis}, \text{basisDen}, \text{basisInv} \} \) containing information regarding
++ the local integral closure of \( R \) at the prime \( p \) in the quotient
++ field of \( F \), where \( F \) is a framed algebra with \( R \)-module basis
++ \( \{w_1, w_2, \ldots, w_n\} \).
++ If \( \text{basis} \) is the matrix \( \{ (a_{ij}, i = 1..n, j = 1..n) \} \), then
++ the \( i \)th element of the local integral basis is
++ \( \text{basis}(v_i) = \left( \frac{1}{\text{basisDen}} \right) \sum a_{ij} \cdot w_j \), i.e., the
++ \( i \)th row of \( \text{basis} \) contains the coordinates of the
++ \( i \)th basis vector. Similarly, the \( i \)th row of the
++ matrix \( \text{basisInv} \) contains the coordinates of \( \text{basis}(w_i) \) with
++ respect to the basis \( \{v_1, \ldots, v_n\} \) if \( \text{basisInv} \) is the
++ matrix \( \left( (b_{ij}, i = 1..n, j = 1..n) \right) \), then
++ \( \text{basis}(w_i) = \sum b_{ij} \cdot v_j \).

Implementation \( \Rightarrow \) add
import IntegralBasisTools\( (R, \text{UP}, F) \)
import ModularHermitianRowReduction\( (R) \)
import TriangularMatrixOperations\( (R, \text{Vector} R, \text{Vector} R, \text{Matrix} R) \)
import DistinctDegreeFactorize\( (K,R) \)

**listSquaredFactors :** \( R \to \text{List} R \)
**listSquaredFactors** \( px \) ==
-- returns a list of the factors of \( px \) which occur with
-- exponent > 1
ans : List R := empty()
factored := factor\( (px) \)$\text{DistinctDegreeFactorize}(K,R)
for \( f \) in factors\( (\text{factored}) \) repeat
if \( f \cdot \text{exponent} > 1 \) then ans := concat\( (f \cdot \text{factor}, \text{ans}) \)
ans

**iLocalIntegralBasis :** \( (\text{Vector} F,\text{Vector} F,\text{Matrix} R,\text{Matrix} R, R, R) \to \text{IResult} \)
iLocalIntegralBasis\( (\text{bas}, \text{pows}, \text{tfm}, \text{matrixOut}, \text{disc}, \text{prime}) \) ==
n := rank\( () \)$F; \text{standardBasis} := basis\( () \)$F
-- 'standardBasis' is the basis for \( F \) as a FramedAlgebra;
-- usually this is \( \{1, y, y^2, \ldots, y^{(n-1)}\} \)
p2 := prime \* prime; \text{sae} := SAE\( (K,R,\text{prime}) \)
p := characteristic\( () \)$F; \text{q} := size\( () \)$\text{sae}
lp := leastPower(q,n)
rb := scalarMatrix(n,1); rbinv := scalarMatrix(n,1)
-- rb = basis matrix of current order
-- rbinv = inverse basis matrix of current order
-- these are wrt the orginal basis for F
rbden := 1; index := 1; oldIndex := 1
-- rbeden = denominator for current basis matrix
-- index = index of original order in current order
repeat
  -- pows = [(w1 * rbeden) ** q,...,(wn * rbeden) ** q], where
  -- bas = [w1,...,wn] is 'rbden' times the basis for the order B = 'rb'
  for i in 1..n repeat
    bi := 0
    for j in 1..n repeat
      bi := bi + qelt(rb,i,j) * qelt(standardBasis,j)
    end for
    qsetelt!(bas,i,bi)
    qsetelt!(pows,i,bi ** p)
  end for
  coor0 := transpose coordinates(pows,bas)
  denPow := rbeden ** ((p - 1) :: NNI)
  (coMat0 := coor0 exquo denPow) case "failed" =>
    error "can't happen"
  -- the jth column of coMat contains the coordinates of (wj/rbden)**q
  -- with respect to the basis [w1/rbden,...,wn/rbden]
  coMat := coMat0 :: Matrix R
  -- the ith column of 'pPows' contains the coordinates of the pth power
  -- of the ith basis element for B/prime.B over 'sae' = R/prime.R
  pPows := map(reduce,coMat)$MatrixCategoryFunctions2(R,Vector R,
                                                Vector R,Matrix R,
                                                Vector R,Vector R,Matrix R)
  -- 'frob' will eventually be the Frobenius matrix for B/prime.B over
  -- 'sae' = R/prime.R; at each stage of the loop the ith column will
  -- contain the coordinates of p^r-th powers of the ith basis element
  for r in 2..leastPower(p,q) repeat
    for i in 1..n repeat for j in 1..n repeat
      qsetelt!(tmpMat,i,j,qelt(frob,i,j) ** p)
    end for
  end for
  frobPow := frob ** lp
-- compute the p-radical
  ns := nullSpace frobPow
  for i in 1..n repeat for j in 1..n repeat qsetelt!(tfm,i,j,0)
  for vec in ns for i in 1.. repeat
    for j in 1..n repeat
      qsetelt!(tfm,i,j,lift qelt(vec,j))
    end for
  end for
  id := squareTop rowEchelon(tfm,prime)
-- id = basis matrix of the p-radical
  idinv := UpTriBddDenomInv(id, prime)
-- id * idinv = prime * identity
-- no need to check for inseparability in this case
  rbinv := idealiser(id * rb, rbinv * idinv, prime * rbeden)
  index := diagonalProduct rbinv

rb := rowEchelon LowTriBddDenomInv(rbinv,rbden * prime)
if divideIfCan_!(rb,matrixOut,prime,n) = 1
   then rb := matrixOut
   else rbden := rbden * prime
rbinv := UpTriBddDenomInv(rb,rbden)
indexChange := index quo oldIndex
oldIndex := index
disc := disc quo (indexChange * indexChange)
(not sizeLess?(1,indexChange)) or ((disc exquo p2) case "failed") =>
   return [rb, rbden, rbinv, disc]

integralBasis() ==
traceMat := traceMatrix()$F; n := rank()$F
disc := determinant traceMat  -- discriminant of current order
zero? disc => error "integralBasis: polynomial must be separable"
singList := listSquaredFactors disc  -- singularities of relative Spec
runningRb := scalarMatrix(n,1); runningRbinv := scalarMatrix(n,1)
   -- runningRb = basis matrix of current order
   -- runningRbinv = inverse basis matrix of current order
-- these are wrt the original basis for F
runningRbden : R := 1
   -- runningRbden = denominator for current basis matrix
empty? singList => [runningRb, runningRbden, runningRbinv]
bas : Vector F := new(n,0); pows : Vector F := new(n,0)
   -- storage for basis elements and their powers
tfm : Matrix R := new(n,n,0)
   -- 'tfm' will contain the coordinates of a lifting of the kernel
-- of a power of Frobenius
matrixOut : Matrix R := new(n,n,0)
for prime in singList repeat
   lb := iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime)
   rb := lb.basis; rbinv := lb.basisInv; rbden := lb.basisDen
   disc := lb.dscr
   -- update 'running integral basis' if newly computed
-- local integral basis is non-trivial
if sizeLess?(1,rbden) then
   mat := vertConcat(rbden * runningRb,runningRbden * rb)
   runningRbden := runningRbden * rbden
   runningRb := squareTop rowEchelon(mat,runningRbden)
   runningRbinv := UpTriBddDenomInv(runningRb,runningRbden)
   [runningRb, runningRbden, runningRbinv]

localIntegralBasis prime ==
traceMat := traceMatrix()$F; n := rank()$F
disc := determinant traceMat  -- discriminant of current order
zero? disc => error "localIntegralBasis: polynomial must be separable"
(disc exquo (prime * prime)) case "failed" =>
   [scalarMatrix(n,1), 1, scalarMatrix(n,1)]
bas : Vector F := new(n,0); pows : Vector F := new(n,0)
   -- storage for basis elements and their powers
tfm : Matrix R := new(n,n,0)  
-- 'tfm' will contain the coordinates of a lifting of the kernel  
-- of a power of Frobenius  
matrixOut : Matrix R := new(n,n,0)  
lb := iLocalIntegralBasis(bas,pows,tfm,matrixOut,disc,prime)  
[lb.basis, lb.basisDen, lb.basisInv]

— WFFINTBS.dotabb —

"WFFINTBS" [color="#FF4488",href="bookvol10.4.pdf#nameddest=WFFINTBS"]  
"MONOGEN" [color="#4488FF",href="bookvol10.2.pdf#nameddest=MONOGEN"]  
"WFFINTBS" -> "MONOGEN"
package XEXPPKG XExponentialPackage

— XExponentialPackage.input —

)set break resume
/sys rm -f XExponentialPackage.output
/spool XExponentialPackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 1
)show XExponentialPackage

--R
--R XExponentialPackage(R: Join(Ring,Module(Fraction(Integer))),VarSet: OrderedSet,XPOLY: XPolynomialsCat(VarSet,R)) is a package constructor
--R Abbreviation for XExponentialPackage is XEXPPKG
--R This constructor is not exposed in this frame.
--R Issue )edit bookvol10.4.pamphlet to see algebra source code for XEXPPKG

--R------------------------------- Operations --------------------------------
--R Hausdorff : (XPOLY,XPOLY,NonNegativeInteger) -> XPOLY
--R exp : (XPOLY,NonNegativeInteger) -> XPOLY
--R log : (XPOLY,NonNegativeInteger) -> XPOLY

--E 1

)spool
)lisp (bye)

___
This package provides computations of logarithms and exponentials for polynomials in non-commutative variables.

See Also:
c )show XExponentialPackage

---

XExponentialPackage (XEXPPKG)

Exports:
exp  Hausdorff  log

---

"XEXPPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=XEXPPKG"]

---

)abbrev package XEXPPKG XExponentialPackage
++ Author: Michel Petitot (petitot@lifl.fr).
++ Date Created: 91
-- Date Last Updated: 7 Juillet 92
-- Description:
-- This package provides computations of logarithms and exponentials
-- for polynomials in non-commutative variables.

XExponentialPackage(R, VarSet, XPOLY): Public == Private where
RN ==> Fraction Integer
NNI ==> NonNegativeInteger
I ==> Integer
R : Join(Ring, Module RN)
-- R : Field
VarSet : OrderedSet
XPOLY : XPolynomialsCat(VarSet, R)

Public == with
exp: (XPOLY, NNI) -> XPOLY
++ \texttt{exp(p, n)} returns the exponential of \texttt{p}
++ truncated at order \texttt{n}.
log: (XPOLY, NNI) -> XPOLY
++ \texttt{log(p, n)} returns the logarithm of \texttt{p}
++ truncated at order \texttt{n}.
Hausdorff: (XPOLY, XPOLY, NNI) -> XPOLY
++ \texttt{Hausdorff(a,b,n)} returns log(exp(a)*exp(b))
++ truncated at order \texttt{n}.

Private == add

log (p,n) ==
p1 : XPOLY := p - 1
not quasiRegular? p1 =>
    error "constant term <> 1, impossible log"
s : XPOLY := 0 -- resultat
k : I := n :: I
for i in 1 .. n repeat
    k1 :RN := 1/k
    k2 : R := k1 * 1$R
    s := trunc( trunc(p1,i) * (k2 :: XPOLY - s) , i)
k := k - 1
s

exp (p,n) ==
not quasiRegular? p =>
    error "constant term <> 0, exp impossible"
p = 0 => 1
s : XPOLY := 1$XPOLY -- resultat
k : I := n :: I
for i in 1 .. n repeat
    k1 :RN := 1/k
    k2 : R := k1 * 1$R
    s := trunc( 1 +$XPOLY k2 * trunc(p,i) * s , i)
\[
k := k - 1
\]

Hausdorff(p, q, n) ==
\[
p1: XPOLY := \exp(p, n)
q1: XPOLY := \exp(q, n)
\]
log(p1*q1, n)

— XEXPPKG.dotabb —

"XEXPPKG" [color="#FF4488",href="bookvol10.4.pdf#nameddest=XEXPPKG"]
"XPOLYC" [color="#4488FF",href="bookvol10.2.pdf#nameddest=XPOLYC"]
"XEXPPKG" -> "XPOLYC"
Chapter 26

Chapter Y
package ZDSOLVE ZeroDimensionalSolvePackage

Based on triangular decompositions and the RealClosure constructor, the package ZeroDimensionalSolvePackage provides operations for computing symbolically the real or complex roots of polynomial systems with finitely many solutions.

— ZeroDimensionalSolvePackage.input —

)set break resume
)spool ZeroDimensionalSolvePackage.output
)set message test on
)set message auto off
)clear all

--S 1 of 29
R := Integer
--R
--R
--R (1) Integer
--R
--E 1

--S 2 of 29
ls : List Symbol := [x,y,z,t]
--R
--R
--R (2) [x,y,z,t]
--R
--E 2

--S 3 of 29
ls2 : List Symbol := [x,y,z,t,new()$Symbol]

4189
pack := ZDSOLVE(R,ls,ls2)

lp := [p1, p2, p3]
triangSolve(lp)$pack

\[(9) \]
\[
\left\{ \\
\begin{array}{ccccccccc}
20 & 19 & 18 & 17 & 16 & 15 & 14 & 13 & 12 \\
& z & -6z & -41z & +71z & +106z & +92z & +197z & +145z & +257z \\
\end{array}
\right.
\]
\[
\begin{array}{ccccccccc}
278z & +201z & +278z & +257z & +145z & +197z & +92z & +106z & +71z \\
& + & 2 & -41z & -6z & +1 \\
\end{array}
\]
\[
\begin{array}{cccccccc}
19 & 18 & 17 & 16 & 15 \\
14745844z & +50357474z & -130948857z & -185261586z & \ + \\
15 & 14 & 13 & 12 \\
-18007775z & -338007307z & -275379623z & -453190404z & + \\
11 & 10 & 9 & 8 \\
474597456z & -366147695z & -481433567z & -430613166z & + \\
7 & 6 & 5 & 4 \\
261878358z & -326073537z & -163008796z & -177213227z & + \\
3 & 2 \\
104356756z & +65241699z & +9237732z & -1567348 & * \\
\end{array}
\]
\[
\begin{array}{cccccccc}
19 & 18 & 17 & 16 & 15 \\
1917314z & +6508991z & -16973165z & -24000259z & -23349192z & + \\
14 & 13 & 12 & 11 & 10 \\
43786426z & -35696474z & -58724172z & -61480792z & -47452440z & + \\
9 & 8 & 7 & 6 & 5 \\
62378085z & -55776527z & -33940618z & -42233406z & -21122875z & + \\
4 & 3 & 2 \\
22958177z & -13504569z & +8448317z & +1195888z & -202934 & 
\end{array}
\]
univariateSolve(lp)$pack

complexRoots = [12 11 10 9 8 7 6 5 4 3

+ 2
24? - 12? + 1

coordinates = [11 10 9 8 7 6

63x + 62%A - 721%A + 1220%A + 705%A - 285%A + 1512%A
+ 5 4 3 2

- 735%A + 1401%A - 21%A + 215%A + 1577%A - 142
,

63y - 75%A + 890%A - 1682%A - 516%A + 588%A - 1953%A
+ 5 4 3 2

1323%A - 1815%A + 426%A - 243%A - 1801%A + 679
, z - %A]

]
lr := realSolve(lp)$pack

\[
\begin{align*}
\text{(11)} \quad &\left[ \frac{1184459}{1645371} - \frac{1233702}{832819} - \frac{5460230}{1645371} + \frac{79900378}{1645371} \\
&\frac{43953929}{1645371} + \frac{13420192}{548457} + \frac{553986}{182819} + \frac{193381378}{1645371} \\
&\frac{548457}{1645371} + \frac{3731}{1645371} + \frac{548457}{182819} + \frac{182819}{1645371} \\
&\frac{35978916}{1645371} + \frac{358660781}{1645371} + \frac{9}{118784873} + \frac{8}{1645371} \\
&\frac{182819}{1645371} + \frac{11193}{4459} + \frac{4459}{42189} \\
&\frac{337505020}{1645371} + \frac{1389370}{548457} + \frac{688291}{182819} + \frac{538002}{1645371} \\
&\frac{1645371}{1645371} + \frac{11193}{4459} + \frac{4459}{42189} \\
&\frac{14076176}{1645371} + \frac{32325724}{1645371} + \frac{8270}{548457} + \frac{9741532}{1645371} \\
&\frac{1645371}{1645371} + \frac{11193}{4459} + \frac{4459}{42189} \\
&\frac{91729}{1645371} + \frac{487915}{548457} + \frac{4114333}{182819} + \frac{1276987}{1645371} \\
&\frac{705159}{705159} + \frac{705159}{705159} + \frac{705159}{235053} \\
&\frac{13243117}{705159} + \frac{16292173}{705159} + \frac{26563060}{705159} + \frac{722714}{1645371} \\
&\frac{705159}{705159} + \frac{705159}{705159} + \frac{705159}{235053} \\
&\frac{5382578}{705159} + \frac{15499995}{705159} + \frac{14279770}{705159} + \frac{6603890}{705159} \\
&\frac{100737}{235053} + \frac{235053}{235053} + \frac{100737}{705159} \\
&\frac{409930}{705159} + \frac{37340389}{705159} + \frac{34893175}{705159} + \frac{26686318}{705159} \\
&\frac{6027}{705159} + \frac{705159}{705159} + \frac{705159}{705159} \\
&\frac{801511}{705159} + \frac{17206178}{705159} + \frac{4406102}{377534} + \frac{377534}{705159} \\
&\frac{26117}{705159} + \frac{705159}{705159} + \frac{705159}{705159} \\
&\]
\end{align*}
\]
<table>
<thead>
<tr>
<th>%B2</th>
<th>1184459</th>
<th>2335702</th>
<th>5460230</th>
<th>79900378</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>%B2</td>
<td>1645371</td>
<td>548457</td>
<td>182819</td>
<td>1645371</td>
<td>14</td>
</tr>
<tr>
<td>%B2</td>
<td>43953929</td>
<td>15</td>
<td>13420192</td>
<td>14</td>
<td>563986</td>
</tr>
<tr>
<td>%B2</td>
<td>548457</td>
<td>182819</td>
<td>3731</td>
<td>1645371</td>
<td>10</td>
</tr>
<tr>
<td>%B2</td>
<td>35978916</td>
<td>11</td>
<td>358660781</td>
<td>10</td>
<td>271667666</td>
</tr>
<tr>
<td>%B2</td>
<td>182819</td>
<td>1645371</td>
<td>1645371</td>
<td>548457</td>
<td>8</td>
</tr>
<tr>
<td>%B2</td>
<td>337505020</td>
<td>7</td>
<td>1389370</td>
<td>6</td>
<td>688291</td>
</tr>
<tr>
<td>%B2</td>
<td>1645371</td>
<td>11193</td>
<td>4459</td>
<td>42189</td>
<td>6</td>
</tr>
<tr>
<td>%B2</td>
<td>140671876</td>
<td>3</td>
<td>32325724</td>
<td>2</td>
<td>271667666</td>
</tr>
<tr>
<td>%B2</td>
<td>1645371</td>
<td>548457</td>
<td>343</td>
<td>1645371</td>
<td>4</td>
</tr>
<tr>
<td>%B2</td>
<td>91729</td>
<td>487915</td>
<td>18</td>
<td>4114333</td>
<td>17</td>
</tr>
<tr>
<td>%B2</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>235053</td>
<td>10</td>
</tr>
<tr>
<td>%B2</td>
<td>13243117</td>
<td>15</td>
<td>16292173</td>
<td>14</td>
<td>26536060</td>
</tr>
<tr>
<td>%B2</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>18081</td>
<td>8</td>
</tr>
<tr>
<td>%B2</td>
<td>5382578</td>
<td>11</td>
<td>15499995</td>
<td>10</td>
<td>14279770</td>
</tr>
<tr>
<td>%B2</td>
<td>100737</td>
<td>235053</td>
<td>235053</td>
<td>100737</td>
<td>4</td>
</tr>
<tr>
<td>%B2</td>
<td>409930</td>
<td>7</td>
<td>37340389</td>
<td>6</td>
<td>34893715</td>
</tr>
<tr>
<td>%B2</td>
<td>6027</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>8</td>
</tr>
<tr>
<td>%B2</td>
<td>801511</td>
<td>3</td>
<td>17206178</td>
<td>2</td>
<td>4406102</td>
</tr>
<tr>
<td>%B2</td>
<td>26117</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>4</td>
</tr>
<tr>
<td>%B3</td>
<td>1184459</td>
<td>2335702</td>
<td>5460230</td>
<td>79900378</td>
<td>16</td>
</tr>
</tbody>
</table>
-- R       ------- %B3 - ------- %B3 - ------- %B3 + ------- %B3
-- R       1645371  548457  182819  1645371
-- R +
-- R 43953929  15 13420192  14  553986  13 193381378  12
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 548457  182819  3731  1645371
-- R +
-- R 35978916  11 358660781  10 271667666  9 118784873  8
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 182819  1645371  1645371  548457
-- R +
-- R 337505020  7 1389370  6  688291  5 3378002  4
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 1645371  11193  4459  42189
-- R +
-- R 140671876  3 32325724  2  8270  9741532
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 1645371  548457  343  1645371
-- R +
-- R 91729  19 487915  18 4114333  17 1276987  16
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 705159  705159  705159  235053
-- R +
-- R 13243117  15 16292173  14 26536060  13 722714  12
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 705159  705159  705159  18081
-- R +
-- R 5382578  11 15449995  10 14279770  9 6603890  8
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 100737  235053  235053 100737
-- R +
-- R 409930  7 37340389  6 34893715  5 26686318  4
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 6027  705159  705159  705159
-- R +
-- R 801511  3 17206178  2 4406102  377534
-- R       ------- %B3 + ------- %B3 + ------- %B3 + ------- %B3
-- R 26117  705159  705159  705159
-- R ]
-- R [ %B4,
-- R |
-- R |
-- R 1184459  19 2335702  18 5460230  17 79900378  16
-- R       ------- %B4 + ------- %B4 + ------- %B4 + ------- %B4
-- R 1645371  548457  182819  1645371
-- R +
-- R 43953929  15 13420192  14  553986  13 193381378  12
-- R       ------- %B4 + ------- %B4 + ------- %B4 + ------- %B4
<table>
<thead>
<tr>
<th>Package: ZDSOLVE ZERODIMENSIONALSOLVEPACKAGE</th>
</tr>
</thead>
</table>

```plaintext
---R +
---R 337505020 7 1389370 6 688291 5 3378002 4
---R %B5 + %B5 + %B5 + %B5 + %B5
---R 1645371 11193 4459 42189
---R +
---R 140671876 3 32325724 2 8270 9741532
---R %B5 + %B5 - %B5 - %B5 - %B5
---R 1645371 548457 343 1645371

---R
---R 91729 19 487915 18 4114333 17 1276987 16
---R %B5 + %B5 + %B5 + %B5 + %B5
---R 705159 705159 705159 235053
---R +
---R 13243117 15 16292173 14 26536060 13 722714 12
---R %B5 + %B5 - %B5 - %B5 - %B5
---R 705159 705159 705159 18081
---R +
---R 5382578 11 15449995 10 14279770 9 6603890 8
---R %B5 - %B5 - %B5 - %B5 - %B5
---R 100737 235053 235053 100737
---R +
---R 409930 7 37340389 6 34893715 5 26686318 4
---R %B5 - %B5 - %B5 - %B5 - %B5
---R 6027 705159 705159 705159
---R +
---R 801511 3 17206178 2 4406102 377534
---R %B5 - %B5 - %B5 - %B5 + %B5
---R 26117 705159 705159 705159

---R ]
---R ,
---R [%B6,
---R
---R 1184459 19 2335702 18 5460230 17 79900378 16
---R %B6 - %B6 - %B6 - %B6 + %B6
---R 1645371 548457 182819 1645371
---R +
---R 43953929 15 13420192 14 553986 13 193381378 12
---R %B6 + %B6 + %B6 + %B6 + %B6
---R 548457 182819 3731 1645371
---R +
---R 35978916 11 358660781 10 271667666 9 118784873 8
---R %B6 + %B6 + %B6 + %B6 + %B6
---R 182819 1645371 1645371 548457
---R +
---R 337505020 7 1389370 6 688291 5 3378002 4
---R %B6 + %B6 + %B6 + %B6 + %B6
---R 1645371 11193 4459 42189
---R +
```
(12) 8

Type: PositiveInteger

(13) }

[ ]

Type: List/List(RealClosure(Fraction(Integer)))

# lr

[ ]

Type: List(RealClosure(Fraction(Integer))))
--R 3
--R (18) z + y + x + t - 1
--R Type: Polynomial(Integer)
--E 18

--S 19 of 29
lf := [f0, f1, f2, f3]
--R
--R
--R (19)
--R 3 3 3 3
--R [z + y + x + t - 1, z + y + x + t - 1, z + y + x + t - 1, z + y + x + t - 1]
--R Type: List(Polynomial(Integer))
--E 19

--S 20 of 29
lts := triangSolve(lf)$pack
--R
--R
--R (20)
--R [ 2 3 3
--R {t + t + 1, z - z - t + t,}
--R
--R 3 2 2 3 6 3 3 2
--R (3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z
--R +
--R
--R 6 3 9 6 3
--R 3t - 3t + 5t - 3t
--R ,
--R
--R x + y + z}
--R ,
--R
--R 16 13 10 7 4 2
--R {t - 6t + 9t + 4t + 15t - 54t + 27,
--R
--R 15 14 13 12 11
--R 4907232t + 40893984t - 115013088t + 22805712t + 36330336t
--R +
--R
--R 10 9 8 7
--R 162959040t - 159859440t - 156802608t + 117168768t
--R +
--R
--R 6 5 4 3
--R 126282384t - 129351600t + 306646992t + 475302816t
--R +
--R
--R 2
--R - 1006837776t - 237269088t + 480716208
--R *
--R
--R z
--R +
\begin{verbatim}
--R  54  51  48  46  45  43  42
--R + 48t - 912t + 8232t - 72t - 46848t + 1152t + 186324t
--R + 40  39  38  37  36  35
--R - 3780t - 543144t - 3168t - 21384t + 1175251t + 41184t
--R + 34  33  32  31  30
--R 278003t - 1843242t - 301815t - 1440726t + 1912012t
--R + 29  28  27  26  25
--R 1442826t + 4696626t - 922481t - 4816188t - 10583524t
--R + 24  23  22  21  20
--R - 208751t + 11472138t + 16762859t - 857663t - 19328175t
--R + 19  18  17  16  15
--R - 18270421t + 4914903t + 22483044t + 12926517t - 8605511t
--R + 14  13  12  11  10
--R - 17455518t - 5014597t + 8108814t + 9465535t + 190542t
--R + 9  8  7  6  5  4
--R - 4305624t - 2226123t + 661905t + 1169775t + 226260t - 209952t
--R + 3
--R - 141183t + 27216t
--R ,
--R 3 2 2 3 6 3 3 2
--R (3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z
--R + 6 3 9 6 3
--R ,
--R (3t - 6t + 3)z + t - 3t + 5t - 3t
--R ,
--R 3
--R x + y + z + t - 1}
--R ,
--R 2
--R \{t,z - 1,y - 1,x + y\}, \{t - 1,z,y - 1,x + y\}, \{t - 1,z - 1,z y + 1,x\},
--R ,
--R 16 13 10 7 4 2
--R \{t - 6t + 9t + 4t + 15t - 54t + 27,}
--R 29 28 27 26 25
--R 4907232t + 40893984t - 115013088t - 1730448t - 168139584t
--R + 24 23 22 21
--R 738024480t - 195372288t + 315849456t - 2567279232t
--R +
--R 20 19 18 17
\end{verbatim}
\[ 937147968t + 1026357696t + 4780488240t - 2893767696t + 12t + 11t + 10t + 9t + 8t + 7t + 6t + 5t + 4t + 3t + 2t + z + 68t + 65t + 62t + 60t + 59t + 57t + 56t - 48t + 1152t - 13560t + 360t + 103656t - 7560t - 572820t + 54t + 53t + 52t + 51t + 50t + 49t + 48t + 47t + 46t + 45t + 44t + 43t + 42t + 41t + 40t + 39t - 3756573t - 2093410t + 71546495t + 19699032t + 35025028t + 38t + 37t + 36t + 35t + 34t + 33t + 32t + 31t + 30t + 29t + 349607642t - 93299834t - 551563167t - 630995176t + 186818962t + 28t + 27t + 26t + 25t + 24t + 23t + 22t + 21t - 1609479791t + 595738126t + 1198787136t + 4342832069t + 20t + 19t + 18t + 17t + 16t + 15t + 14t + 13t + 12t + 11t + 10t + 9t - 7404299976t - 157295760t + 29124027630t + 14856038208t \]
\[\begin{align*}
8 - 16184101410t - 9780477840t - 3586674168t + 2884297248 & \quad + \\
3 + (6t - 6)z + (6t - 12t + 3)z + 2t - 6t + t + 3t)y & \quad + \\
(3t - 3)z + (6t - 12t + 6)z + (4t - 12t + 11t - 3)z + t - 4t & \quad + \\
5t - 2t & , \\
x + y + z + t - 1} & , \\
\{t - 1, z - 1, y, x + z\}, \\
\{t + t + t - 2t - 2t - 2t + 19t + 19t - 8, \\
7 - 463519t + 3586833t + 9494955t - 8539305t - 33283098t & \quad + \\
2 & , \\
35479377t + 46263256t - 17419896 & , \\
4 + 3 3 6 3 & 2 \quad 3 \\
3z + (9t - 9)z + (12t - 24t + 9)z + (- 152t + 219t - 67)z & \quad + \\
6 - 41t + 57t + 25t - 57t + 16 & \quad * \\
y & , \\
3 4 6 3 & 2 
\end{align*}\]
(3t - 3)z + (9t - 18t + 9)z + (-181t + 270t - 89)z
+ (-92t + 135t + 49t - 135t + 43)z + 27t - 54t + 396t
+ (-486t + 144)

x + y + z + t - 1

{t, z - t + 1, y - 1, x - 1}, {t - 1, z, y, x}, {t, z - 1, y, x}, {t, z, y - 1, x},
{t, z, y, x - 1}  

Type: List(RegularChain(Integer,[x,y,z,t]))

univariateSolve(lf)$pack

[[complexRoots= ?,coordinates= [x - 1, y - 1, z + 1, t - %A]],
[complexRoots= ?,coordinates= [x, y - 1, z, t - %A]],
[complexRoots= ? - 1,coordinates= [x, y, z, t - %A]],
[complexRoots= ?,coordinates= [x - 1, y, z, t - %A]],
[complexRoots= ? - 2,coordinates= [x - 1, y + 1, z, t - 1]],
[complexRoots= ?,coordinates= [x + 1, y - 1, z, t - 1]],
[complexRoots= ? - 1,coordinates= [x - 1, y + 1, z - 1, t]],
[complexRoots= ? + 1,coordinates= [x + 1, y - 1, z - 1, t]],

[complexRoots= ? - 2? + 3? - 3,
coordinates= [2x + %A + %A - 1, 2y + %A + %A - 1, z - %A, t - %A]]

[complexRoots= ? + 3? - 2? + 3? - 3,
coordinates= [x - %A, y - %A, z + %A + 2%A - 1, t - %A]]

[complexRoots= ? - ? - 2? + 3,
coordinates= [x + %A - %A - 1, y + %A - %A - 1, z - %A + 2%A + 1, t - %A]]

[complexRoots= ? + 1,coordinates= [x - 1, y - 1, z, t - %A]],
\[
\text{coordinates } = \begin{bmatrix} 2x - \%A - \%A - 1, & y + \%A, & 2z - \%A - \%A - 1, & t + \%A \end{bmatrix}
\]

\[
\text{coordinates } = \begin{bmatrix} 8x + \%A + 8\%A - 8, & 2y - \%A, & 2z - \%A, & 2t - \%A \end{bmatrix}
\]

\[
\text{coordinates } = \begin{bmatrix} 9x - 2\%A + 4\%A - \%A + 2, & 9y + \%A - 2\%A + 5\%A - 1, & 9z + \%A - 2\%A + 5\%A - 1, & 9t + \%A - 2\%A - 4\%A - 1 \end{bmatrix}
\]
\[
\text{complexRoots} = \{-11\} + 37,
\]
\[
\text{coordinates} = \begin{bmatrix}
2 & 2 & 2 & 2 \\
3x - \%A + 7,6y + \%A + 3\%A - 7,3z - \%A + 7,6t + \%A - 3\%A - 7
\end{bmatrix}
\]

\[
\text{complexRoots} = \{+1\} \text{, coordinates} = [x - 1,y,z - 1,t + 1],
\]
\[
\text{complexRoots} = \{+2\} \text{, coordinates} = [x,y - 1,z + 1,t - 1],
\]
\[
\text{complexRoots} = \{-2\} \text{, coordinates} = [x,y - 1,z + 1,t - 1],
\]
\[
\text{complexRoots} = \{?\} \text{, coordinates} = [x + 1,y,z - 1,t - 1],
\]
\[
\text{complexRoots} = \{-2\} \text{, coordinates} = [x - 1,y,z + 1,t - 1],
\]
\[
\text{complexRoots} = \{?\} \text{, coordinates} = [x + 1,y,z - 1,t - 1],
\]
\[
\text{complexRoots} = \{5\} + 16z + 30t + 57,
\]
\[
\text{coordinates} = \begin{bmatrix}
3 & 2 & 3 & 2 \\
151x + 15\%A + 54\%A + 104\%A + 93, 151y - 10\%A - 36\%A - 19\%A - 62, \\
3 & 2 & 3 & 2 \\
\end{bmatrix}
\]

\[
\text{complexRoots} = \{-?\} - 2\%A + 3,
\]
\[
\text{coordinates} = [x - \%A + 2\%A + 1,y + \%A - \%A - 1,z - \%A,t + \%A - \%A - 1]
\]
\[
\text{complexRoots} = \{?\} + 2\%A - 8\%A + 48,
\]
\[
\text{coordinates} = \begin{bmatrix}
3 & 3 & 3 \\
8x - \%A + 4\%A - 8,2y + \%A,8z + \%A - 8\%A + 8,8t - \%A + 4\%A - 8
\end{bmatrix}
\]

\[
\text{complexRoots} = \{+?\} - 2\%A - 4? + 5\%A + 8,
\]
\[
\text{coordinates} = [3x + \%A - 1,3y + \%A - 1,3z + \%A - 1,t - \%A]
\]

\[
\text{complexRoots} = \{+3\} - 1, \text{ coordinates} = [x - \%A,y - \%A,z - \%A,t - \%A]
\]
PACKAGE ZDSSOLVE ZERODIMENSIONALSOLVE PACKAGE

--S 22 of 29

\[ts := \{t + t + 1, z - z - t + t,\}\]

\[\{t + t + 1, z - z - t + t,\}\]

\[\frac{3}{2} 2 2 3 6 3 3 2\]

\[\frac{(3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z}{6 3 9 6 3}\]

\[x + y + z\]

Type: RegularChain(Integer,[x,y,z,t])

--E 22

--S 23 of 29

univariateSolve(ts)$pack

\[\{\text{complexRoots} = ? + 5? + 16? + 30? + 57,\text{coordinates} = [151x + 15%A + 54%A + 104%A + 93, 151y - 10%A - 36%A - 19%A - 62, 151z - 5%A - 18%A - 85%A - 31, 151t - 5%A - 18%A - 85%A - 31]\}\]

\[\{\text{complexRoots} = ? - ? - 2? + 3,\text{coordinates} = [x - %A + 2%A + 1,y + %A - %A - 1,z - %A,t + %A - %A - 1]\}\]

\[\{\text{complexRoots} = ? + 2? - 8? + 48,\text{coordinates} = [8x - %A + 4%A - 8,2y + %A,8z + %A - 8%A + 8,8t - %A + 4%A - 8]\}\]

Type: List(Record(complexRoots: SparseUnivariatePolynomial(Integer),coordinates: List(Polynomial(Integer))))
realSolve(ts) \$ pack

\[
\begin{align*}
\text{Type: } & \text{List(List(RealClosure(Fraction(Integer)))))}
\end{align*}
\]

\[
\begin{array}{cccccccccc}
1 & 15 & 2 & 14 & 1 & 13 & 4 & 12 & 11 & 11 & 10 \\
27 & 27 & 27 & 27 & 27 & 27 & 27 & 27 & 27 &
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 9 & 14 & 8 & 1 & 7 & 2 & 6 & 1 & 5 & 2 & 4 & 3 \\
27 & 27 & 27 & 9 & 3 & 9 & 4 & 2 &
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 15 & 1 & 14 & 1 & 13 & 2 & 12 & 11 & 11 & 2 & 10 \\
54 & 27 & 54 & 27 & 54 & 27 & 54 & 27 &
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 9 & 7 & 8 & 1 & 7 & 1 & 6 & 1 & 5 & 1 & 4 & 3 \\
54 & 27 & 54 & 9 & 6 & 9 &
\end{array}
\]

\[
\begin{array}{cccccccccc}
2 & 2 & 1 & 3 \\
3 & 2 & 2 &
\end{array}
\]

\[
\begin{array}{cccccccccc}
1 & 15 & 1 & 14 & 1 & 13 & 2 & 12 & 11 & 11 & 2 & 10 \\
54 & 27 & 54 & 27 & 54 & 27 & 54 & 27 &
\end{array}
\]

\[
\begin{array}{cccccccccc}
2 & 2 & 1 & 3 \\
3 & 2 & 2 &
\end{array}
\]
PACKAGE ZDSOLVE ZERODIMENSIONALSOLVEPACKAGE

\[
\begin{align*}
&\text{1 9 7 8 1 7 1 6 1 5 1 4 3} \\
&\text{2 2 1 3} \\
&\text{3 2 2} \\
&\%B32, \\
&\%B33, \\
&1 15 2 14 1 13 4 12 11 11 4 10 \\
&27 27 27 27 27 27 27 \\
&1 9 14 8 1 7 2 6 1 5 2 4 3 \\
&27 27 27 9 3 9 \\
&4 2 \\
&\%B33 - \%B33 - 2 \\
&3 \\
&1 15 1 14 1 13 2 12 11 11 2 10 \\
&54 27 54 27 54 27 \\
&1 9 7 8 1 7 1 6 1 5 1 4 3 \\
&54 27 54 9 6 9 \\
&2 2 1 3 \\
&3 2 2 \\
&1 15 1 14 1 13 2 12 11 11 2 10 \\
&54 27 54 27 54 27 \\
&1 9 7 8 1 7 1 6 1 5 1 4 3 \\
&54 27 54 9 6 9 \\
&2 2 1 3 \\
&3 2 2
\end{align*}
\]
CHAPTER 27. CHAPTER Z

--R  - %B23  +  - %B23  -  - %B23  -  -
--R  9 3 2 2
--R ]
--R ,
--R [%B24,
--R ]

--R 1 15 1 14 1 13 2 12 11 11 2 10
--R -  - %B24  -  - %B24  -  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 54 27 54 27 54 27
--R +
--R 1 9 7 8 1 7 1 6 1 5 1 4 3
--R -  - %B24  -  - %B24  -  - %B24  -  - %B24  -  - %B24  -  - %B24  -  - %B24
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R -  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 3 2 2
--R ,
--R [%B28,
--R ]

--R 1 15 1 14 1 13 2 12 11 11 11
--R -  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 54 27 54 27 54 27
--R +
--R 2 10 1 9 7 8 1 7 1 6 1 5
--R -  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 27 54 27 54 9 6
--R +
--R 1 4 2 2 1 1
--R -  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 9 3 2 2
--R ]
--R ,
--R [%B24,
--R ]

--R 1 15 1 14 1 13 2 12 11 11 2 10
--R -  - %B24  -  - %B24  -  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 54 27 54 27 54 27
--R +
--R 1 9 7 8 1 7 1 6 1 5 1 4 3
--R -  - %B24  -  - %B24  -  - %B24  -  - %B24  -  - %B24  -  - %B24  -  - %B24
--R 54 27 54 9 6 9
--R +
--R 2 2 1 3
--R -  - %B24  +  - %B24  +  - %B24  +  - %B24
--R 3 2 2
--R ,
--R [%B29,
--R
  1 15 1 14 1 13 2 12 11 11
--R - %B29 + -- %B25 + -- %B25 + -- %B25 - -- %B24 + -- %B24
--R  54  27  54  27  54
--R +
--R  2 10 1 9 7 8 1 7 1 6 1 5
--R - -- %B24 + -- %B24 - -- %B24 + -- %B24 + - %B24 - -- %B24
--R  27  54  27  54  9  6
--R +
--R  1 4 2 2 1 1
--R - %B24 - %B24 - - %B24 - -
--R  9  3  2  2
--R ]
--R ,
--R [ %B25,
--R
  1 15 1 14 1 13 2 12 11 11 2 10
--R - -- %B25 - -- %B25 - -- %B25 + -- %B25 + -- %B25
--R  54  27  54  27  54  27
--R +
--R  1 9 7 8 1 7 1 6 1 5 1 4 3
--R  54  27  54  27  54  9  6  9
--R +
--R  2 2 1 3
--R - - %B25 + - %B25 + -
--R  3  2  2
--R ,
--R %B26,
--R
  1 15 1 14 1 13 2 12 11 11
--R - %B26 + -- %B25 + -- %B25 + -- %B25 - -- %B25 + -- %B25
--R  54  27  54  27  54
--R +
--R  2 10 1 9 7 8 1 7 1 6 1 5
--R - -- %B25 + -- %B25 + -- %B25 + -- %B25 + - %B25 + - %B25
--R  27  54  27  54  9  6
--R +
--R  1 4 2 2 1 1
--R - %B25 + - %B25 - - %B25 - -
--R  9  3  2  2
--R ]
--R ,
--R [ %B25,
Type: List(List(RealClosure(Fraction(Integer))))


PACKAGE ZDSOLVE ZERODIMENSIONALSOLVEPACKAGE

--E 27

--S 28 of 29

[approximate(r,1/10**21)::Float for r in lpr2.1]

--R

--R (28)

--R [0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291]

--R Type: List(Float)

--E 28

--S 29 of 29

)show ZeroDimensionalSolvePackage

--R

--R ZeroDimensionalSolvePackage(R: Join(OrderedRing,EuclideanDomain,CharacteristicZero,RealConstant),ls: List(Symbol),ls2: List(Symbol)) is a package constructor

--R Abbreviation for ZeroDimensionalSolvePackage is ZDSOLVE

--R This constructor is exposed in this frame.

--R Issue )edit bookvol10.4.pamphlet to see algebra source code for ZDSOLVE

--R

--R------------------------------- Operations ----------------------------------


--R convert : Polynomial(R) -> Polynomial(RealClosure(Fraction(R)))

--R convert : NewSparseMultivariatePolynomial(R,OrderedVariableList(ls2)) -> Polynomial(RealClosure(Fraction(R)))

--R convert : SquareFreeRegularTriangularSet(R,IndexedExponents(OrderedVariableList(ls2)),OrderedVariableList(ls2),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls2))) -> List(NewSparseMultivariatePolynomial(R,OrderedVariableList(ls2)))

--R positiveSolve : RegularChain(R,ls) -> List(List(RealClosure(Fraction(R))))

--R positiveSolve : (List(Polynomial(R)),Boolean,Boolean) -> List(List(RealClosure(Fraction(R))))

--R positiveSolve : (List(Polynomial(R)),Boolean) -> List(List(RealClosure(Fraction(R))))

--R positiveSolve : List(Polynomial(R)) -> List(List(RealClosure(Fraction(R))))

--R realSolve : RegularChain(R,ls) -> List(List(RealClosure(Fraction(R))))

--R realSolve : (List(Polynomial(R)),Boolean,Boolean,Boolean) -> List(List(RealClosure(Fraction(R))))

--R realSolve : (List(Polynomial(R)),Boolean,Boolean) -> List(List(RealClosure(Fraction(R))))

--R realSolve : (List(Polynomial(R)),Boolean) -> List(List(RealClosure(Fraction(R))))

--R realSolve : List(Polynomial(R)) -> List(List(RealClosure(Fraction(R))))

--R squareFree : RegularChain(R,ls) -> List(SquareFreeRegularTriangularSet(R,IndexedExponents(OrderedVariableList(ls2)),OrderedVariableList(ls2),NewSparseMultivariatePolynomial(R,OrderedVariableList(ls2))))

--R triangSolve : (List(Polynomial(R)),Boolean,Boolean) -> List(RegularChain(R,ls))

--R triangSolve : (List(Polynomial(R)),Boolean) -> List(RegularChain(R,ls))

--R triangSolve : List(Polynomial(R)) -> List(RegularChain(R,ls))

--R univariateSolve : RegularChain(R,ls) -> List(Record(complexRoots: SparseUnivariatePolynomial(R),coordinates: List(Polynomial(R))))

--R univariateSolve : (List(Polynomial(R)),Boolean,Boolean,Boolean) -> List(Record(complexRoots: SparseUnivariatePolynomial(R),coordinates: List(Polynomial(R))))

--R univariateSolve : (List(Polynomial(R)),Boolean,Boolean) -> List(Record(complexRoots: SparseUnivariatePolynomial(R),coordinates: List(Polynomial(R))))

--R univariateSolve : (List(Polynomial(R)),Boolean) -> List(Record(complexRoots: SparseUnivariatePolynomial(R),coordinates: List(Polynomial(R))))

--R univariateSolve : List(Polynomial(R)) -> List(Record(complexRoots: SparseUnivariatePolynomial(R),coordinates: List(Polynomial(R))))

--E 29

)spool

)lisp (bye)
A package for computing symbolically the complex and real roots of zero-dimensional algebraic systems over the integer or rational numbers. Complex roots are given by means of univariate representations of irreducible regular chains. Real roots are given by means of tuples of coordinates lying in the RealClosure of the coefficient ring.

This constructor takes three arguments. The first one $R$ is the coefficient ring. The second one $\text{ls}$ is the list of variables involved in the systems to solve. The third one must be $\text{concat}(\text{ls}, s)$ where $s$ is an additional symbol used for the univariate representations.

WARNING. The third argument is not checked.

All operations are based on triangular decompositions. The default is to compute these decompositions directly from the input system by using the RegularChain domain constructor.

The lexTriangular algorithm can also be used for computing these decompositions (see LexTriangularPackage package constructor).

For that purpose, the operations univariateSolve, realSolve and positiveSolve admit an optional argument.

The ZeroDimensionalSolvePackage package constructor provides operations for computing symbolically the complex or real roots of zero-dimensional algebraic systems.

The package provides no multiplicity information (i.e. some returned roots may be double or higher) but only distinct roots are returned.

Complex roots are given by means of univariate representations of irreducible regular chains. These representations are computed by the univariateSolve operation (by calling the InternalRationalUnivariateRepresentationPackage package constructor which does the job).

Real roots are given by means of tuples of coordinates lying in the RealClosure of the coefficient ring. They are computed by the realSolve and positiveSolve operations. The former computes all the solutions of the input system with real coordinates whereas the later concentrate on the solutions with (strictly) positive coordinates. In
both cases, the computations are performed by the RealClosure constructor.

Both computations of complex roots and real roots rely on triangular decompositions. These decompositions can be computed in two different ways. First, by applying the zeroSetSplit operation from the REGSET domain constructor. In that case, no Groebner bases are computed. This strategy is used by default. Secondly, by applying the zeroSetSplit from LEXTRIPK. To use this later strategy with the operations univariateSolve, realSolve and positiveSolve one just needs to use an extra boolean argument.

Note that the way of understanding triangular decompositions is detailed in the example of the RegularTriangularSet constructor.

The ZeroDimensionalSolvePackage constructor takes three arguments. The first one R is the coefficient ring; it must belong to the categories OrderedRing, EuclideanDomain, CharacteristicZero and RealConstant. This means essentially that R is Integer or Fraction(Integer). The second argument ls is the list of variables involved in the systems to solve. The third one MUST BE concat(ls,s) where s is an additional symbol used for the univariate representations. The abbreviation for ZeroDimensionalSolvePackage is ZDSOLVE.

We illustrate now how to use the constructor ZDSOLVE by two examples: the Arnborg and Lazard system and the L-3 system (Aubry and Moreno Maza). Note that the use of this package is also demonstrated in the example of the LexTriangularPackage constructor.

Define the coefficient ring.

```
R := Integer
       Integer
       Type: Domain
```

Define the lists of variables:

```
ls : List Symbol := [x,y,z,t]
    [x,y,z,t]
    Type: List Symbol
```

and:

```
ls2 : List Symbol := [x,y,z,t,new()$Symbol]
    [x,y,z,t,%A]
    Type: List Symbol
```

Call the package:

```
pack := ZDSOLVE(R,ls,ls2)
       ZeroDimensionalSolvePackage(Integer,[x,y,z,t],[x,y,z,t,%A])
```
CHAPTER 27. CHAPTER Z

Type: Domain

Define a polynomial system (Arnborg-Lazard)

\[ p_1 := x^{2}y^{2}z + x^{2}y^{2}z^2 + x^{2}y^{2}z + x^{2}y^2 + x^{2}y + x^2 + yz + y^2 + x + y \]
\[ Type: Polynomial Integer \]

\[ p_2 := x^{2}y^{2}z^2 + x^{2}y^{2}z^2 + x^{2}y^{2}z + x^{2}y^2 + yz + x + z \]
\[ Type: Polynomial Integer \]

\[ p_3 := x^{2}y^{2}z^2 + x^{2}y^{2}z^2 + x^{2}y^{2}z + x^{2}y^2 + xz + z + 1 \]
\[ Type: Polynomial Integer \]

\[ lp := [p_1, p_2, p_3] \]
\[ Type: List Polynomial Integer \]

Note that these polynomials do not involve the variable t; we will use it in the second example.

First compute a decomposition into regular chains (i.e. regular triangular sets).

\[ \text{triangSolve}(lp) \]
\[ \{ \]
\[ 20 19 18 17 16 15 14 13 12 \]
\[ z - 6z - 41z + 71z + 106z + 92z + 197z + 145z + 257z + \]
\[ 11 10 9 8 7 6 5 4 3 \]
\[ 278z + 201z + 145z + 197z + 92z + 106z + 71z + \]
\[ 2 \]
\[ - 41z - 6z + 1 \]
\[ 14745844z + 50357474z - 130948867z - 185261586z + \]
\[ 15 14 13 12 \]

CHAPTER 27. CHAPTER Z
We can see easily from this decomposition (consisting of a single regular chain) that the input system has 20 complex roots.

Then we compute a univariate representation of this regular chain.

```
univariateSolve(lp)$pack
[
  complexRoots =
    12 11 10 9 8 7 6 5 4 3
    +
    2
    24?  - 12?  + 1
  ,
  coordinates =
    11 10 9 8 7 6
    63x  + 62%A  - 721%A  + 1220%A  + 705%A  - 285%A  + 1512%A
    +
```
We see that the zeros of our regular chain are split into three components. This is due to the use of univariate polynomial factorization.

Each of these components consists of two parts. The first one is an irreducible univariate polynomial $p(\%)$ which defines a simple algebraic extension of the field of fractions of $R$. The second one consists of multivariate polynomials $\text{pol1}(x,\%), \text{pol2}(y,\%)$ and $\text{pol3}(z,\%)$. Each of these polynomials involves two variables: one is an indeterminate $x$, $y$ or $z$ of the input system $lp$ and the other is $\%$ which represents any root of $p(\%)$. Recall that this $\%$ is the last element of the third parameter of ZDSOLVE. Thus any complex root $\%$ of $p(\%)$ leads to a solution of the input system $lp$ by replacing $\%$ by this $\%$ in $\text{pol1}(x,\%), \text{pol2}(y,\%)$ and $\text{pol3}(z,\%)$. Note that these polynomials $\text{pol1}(x,\%), \text{pol2}(y,\%)$ and $\text{pol3}(z,\%)$ have degree one w.r.t. $x$, $y$ or $z$ respectively. This is always the case for all univariate representations. Hence the operation $\text{univariateSolve}$ replaces a system of multivariate polynomials by a list of univariate polynomials, what justifies its name. Another example of univariate representations illustrates the LexTriangularPackage package constructor.

We now compute the solutions with real coordinates:

```plaintext
lr := realSolve(lp)$pack
[ [\%1, 
  [1184459 19 2335702 18 5460230 17 79900378 16
   -------- \%1 - -------- \%1 - -------- \%1 + -------- \%1
  1645371 548457 182819 1645371
] ]
```
\[
\begin{align*}
&\text{\textbf{PACKAGE} ZDSOLVE ZERODIMENSIONALSOLVEPACKAGE} \\
&\text{4227} \\
&+ \\
&\quad 43953929 \quad 15 \quad 13420192 \quad 14 \quad 553986 \quad 13 \quad 193381378 \quad 12 \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \\
&\quad 548457 \quad 182819 \quad 3731 \quad 1645371 \\
&+ \\
&\quad 35978916 \quad 11 \quad 358660781 \quad 10 \quad 271667666 \quad 9 \quad 118784873 \quad 8 \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \\
&\quad 182819 \quad 1645371 \quad 1645371 \quad 548457 \\
&+ \\
&\quad 337505020 \quad 7 \quad 1389370 \quad 6 \quad 688291 \quad 5 \quad 3378002 \quad 4 \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \\
&\quad 1645371 \quad 1645371 \\
&+ \\
&\quad 140671876 \quad 3 \quad 32325724 \quad 2 \quad 8270 \quad 9741532 \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \\
&\quad 1645371 \quad 1645371 \\
&+ \\
&\quad 91729 \quad 19 \quad 487915 \quad 18 \quad 4114333 \quad 17 \quad 1276987 \quad 16 \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \\
&\quad 705159 \quad 705159 \quad 18081 \\
&+ \\
&\quad 13243117 \quad 15 \quad 16292173 \quad 14 \quad 26536060 \quad 13 \quad 722714 \quad 12 \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \\
&\quad 705159 \quad 705159 \quad 235053 \\
&+ \\
&\quad 5382578 \quad 11 \quad 15449995 \quad 10 \quad 14279770 \quad 9 \quad 6603890 \quad 8 \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \\
&\quad 100737 \quad 235053 \quad 100737 \\
&+ \\
&\quad 409930 \quad 7 \quad 37340389 \quad 6 \quad 34893715 \quad 5 \quad 26686318 \quad 4 \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \\
&\quad 6027 \quad 705159 \quad 705159 \\
&+ \\
&\quad 801511 \quad 3 \quad 17206178 \quad 2 \quad 4406102 \quad 377534 \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad - \quad \text{---} \%B_1 \quad + \quad \text{---} \%B_1 \\
&\quad 26117 \quad 705159 \quad 705159 \\
\]
\[
\begin{align*}
&\text{\text{
\begin{array}{ccccccc}
91729 & 19 & 487915 & 18 & 4114333 & 17 & 1276987 & 16 \\
705159 & 705159 & 705159 & 235053
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
13243117 & 15 & 16292173 & 14 & 26536060 & 13 & 722714 & 12 \\
705159 & 705159 & 705159 & 18081
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
5382578 & 11 & 15449995 & 10 & 14279770 & 9 & 6603890 & 8 \\
100737 & 235053 & 235053 & 100737
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
409930 & 7 & 37340389 & 6 & 34893715 & 5 & 26686318 & 4 \\
6027 & 705159 & 705159 & 705159
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
801511 & 3 & 17206178 & 2 & 4406102 & 377534 \\
26117 & 705159 & 705159 & 705159
\end{array}
\}} \\
&\text{\text{
\begin{array}{ccccccc}
1184459 & 19 & 2335702 & 18 & 5460230 & 17 & 79900378 & 16 \\
1645371 & 548457 & 182819 & 1645371
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
43953929 & 15 & 13420192 & 14 & 553986 & 13 & 193381378 & 12 \\
548457 & 182819 & 3731 & 1645371
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
35978916 & 11 & 358660781 & 10 & 271667666 & 9 & 118784873 & 8 \\
182819 & 1645371 & 1645371 & 548457
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
337505020 & 7 & 1389370 & 6 & 688291 & 5 & 3378002 & 4 \\
1645371 & 11193 & 4459 & 42189
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
140671876 & 3 & 32325724 & 2 & 8270 & 9741532 \\
1645371 & 548457 & 343 & 1645371
\end{array}
\}} \\
&\text{\text{
\begin{array}{ccccccc}
91729 & 19 & 487915 & 18 & 4114333 & 17 & 1276987 & 16 \\
705159 & 705159 & 705159 & 235053
\end{array}
\}} + \\
&\text{\text{
\begin{array}{ccccccc}
13243117 & 15 & 16292173 & 14 & 26536060 & 13 & 722714 & 12 \\
705159 & 705159 & 705159 & 18081
\end{array}
\}}
\end{align*}
\]
<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>5382578</td>
<td>11</td>
<td>15449995</td>
<td>10</td>
<td>14279770</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B4</td>
<td>-------</td>
<td>%B4</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>100737</td>
<td>235053</td>
<td>235053</td>
<td>100737</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>409930</td>
<td>7</td>
<td>37340389</td>
<td>6</td>
<td>34893715</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B4</td>
<td>-------</td>
<td>%B4</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>6027</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>801511</td>
<td>3</td>
<td>17206178</td>
<td>2</td>
<td>4406102</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B4</td>
<td>-------</td>
<td>%B4</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>26117</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td></td>
</tr>
</tbody>
</table>
| ]

% B5,

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1184459</td>
<td>19</td>
<td>2335702</td>
<td>18</td>
<td>5460230</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>-------</td>
<td>%B5</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>1645371</td>
<td>548457</td>
<td>182819</td>
<td>1645371</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>43953929</td>
<td>15</td>
<td>13420192</td>
<td>14</td>
<td>553986</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>+ -------</td>
<td>%B5</td>
<td>+ -------</td>
</tr>
<tr>
<td></td>
<td>548457</td>
<td>182819</td>
<td>3731</td>
<td>1645371</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>35978916</td>
<td>11</td>
<td>358660781</td>
<td>10</td>
<td>271667666</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>+ -------</td>
<td>%B5</td>
<td>+ -------</td>
</tr>
<tr>
<td></td>
<td>182819</td>
<td>1645371</td>
<td>1645371</td>
<td>548457</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>337505020</td>
<td>7</td>
<td>1389370</td>
<td>6</td>
<td>688291</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>+ -------</td>
<td>%B5</td>
<td>+ -------</td>
</tr>
<tr>
<td></td>
<td>1645371</td>
<td>11193</td>
<td>4459</td>
<td>42189</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>140671976</td>
<td>3</td>
<td>32235724</td>
<td>2</td>
<td>8270</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>+ -------</td>
<td>%B5</td>
<td>- -------</td>
</tr>
<tr>
<td></td>
<td>1645371</td>
<td>548457</td>
<td>343</td>
<td>1645371</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>91729</td>
<td>19</td>
<td>487915</td>
<td>18</td>
<td>4114333</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>+ -------</td>
<td>%B5</td>
<td>+ -------</td>
</tr>
<tr>
<td></td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>235053</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>13243117</td>
<td>15</td>
<td>16292173</td>
<td>14</td>
<td>26536060</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>+ -------</td>
<td>%B5</td>
<td>+ -------</td>
</tr>
<tr>
<td></td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td>18081</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>5382578</td>
<td>11</td>
<td>15449995</td>
<td>10</td>
<td>14279770</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>-------</td>
<td>%B5</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>100737</td>
<td>235053</td>
<td>235053</td>
<td>100737</td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>409930</td>
<td>7</td>
<td>37340389</td>
<td>6</td>
<td>34893715</td>
</tr>
<tr>
<td>-</td>
<td>-------</td>
<td>%B5</td>
<td>-------</td>
<td>%B5</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>6027</td>
<td>705159</td>
<td>705159</td>
<td>705159</td>
<td></td>
</tr>
<tr>
<td>1184459</td>
<td>19 2335702</td>
<td>18 5460230</td>
<td>17 79900378</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>%B6 + -------</td>
<td>%B6 + -------</td>
<td>%B6 + -------</td>
<td>%B6</td>
<td></td>
</tr>
<tr>
<td>1645371</td>
<td>548457</td>
<td>182819</td>
<td>1645371</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 43953929 | 15 13420192 | 14 553986 | 13 193381378 | 12 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 548457 | 182819 | 3731 | 1645371 |

| 35978916 | 11 358660781 | 10 271667666 | 9 118784873 | 8 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 182819 | 1645371 | 1645371 | 548457 |

| 337505020 | 7 1389370 | 6 688291 | 5 3378002 | 4 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 1645371 | 11193 | 4459 | 42189 |

| 140671876 | 3 32325724 | 2 8270 | 9741532 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 1645371 | 11193 | 4459 | 42189 |

| 91729 | 19 487915 | 18 4114333 | 17 1276987 | 16 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 705159 | 705159 | 705159 | 235053 |

| 13243117 | 15 16292173 | 14 26536060 | 13 722714 | 12 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 705159 | 705159 | 705159 | 18081 |

| 5382678 | 11 15449995 | 10 14279770 | 9 6603890 | 8 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 100737 | 235053 | 235053 | 100737 |

| 409930 | 7 37340389 | 6 34893715 | 5 26686318 | 4 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 6027 | 705159 | 705159 | 705159 |

| 801511 | 3 17206178 | 2 4406102 | 377534 |
|-------- | %B6 + ------- | %B6 + ------- | %B6 + ------- | %B6 |
| 26117 | 705159 | 705159 | 705159 |

<p>| 1184459 | 19 2335702 | 18 5460230 | 17 79900378 | 16 |</p>
<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th></th>
<th>Result</th>
<th></th>
<th>Result</th>
<th></th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$43953929$</td>
<td>$15$</td>
<td>$13420192$</td>
<td>$14$</td>
<td>$553986$</td>
<td>$13$</td>
<td>$193381378$</td>
<td>$12$</td>
</tr>
<tr>
<td>$548457$</td>
<td>$182819$</td>
<td>$3731$</td>
<td>$1645371$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$35978916$</td>
<td>$11$</td>
<td>$358660781$</td>
<td>$10$</td>
<td>$271667666$</td>
<td>$9$</td>
<td>$118784873$</td>
<td>$8$</td>
</tr>
<tr>
<td>$182819$</td>
<td>$1645371$</td>
<td>$1645371$</td>
<td>$548457$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$337505020$</td>
<td>$7$</td>
<td>$1389370$</td>
<td>$6$</td>
<td>$688291$</td>
<td>$5$</td>
<td>$3378002$</td>
<td>$4$</td>
</tr>
<tr>
<td>$1645371$</td>
<td>$11193$</td>
<td>$4459$</td>
<td>$42189$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$140671876$</td>
<td>$3$</td>
<td>$32325724$</td>
<td>$2$</td>
<td>$8270$</td>
<td>$9741532$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1645371$</td>
<td>$548457$</td>
<td>$343$</td>
<td>$1645371$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$91729$</td>
<td>$19$</td>
<td>$487915$</td>
<td>$18$</td>
<td>$4114333$</td>
<td>$17$</td>
<td>$1276987$</td>
<td>$16$</td>
</tr>
<tr>
<td>$705159$</td>
<td>$705159$</td>
<td>$705159$</td>
<td>$235053$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$13243117$</td>
<td>$15$</td>
<td>$16292173$</td>
<td>$14$</td>
<td>$26536060$</td>
<td>$13$</td>
<td>$722714$</td>
<td>$12$</td>
</tr>
<tr>
<td>$705159$</td>
<td>$705159$</td>
<td>$705159$</td>
<td>$18081$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$5382578$</td>
<td>$11$</td>
<td>$15449995$</td>
<td>$10$</td>
<td>$14279770$</td>
<td>$9$</td>
<td>$6603890$</td>
<td>$8$</td>
</tr>
<tr>
<td>$100737$</td>
<td>$235053$</td>
<td>$235053$</td>
<td>$100737$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$409930$</td>
<td>$7$</td>
<td>$37340389$</td>
<td>$6$</td>
<td>$34993715$</td>
<td>$5$</td>
<td>$26686318$</td>
<td>$4$</td>
</tr>
<tr>
<td>$6027$</td>
<td>$705159$</td>
<td>$705159$</td>
<td>$705159$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$801511$</td>
<td>$3$</td>
<td>$17206178$</td>
<td>$2$</td>
<td>$4406102$</td>
<td>$377534$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$26117$</td>
<td>$705159$</td>
<td>$705159$</td>
<td>$705159$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$1184459$</td>
<td>$19$</td>
<td>$2335702$</td>
<td>$18$</td>
<td>$5460230$</td>
<td>$17$</td>
<td>$79900378$</td>
<td>$16$</td>
</tr>
<tr>
<td>$1645371$</td>
<td>$548457$</td>
<td>$182819$</td>
<td>$1645371$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$43953929$</td>
<td>$15$</td>
<td>$13420192$</td>
<td>$14$</td>
<td>$553986$</td>
<td>$13$</td>
<td>$193381378$</td>
<td>$12$</td>
</tr>
<tr>
<td>$548457$</td>
<td>$182819$</td>
<td>$3731$</td>
<td>$1645371$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$35978916$</td>
<td>$11$</td>
<td>$358660781$</td>
<td>$10$</td>
<td>$271667666$</td>
<td>$9$</td>
<td>$118784873$</td>
<td>$8$</td>
</tr>
</tbody>
</table>
The number of real solutions for the input system is:

```
# lr
8
```

Each of these real solutions is given by a list of elements in RealClosure(R). In these 8 lists, the first element is a value of z, the second of y and the last of x. This is logical since by setting the list of variables of the package to [x,y,z,t] we mean that the elimination ordering on the variables is t < z < y < x. Note that each system treated by the ZDSOLVE package constructor needs only to be zero-dimensional w.r.t. the variables involved in the system itself and not necessarily w.r.t. all the variables used to define the package.

We can approximate these real numbers as follows. This computation
takes between 30 sec. and 5 min, depending on your machine.

\[
\text{[approximate}(r,1/100000) \text{ for } r \text{ in point] for point in lr}]
\]

\[
\begin{align*}
10048059 \\
- \cdots, \quad 2097152 \\
4503057316985387943524397913838966414596731976211768219335881208385516_ \\
314058924567176091423629695777403099833360761048898228916578137094309_ \\
8388797311372052584846939132376157019506760357601165917459486815382098_ \\
78909485152342039281129312614132985656469771145464661495487825919941108_ \\
44704172244049192156726354215820861437758844364634410045253024786661_ \\
923163288214175 \\
/ \\
45030572830252454885165118069855826635083100693753732046528055470686564_ \\
949577509916672018894380940835481793117859862797624551518983570793_ \\
04877442249148870829840324189200301436123314860200821144373790755311_ \\
24363291986489542170422894957129001611949880795702366386443069392027_ \\
1497996826671232335604391523434068924275280417338574817381189277066_ \\
143312396681216 \\
, \\
2106260768823475073789479868604868016596249607148969068553786368093715020639_ \\
680858646960790055889505646893309447007909937802187329095325898785247_ \\
249020715049836604820751566167387245146853390011202964635166381351_ \\
54325568220025035283981086837110614842307020691211297929876896285681_ \\
830479054760056380762664905618462055306047816191780211588703789138988_ \\
1895 \\
/ \\
21062606094998464192472113804816474175341962953296434102413903142368757_ \\
9676685273888555909759652112778862189872891953943640246297375061959812_ \\
326103659799025126863258876567202342106877031710184247484181423288921_ \\
8376812370627084702957062184859288674007719378284992002937659314168_ \\
901000666373896347598118228556731037072026474496776228383762993923280_ \\
0768 \\
] \\
, \\
2563013 \\
[- \cdots, \quad 2097152 \\
- \\
26113461767791927789686791327757719238259963063541781922752330440_ \\
1898996680729283384907686235932074421259259867338115932234504809294_ \\
83752303023733723680666816744617300172721353311571242897 \\
/ \\
116522540050522253058389191600458914375722661020768589900087990314819_ \\
914940922413753983971394019523433320408139928153188829495755455163_
963417619308395977544797140231469234269034921938055593984
,
3572594550275917221096588729615788272998517054675603239678198141006034_
09173528282659062190230446696394197103892330452627332916373757450061_
978982286110976997087250466235373
/
1039648269345598936877071224834026055800814551120170592200522366591759_
40956948644233914102945295026517998960104811875822530205346505131581_
2439017247289173865014702966308864
]
,
1715967
[- -------,
2097152
-
4213093533784303521084839517977082390377261503969586224828998436606_
03065607635937456481377398376603121267822565801436206939519951465_
18222580524697287410022543952491
/
94418141441853744586496920344922405243659747096625366390641960795_
805882585493199840191699917659443264824641135187383583888147867340_
19307857605820364195856822304768
,
7635833347112644222515625244410831225347475669008589338834162172501904_
994376346730876809042845208919919925302105720971453918982731389072591_
4035
/
2624188764086097199784297610478066633934230467895851602278580978503784_
549205788499019640602266966026891580103543567625039018629887141284916_
75648
]
,
437701
[- -------,
2097152
168310690863839588322172332654225913562986313181951031452750161441497_
4734555328150721364868355579646781603507777199075077835213366484533654_
9138623741304759
/
1683106868095213389001709982705913638963077668731226111167785188004907_
42522629863023588781096261414029859736698426488798980377068799998454_
23381649008099328
,
We can also concentrate on the solutions with real (strictly) positive coordinates:

\[
\text{lpr := positiveSolve(lp)$pack}
\]

Type: List List RealClosure Fraction Integer

Thus we have checked that the input system has no solution with strictly positive coordinates.

Let us define another polynomial system (L-3).

\[
f_0 := x^3 + y + z + t - 1
\]

\[
z + y + x + t - 1
\]

Type: Polynomial Integer

\[
f_1 := x + y^3 + z + t - 1
\]

Type: Polynomial Integer
\[ z + y + x + t - 1 \]

Type: Polynomial Integer

\[ f_2 := x + y + z^3 + t - 1 \]

\[ z + y + x + t - 1 \]

Type: Polynomial Integer

\[ f_3 := x + y + z + t^3 - 1 \]

\[ z + y + x + t - 1 \]

Type: Polynomial Integer

\[ l_1 := [f_0, f_1, f_2, f_3] \]

\[ z + y + x + t - 1, z + y + x + t - 1, z + y + x + t - 1, z + y + x + t - 1 \]

Type: List Polynomial Integer

First compute a decomposition into regular chains (i.e. regular triangular sets).

\[ lts := \text{triangSolve}(l_1) \]

\[
\begin{align*}
2 & 3 & 3 \\
\{t + t + 1, & 3 & z - z - t + t, \\
3 & 2 & 3 & 3 \\
(3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z & + & 6 & 3 & 9 & 6 & 3 \\
(3t - 6t + 3)z + t - 3t + 5t - 3t & , & x + y + z \\
16 & 13 & 10 & 7 & 4 & 2 \\
\{t - 6t + 9t + 4t + 15t - 54t + 27, & 15 & 14 & 13 & 12 & 11 \\
14907232t & + 40893984t & - 115013088t & + 22805712t & + 36330336t & + \\
10 & 9 & 8 & 7 \\
162959040t & - 159859440t & - 156802608t & + 117168768t & + \\
6 & 5 & 4 & 3 \\
126282384t & - 129351600t & + 306646992t & + 475302816t & + \\
2 & - 1006837776t & - 237269088t & + 480716208 * \\
z & + \\
54 & 51 & 48 & 46 & 45 & 43 & 42
\end{align*}
\]
48t - 912t + 8232t - 72t - 46848t + 1152t + 186324t + 
+ 40 39 38 37 36 35 
- 3780t - 543144t - 3168t - 21384t + 1175251t + 41184t + 
+ 34 33 32 31 30 
278003t - 1843242t - 301815t - 1440726t + 1912012t + 
+ 29 28 27 26 25 
1442826t + 4696262t - 922481t - 4816188t - 10583524t + 
+ 24 23 22 21 20 
- 208751t + 11472138t + 16762859t - 857663t - 19328175t + 
+ 19 18 17 16 15 
- 18270421t + 4914903t + 22483044t + 12926517t - 8605511t + 
+ 14 13 12 11 10 
- 17455518t - 5014597t + 8108814t + 8465535t + 190542t + 
+ 9 8 7 6 5 4 
4305624t - 2226123t + 661905t + 1169775t + 226260t - 209952t + 
+ 3 
- 141183t + 27216t , 
+ 3 2 2 3 6 3 3 2 
(3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z + 
+ 6 3 9 6 3 
(3t - 6t + 3)z + t - 3t + 5t - 3t , 
+ 3 
x + y + z + t - 1} , 
+ 2 2 2 
{t, z - 1, y - 1, x + y}, {t - 1, z, y - 1, x + y}, {t - 1, z - 1, z y + 1, x}, 
+ 16 13 10 7 4 2 
{t - 6t + 9t + 4t + 15t - 54t + 27, 
+ 29 28 27 26 25 
4907232t + 40893984t - 115013088t - 1730448t - 168139584t + 
+ 24 23 22 21 
738024480t - 195372288t + 315849456t - 2567279232t + 
+ 20 19 18 17 
937147968t + 1026357696t + 4780488240t - 2893767696t + 
+ 16 15 14 13 
- 5617160352t - 3427651728t + 5001100848t + 8720098416t
\[ \begin{align*} &+ \quad 12 \quad 11 \quad 10 \quad 9 \\
&\quad 2331732960t - 499046544t - 16243306272t - 9748123200t \\
&+ \quad 8 \quad 7 \quad 6 \quad 5 \\
&\quad 3927244320t + 25257280896t + 10348032096t - 17128672128t \\
&+ \quad 4 \quad 3 \quad 2 \\
&\quad - 14755488768t + 544086720t + 10848188736t + 1423614528t \\
&+ \quad - 2884297248 \\
&* \\
&z \\
&+ \quad 68 \quad 65 \quad 62 \quad 60 \quad 59 \quad 57 \quad 56 \\
&\quad - 48t + 1152t - 13560t + 360t + 103656t - 7560t - 572820t \\
&+ \quad 54 \quad 53 \quad 52 \quad 51 \quad 50 \quad 49 \\
&\quad 71316t + 2414556t + 2736t - 402876t - 7985131t - 49248t \\
&+ \quad 48 \quad 47 \quad 46 \quad 45 \quad 44 \\
&\quad 1431133t + 20977409t + 521487t - 2697635t - 43763654t \\
&+ \quad 43 \quad 42 \quad 41 \quad 40 \quad 39 \\
&\quad - 3756573t - 2093410t + 71546495t + 19699032t + 35025028t \\
&+ \quad 38 \quad 37 \quad 36 \quad 35 \quad 34 \\
&\quad - 89623786t - 77798760t - 138654191t + 87596128t + 235642497t \\
&+ \quad 33 \quad 32 \quad 31 \quad 30 \quad 29 \\
&\quad 349607642t - 93299834t - 551563167t - 630995176t + 186818962t \\
&+ \quad 28 \quad 27 \quad 26 \quad 25 \\
&\quad 995427468t + 828416204t - 393919231t - 1076617485t \\
&+ \quad 24 \quad 23 \quad 22 \quad 21 \\
&\quad - 1609479791t + 595738126t + 1198787136t + 4342832069t \\
&+ \quad 20 \quad 19 \quad 18 \quad 17 \\
&\quad - 2075938757t - 4390835799t - 4822843033t + 6932747678t \\
&+ \quad 16 \quad 15 \quad 14 \quad 13 \\
&\quad 6172196808t + 1141517740t - 4981677585t - 9819815280t \\
&+ \quad 12 \quad 11 \quad 10 \quad 9 \\
&\quad - 7404299976t - 157295760t + 29124027630t + 14856038208t \\
&+ \quad 8 \quad 7 \quad 6 \quad 5 \\
&\quad - 16184101410t - 26935440354t - 3574164258t + 10271338974t \\
&+ \]
\[
\begin{align*}
&1111425264t + 6869861262t - 9780477840t - 3586674168t + 2884297248 \\
+ &36243269t + (6t - 6)z + (6t - 12t + 3)z + 2t - 6t + t + 3t)y \\
+ &36243269t + (6t - 12t + 6)z + (4t - 12t + 11t - 3)z + t - 4t \\
+ &6t + 3 \\
&5t - 2t \\
+ &x + y + z + t - 1} \\
\{t - 1, z - 1, y, x + z}, \\
\{t + t + t - 2t - 2t - 2t + 19t + 19t - 8, \\
2395770t + 3934440t - 3902067t - 10084164t - 1010448t \\
+ &2 \\
&32386932t + 22413225t - 10432368 \\
* &z \\
+ &765432 \\
&- 463519t + 3586833t + 9494955t - 8539305t - 33283098t \\
+ &2 \\
&35479377t + 46263256t - 17419896 \\
* &y \\
+ &346332 \\
&- (3t - 3)z + (9t - 9)z + (12t - 24t + 9)z + (152t + 219t - 67)z \\
+ &643 \\
&- 41t + 57t + 25t - 57t + 16 \\
* &z \\
+ &123643 \\
&- (3t - 3)z + (9t - 18t + 9)z + (- 181t + 270t - 89)z \\
+ &643 \\
&(- 92t + 135t + 49t - 135t + 43)z + 27t - 27t - 54t + 396t \\
+ &- 486t + 144 \\
+ &3 \\
\end{align*}
\]
\[ \{x + y + z + t - 1\}, \]
\[ \{t, z - t + 1, y - 1, x - 1\}, \{t - 1, z, y, x\}, \{t, z - 1, y, x\}, \{t, z, y - 1, x\}, \{t, z, y, x - 1\}\]

Type: List RegularChain(Integer, [x, y, z, t])

Then we compute a univariate representation.

\[\text{univariateSolve}(lf)\$pack\]

\[
\begin{align*}
\text{univariateSolve}(lf) & \text{pack} \\
\text{complexRoots} &= \text{?, coordinates} = [x - 1, y - 1, z + 1, t - %A], \\
\text{complexRoots} &= \text{?, coordinates} = [x, y - 1, z, t - %A], \\
\text{complexRoots} &= ? - 1, coordinates = [x, y, z, t - %A], \\
\text{complexRoots} &= \text{?, coordinates} = [x, y, z, t - %A], \\
\text{complexRoots} &= ? - 2, coordinates = [x - 1, y, z, t - 1], \\
\text{complexRoots} &= \text{?, coordinates} = [x + 1, y - 1, z, t - 1], \\
\text{complexRoots} &= ? - 1, coordinates = [x - 1, y + 1, z, t], \\
\text{complexRoots} &= \text{?, coordinates} = [x - 1, y, z - 1, t - %A], \\
\text{complexRoots} &= \text{?, coordinates} = [2x + %A + %A - 1, 2y + %A - 1, z - %A, t - %A], \\
\text{complexRoots} &= ? + 3? - 2? + 3? - 3, \\
\text{coordinates} &= [x - %A, y - %A, z + %A + 2%A - 1, t - %A], \\
\text{complexRoots} &= ? - 2? + 3, \\
\text{coordinates} &= [x + %A - %A - 1, y + %A - %A - 1, z - %A + 2%A + 1, t - %A], \\
\text{complexRoots} &= \text{?, coordinates} = [x - 1, y - 1, z, t - %A], \\
\text{complexRoots} &= ? + 1, coordinates = [x - 1, y - 1, z, t - %A], \\
\text{complexRoots} &= ? + 2? + 3? - 3, \\
\text{coordinates} &= [2x - %A - %A - 1, y + %A, 2z - %A - 1, t + %A], \\
\text{coordinates} &= 5, 4, 3, 2
\end{align*}
\]
\[
\begin{align*}
12609x + 23A - 49A - 46A + 362A - 5015A - 8239, \\
5 4 3 2 \\
25218y + 23A - 49A - 46A + 362A + 7594A - 8239, \\
5 4 3 2 \\
25218z + 23A + 49A - 46A + 362A + 7594A - 8239, \\
5 4 3 2 \\
12609t + 23A + 49A - 46A + 362A - 5015A - 8239
\end{align*}
\]

\[
\begin{align*}
\text{complexRoots} &= \sqrt[5]{12? + 12\sqrt[5]{16? + 48\sqrt[5]{96}},} \\
\text{coordinates} &= [8x + \%A + 8\%A - 8, 2y - \%A, 2z - \%A, 2t - \%A] \\
\end{align*}
\]

\[
\begin{align*}
\text{complexRoots} &= \sqrt[5]{12? - 5? - 3? + 9? + 3}, \\
\text{coordinates} &= [2x - \%A + 2\%A - 1, 2y + \%A - 4\%A + 1, 2z - \%A + 2\%A - 1, 2t - \%A + 2\%A - 1] \\
\end{align*}
\]

\[
\begin{align*}
\text{complexRoots} &= \sqrt[5]{12? - 11? + 37}, \\
\text{coordinates} &= [3x - \%A + 7, 6y + \%A + 3\%A - 7, 3z - \%A + 7, 6t + \%A - 3\%A - 7] \\
\end{align*}
\]

\[
\begin{align*}
\text{complexRoots} &= \sqrt[5]{12? + 1}, \\
\text{coordinates} &= [x - 1, y, z - 1, t + 1] \\
\text{complexRoots} &= \sqrt[5]{12? + 2}, \\
\text{coordinates} &= [x, y - 1, z - 1, t + 1] \\
\text{complexRoots} &= \sqrt[5]{12? - 2}, \\
\text{coordinates} &= [x, y - 1, z + 1, t - 1] \\
\text{complexRoots} &= \sqrt[5]{12?}, \\
\text{coordinates} &= [x, y + 1, z - 1, t - 1]
\end{align*}
\]
Note that this computation is made from the input system lf.

However it is possible to reuse a pre-computed regular chain as follows:

```plaintext
ts := lts.1
{t + t + 1, z - z - t + t,
(3z + 3t - 3)y + (3z + (6t - 6)z + 3t - 6t + 3)y + (3t - 3)z +
```
We compute now the full set of points with real coordinates:

```plaintext
lr2 := realSolve(lf)$pack
[[0,-1,1,1], [0,0,1,0], [1,0,0,0], [0,0,0,1], [0,1,0,0], [1,0,%B37,-%B37], [1,0,%B38,-%B38], [0,1,%B35,-%B35], [0,1,%B36,-%B36], [-1,0,1,1], [%B32,
  1 15 2 14 1 13 4 12 11 11 4 10
  -- %B32 + -- %B32 + -- %B32 -- -- %B32 -- -- %B32 -- -- %B32
  27 27 27 27 27 27 27
+ 1 9 14 8 1 7 2 6 1 5 2 4 3
  -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32
  27 27 27 27 27 27 27 27 27 27 27 27
+ 1 9 14 8 1 7 2 6 1 5 2 4 3
  -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32 + -- %B32
  27 27 27 27 27 27 27 27 27 27 27 27 27]
```

Type: List List RealClosure Fraction Integer
4  2
- %B32  - %B32  - 2

3

1  15  1  14  1  13  2  12  11  11  2  10
- -- %B32  -- %B32  -- %B32  + -- %B32  + -- %B32  + -- %B32
54  27  54  27  54  27

+ 1  9  7  8  1  7  1  6  1  5  1  4  3
- -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32
54  27  54  9  6  9

+ 2  2  1  3
- - %B32  + - %B32  + -
3  2  2

1  15  1  14  1  13  2  12  11  11  2  10
- -- %B32  -- %B32  -- %B32  + -- %B32  + -- %B32  + -- %B32
54  27  54  27  54  27

+ 1  9  7  8  1  7  1  6  1  5  1  4  3
- -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32  -- %B32
54  27  54  9  6  9

+ 2  2  1  3
- - %B32  + - %B32  + -
3  2  2

] [%B33,

1  15  2  14  1  13  4  12  11  11  4  10
-- %B33  + -- %B33  + -- %B33  -- %B33  -- %B33  -- %B33
27  27  27  27  27  27

+ 1  9  14  8  1  7  2  6  1  5  2  4  3
-- %B33  + -- %B33  + -- %B33  + -- %B33  + -- %B33  + -- %B33
27  27  27  27  3  9

+ 4  2
- %B33  - %B33  - 2

3

1  15  1  14  1  13  2  12  11  11  2  10
- -- %B33  -- %B33  -- %B33  + -- %B33  + -- %B33  + -- %B33
54  27  54  27  54  27

+ 1  9  7  8  1  7  1  6  1  5  1  4  3
- -- %B33  -- %B33  -- %B33  -- %B33  -- %B33  -- %B33  -- %B33  -- %B33
54  27  54  9  6  9

+
```
\begin{align*}
\text{%B30}, & \quad 1 \quad 15 \quad 1 \quad 14 \quad 1 \quad 13 \quad 2 \quad 12 \quad 11 \quad 11 \quad 2 \quad 10 \\
& \quad - \quad %B30 + -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23 \\
& \quad 54 \quad 27 \quad 54 \quad 27 \quad 54 \quad 27 \quad 54 \quad 27 \\
+ & \quad 2 \quad 10 \quad 1 \quad 9 \quad 7 \quad 8 \quad 1 \quad 7 \quad 1 \quad 6 \quad 1 \quad 5 \\
& \quad - \quad -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23 - -- %B23 + -- %B23 \\
& \quad 27 \quad 54 \quad 27 \quad 54 \quad 9 \quad 6 \quad 9 \\
\end{align*}
```

```
\begin{align*}
\text{%B31}, & \quad 1 \quad 15 \quad 1 \quad 14 \quad 1 \quad 13 \quad 2 \quad 12 \quad 11 \quad 11 \quad 2 \quad 10 \\
& \quad - \quad %B31 + -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23 + -- %B23 \\
& \quad 54 \quad 27 \quad 54 \quad 27 \quad 54 \quad 27 \quad 54 \quad 27 \\
\end{align*}
```
| + 2 10 1 9 7 8 1 7 1 6 1 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| - -- | %B23 | + -- | %B23 | + -- | %B23 | + -- | %B23 | + -- | %B23 | + -- | %B23 |
| 27 | 54 | 27 | 54 | 9 | 6 |
| + 1 4 2 2 1 1 |
| - %B23 | + - %B23 | + - %B23 | - |
| 9 | 3 | 2 | 2 |

[ %B24, |
| 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| --- | %B24 | --- | %B24 | --- | %B24 | --- | %B24 | + -- | %B24 | + -- | %B24 |
| 54 | 27 | 54 | 27 | 54 | 27 |
| + 1 9 7 8 1 7 1 6 1 5 1 4 3 |
| - -- | %B24 | --- | %B24 | --- | %B24 | --- | %B24 | --- | %B24 | - %B24 | - %B24 |
| 54 | 27 | 54 | 9 | 6 | 9 |
| + 2 2 1 3 |
| - - | %B24 | + - | %B24 | + - |
| 3 | 2 | 2 |

[ %B28, |
| 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| %B28 | + -- | %B24 | + -- | %B24 | + -- | %B24 | + -- | %B24 | + -- | %B24 |
| 54 | 27 | 54 | 27 | 54 |
| + 2 10 1 9 7 8 1 7 1 6 1 5 |
| --- | %B24 | + -- | %B24 | + -- | %B24 | --- | %B24 | + - | %B24 | + - | %B24 |
| 27 | 54 | 27 | 54 | 9 | 6 |
| + 1 4 2 2 1 1 |
| - %B24 | + - | %B24 | + - | %B24 | - |
| 9 | 3 | 2 | 2 |

[ %B24, |
| 1 | 15 | 1 | 14 | 1 | 13 | 2 | 12 | 11 | 11 | 2 | 10 |
| --- | %B24 | --- | %B24 | --- | %B24 | --- | %B24 | + -- | %B24 | + -- | %B24 |
| 54 | 27 | 54 | 27 | 54 |
| + 1 9 7 8 1 7 1 6 1 5 1 4 3 |
| --- | %B24 | --- | %B24 | --- | %B24 | --- | %B24 | + - | %B24 | + - | %B24 |
| 54 | 27 | 54 | 9 | 6 | 9 |
| + 2 2 1 3 |
| - | %B24 | + - | %B24 | + - |
| 3 | 2 | 2 |
%B29,
1 15 1 14 1 13 2 12 11 11
- %B29 + -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24
54 27 54 27 54 27 54
+
2 10 1 9 7 8 1 7 1 6 1 5
- -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24 + -- %B24
27 54 27 54 9 6
+
1 4 2 2 1 1
- %B24 + - %B24 - - %B24 - -
9 3 2 2
]

, %B25,
1 15 1 14 1 13 2 12 11 11 2 10
- -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25
54 27 54 27 54 27
+
1 9 7 8 1 7 1 6 1 5 1 4 3
- -- %B25 + -- %B25 + -- %B25 - - %B25 - - %B25 - - %B25 - - %B25
54 27 54 9 6 9
+
2 2 1 3
- %B25 + - %B25 + -
3 2 2

, %B26,
1 15 1 14 1 13 2 12 11 11
- %B26 + -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25
54 27 54 27 54 27 54
+
2 10 1 9 7 8 1 7 1 6 1 5
- -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25
27 54 27 54 9 6
+
1 4 2 2 1 1
- %B25 + - %B25 - - %B25 - -
9 3 2 2
]

, %B25,
1 15 1 14 1 13 2 12 11 11 2 10
- -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25 + -- %B25
54 27 54 27 54 27 54
+
1 9 7 8 1 7 1 6 1 5 1 4 3
- -- %B25 + -- %B25 + -- %B25 - - %B25 - - %B25 - - %B25 - %B25
54 27 54 9 6 9
+

The number of real solutions for the input system is:

```
#lr2
27
Type: PositiveInteger
```

Another example of computation of real solutions illustrates the `LexTriangularPackage` package constructor.

We concentrate now on the solutions with real (strictly) positive coordinates:

```
lpr2 := positiveSolve(lf)$pack
[[%B40,- %B40 + -, - - %B40 + -, - %B40 + -],
 3 3 3 3 3 3 3
Type: List List RealClosure Fraction Integer
```

Finally, we approximate the coordinates of this point with 20 exact digits:

```
[approximate(r,1/10**21)::Float for r in lpr2.1]
[0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291, 0.3221853546 2608559291]
```
ZeroDimensionalSolvePackage (ZDSOLVE)

Exports:
convert positiveSolve realSolve squareFree triangSolve univariateSolve
++ The lexTriangular algorithm can also be used for computing these
++ decompositions (see \spadtype{LexTriangularPackage} package constructor).
++ For that purpose, the operations univariateSolve, realSolve and
++ positiveSolve admit an optional argument.

ZeroDimensionalSolvePackage(R,ls,ls2): Exports == Implementation where
R : Join(OrderedRing,EuclideanDomain,CharacteristicZero,RealConstant)
ls: List Symbol
ls2: List Symbol
V ==> OrderedVariableList(ls)
N ==> NonNegativeInteger
Z ==> Integer
B ==> Boolean
P ==> Polynomial R
LP ==> List P
LS ==> List Symbol
Q ==> NewSparseMultivariatePolynomial(R,V)
U ==> SparseUnivariatePolynomial(R)
TS ==> RegularChain(R,ls)
RUR ==> Record(complexRoots: U, coordinates: LP)
K ==> Fraction R
RC ==> RealClosure(K)
PRC ==> Polynomial RC
REALSOL ==> List RC
URC ==> SparseUnivariatePolynomial RC
V2 ==> OrderedVariableList(ls2)
Q2 ==> NewSparseMultivariatePolynomial(R,V2)
E2 ==> IndexedExponents V2
ST ==> SquareFreeRegularTriangularSet(R,E2,V2,Q2)
Q2WT ==> Record(val: Q2, tower: ST)
LQ2WT ==> Record(val: List(Q2), tower: ST)
WIP ==> Record(reals: List(RC), vars: List(Symbol), pols: List(Q2))
polsetpack ==> PolynomialSetUtilitiesPackage(R,E2,V2,Q2)
normpack ==> NormalizationPackage(R,E2,V2,Q2,ST)
rurpack ==> InternalRationalUnivariateRepresentationPackage(R,E2,V2,Q2,ST)
quasicomppack ==> SquareFreeQuasiComponentPackage(R,E2,V2,Q2,ST)
lextripack ==> LexTriangularPackage(R,ls)

Exports == with
  triangSolve: (LP,B,B) -> List RegularChain(R,ls)
    ++ \spad{triangSolve(lp,info?,lextri?)} decomposes the variety
    ++ associated with \axiom{lp} into regular chains.
    ++ Thus a point belongs to this variety iff it is a regular
    ++ zero of a regular set in in the output.
    ++ Note that \axiom{lp} needs to generate a zero-dimensional ideal.
    ++ If \axiom{lp} is not zero-dimensional then the result is only
    ++ a decomposition of its zero-set in the sense of the closure
    ++ (w.r.t. Zarisky topology).
    ++ Moreover, if \spad{info?} is \spad{true} then some information is
    ++ displayed during the computations.
++ See zeroSetSplit from RegularTriangularSetCategory(lp,true,info?).
++ If \spad{lextri?} is \spad{true} then the lexTriangular algorithm
++ is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
++ Otherwise, the triangular decomposition is computed directly from
++ the input
++ system by using the zeroSetSplit from RegularChain
triangSolve: (LP,B) -> List RegularChain(R,ls)
++ \spad{triangSolve(lp,info?)} returns the same as
++ \spad{triangSolve(lp,false)}
triangSolve: LP -> List RegularChain(R,ls)
++ \spad{triangSolve(lp)} returns the same as
++ \spad{triangSolve(lp,false,false)}
univariateSolve: RegularChain(R,ls) -> _
++ \spad{univariateSolve(ts)} returns a univariate representation
++ of \spad{ts}.
++ See rur from RationalUnivariateRepresentationPackage(lp,true).
univariateSolve: (LP,B,B,B) -> List RUR
++ \spad{univariateSolve(lp,info?,check?,lextri?)} returns a univariate
++ representation of the variety associated with \spad{lp}.
++ Moreover, if \spad{info?} is \spad{true} then some information is
++ displayed during the decomposition into regular chains.
++ If \spad{check?} is \spad{true} then the result is checked.
++ See rur from RationalUnivariateRepresentationPackage(lp,true).
++ If \spad{lextri?} is \spad{true} then the lexTriangular
++ algorithm is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see zeroSetSplit from LexTriangularPackage(lp,false)).
++ Otherwise, the triangular decomposition is computed directly
++ from the input
++ system by using the zeroSetSplit from RegularChain
univariateSolve: (LP,B,B) -> List RUR
++ \spad{univariateSolve(lp,info?,check?)} returns the same as
++ \spad{univariateSolve(lp,info?,false,false)}.
univariateSolve: (LP,B) -> List RUR
++ \spad{univariateSolve(lp,info?)} returns the same as
++ \spad{univariateSolve(lp,info?,false,false)}.
univariateSolve: LP -> List RUR
++ \spad{univariateSolve(lp)} returns the same as
++ \spad{univariateSolve(lp,false,false,false)}.
realSolve: RegularChain(R,ls) -> List REALSOL
++ \spad{realSolve(ts)} returns the set of the points in the regular
++ zero set of \spad{ts} whose coordinates are all real.
++ WARNING. For each set of coordinates given by \spad{realSolve(ts)}
++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.
realSolve: RegularChain(R,ls) -> List REALSOL
++ \spad{realSolve(ts,info?,check?,lextri?)} returns the set of the
++ points in the variety associated with \spad{lp} whose coordinates
++ are all real.
++ Moreover, if \spad{info?} is \spad{true} then some information is
++ displayed during decomposition into regular chains.
++ If \spad{check?} is \spad{true} then the result is checked.
++ If \spad{lextri?} is \spad{true} then the \text{lexTriangular} algorithm
++ is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see \spad{zeroSetSplit} from \spadtype{LexTriangularPackage(lp,false)}).
++ Otherwise, the triangular decomposition is computed directly from
++ the input
++ system by using the \spad{zeroSetSplit} from \spadtype{RegularChain}.
++ WARNING. For each set of coordinates given by
++ \spad{realSolve(ts,info?,check?,lextri?)}
++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.

realSolve: \( (LP,B,B) \rightarrow \text{List REALSOL} \)
++ \spad{realSolve(ts,info?,check?)} returns the same as
++ \spad{realSolve(ts,info?,check?,false)}.
realSolve: \( (LP,B) \rightarrow \text{List REALSOL} \)
++ \spad{realSolve(ts,info?)} returns the same as
++ \spad{realSolve(ts,info?,false,false)}.
realSolve: \( LP \rightarrow \text{List REALSOL} \)
++ \spad{realSolve(lp)} returns the same as
++ \spad{realSolve(ts,false,false,false)}.
positiveSolve: \( \text{RegularChain}(R,ls) \rightarrow \text{List REALSOL} \)
++ \spad{positiveSolve(ts)} returns the points of the regular
++ set of \spad{ts} with (real) strictly positive coordinates.
positiveSolve: \( (LP,B,B) \rightarrow \text{List REALSOL} \)
++ \spad{positiveSolve(lp,info?,lextri?)} returns the set of the points
++ in the variety associated with \spad{lp} whose coordinates are
++ (real) strictly positive.
++ Moreover, if \spad{info?} is \spad{true} then some information is
++ displayed during decomposition into regular chains.
++ If \spad{lextri?} is \spad{true} then the \text{lexTriangular} algorithm is called
++ from the \spadtype{LexTriangularPackage} constructor
++ (see \spad{zeroSetSplit} from \spadtype{LexTriangularPackage(lp,false)}).
++ Otherwise, the triangular decomposition is computed directly from
++ the input
++ system by using the \spad{zeroSetSplit} from \spadtype{RegularChain}.
++ WARNING. For each set of coordinates given by
++ \spad{positiveSolve(lp,info?,lextri?)}
++ the ordering of the indeterminates is reversed w.r.t. \spad{ls}.

positiveSolve: \( (LP,B) \rightarrow \text{List REALSOL} \)
++ \spad{positiveSolve(lp)} returns the same as
++ \spad{positiveSolve(lp,info?,false)}.
positiveSolve: \( LP \rightarrow \text{List REALSOL} \)
++ \spad{positiveSolve(lp)} returns the same as
++ \spad{positiveSolve(lp,false,false)}.
squareFree: \( TS \rightarrow \text{List ST} \)
++ \spad{squareFree(ts)} returns the square-free factorization
++ of \spad{ts}. Moreover, each factor is a Lazard triangular set
++ and the decomposition
++ is a Kalkbrener split of \spad{ts}, which is enough here for
++ the matter of solving zero-dimensional algebraic systems.
++ WARNING. \spad{ts} is not checked to be zero-dimensional.

convert: Q -> Q2
++ \spad{convert(q)} converts \spad{q}.

convert: P -> PRC
++ \spad{convert(p)} converts \spad{p}.

convert: Q2 -> PRC
++ \spad{convert(q)} converts \spad{q}.

convert: U -> URC
++ \spad{convert(u)} converts \spad{u}.

convert: ST -> List Q2
++ \spad{convert(st)} returns the members of \spad{st}.

Implementation == add
news: Symbol := last(ls2)$(List Symbol)
newv: V2 := (variable(news)$V2)::V2
newq: Q2 := newv :: Q2

convert(q:Q):Q2 ==
  ground? q => (ground(q))::Q2
  q2: Q2 := 0
  while not ground?(q) repeat
    v: V := mvar(q)
    d: N := mdeg(q)
    v2: V2 := (variable(convert(v)@Symbol)$V2)::V2
    iq2: Q2 := convert(init(q))@Q2
    lq2: Q2 := (v2 :: Q2)
    lq2 := lq2 ** d
    q2 := iq2 * lq2 + q2
    q := tail(q)
  q2 + (ground(q))::Q2

squareFree(ts:TS):List(ST) ==
  irredu?: Boolean := false
  st: ST := [[newq]$(List Q2)]
  lq: List(Q2) := [convert(p)@Q2 for p in parts(ts)]
  lq := sort(infRittWu?,lq)
  toSee: List LQ2WT := []
  if irredu?
    then
      lf := irreducibleFactors([first lq])$polsetpack
      lq := rest lq
      for f in lf repeat
        toSee := cons([cons(f,lq),st]$LQ2WT, toSee)
    else
      toSee := [[lq, st]$LQ2WT]
  toSave: List ST := []
while not empty? toSee repeat
  lqwt := first toSee; toSee := rest toSee
  lq := lqwt.val; st := lqwt.tower
  empty? lq =>
  toSave := cons(st,toSave)
  q := first lq; lq := rest lq
  lsfqwt: List Q2WT := squareFreePart(q,st)$ST
  for sfqwt in lsfqwt repeat
    q := sfqwt.val; st := sfqwt.tower
    if not ground? init(q) then
      q := normalizedAssociate(q,st)$normpack
      newts := internalAugment(q,st)$ST
      newlq := [remainder(q,newts).polnum for q in lq]
      toSee := cons([newlq,newts]$LQ2WT,toSee)
  toSave

triangSolve(lp: LP, info?: B, lextri?: B): List TS ==
  lq: List(Q) := [convert(p)$Q for p in lp]
  lextri? => zeroSetSplit(lq,false)$lextripack
  zeroSetSplit(lq,true,info?)$TS

triangSolve(lp: LP, info?: B): List TS == triangSolve(lp,info?,false)

triangSolve(lp: LP): List TS == triangSolve(lp,false)

convert(u: U): URC ==
  zero? u => 0
  ground? u => ((ground(u) :: K)::RC)::URC
  uu: URC := 0
  while not ground? u repeat
    uu := monomial((leadingCoefficient(u) :: K):: RC,degree(u)) + uu
    u := reductum u
  uu + ((ground(u) :: K)::RC)::URC

coerceFromRtoRC(r:R): RC ==
  (r::K)::RC

convert(p:P): PRC ==
  map(coerceFromRtoRC,p)$PolynomialFunctions2(R,RC)

convert(q2:Q2): PRC ==
  p: P := coerce(q2)$Q2
  convert(p)$PRC

convert(sts:ST): List Q2 ==
  lq2: List(Q2) := parts(sts)$ST
  lq2 := sort(infRittWu?,lq2)
  rest(lq2)
realSolve(ts: TS): List REALSOL ==
  lsts: List ST := squareFree(ts)
  lr: REALSOL := []
  lv: List Symbol := []
  toSee := [(lr,lv,convert(sts)@(List Q2))$WIP for sts in lsts]
  toSave: List REALSOL := []
  while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
    lr := wip.reals; lv := wip.vars; lq2 := wip.pols
    (empty? lq2) and (not empty? lr) =>
      toSave := cons(reverse(lr),toSave)
    q2 := first lq2; lq2 := rest lq2
    qrc := convert(q2)$PRC
    if not empty? lr
      then
        for r in reverse(lr) for v in reverse(lv) repeat
          qrc := eval(qrc,v,r)
          lv := cons((mainVariable(qrc) :: Symbol),lv)
        urc: URC := univariate(qrc)$URC
        urcRoots := allRootsOf(urc)$RC
        for urcRoot in urcRoots repeat
          toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
      toSave

realSolve(lp: List(P), info?:Boolean, check?:Boolean, lextri?: Boolean): List REALSOL ==
  lts: List TS
  lq: List(Q) := [convert(p)$Q for p in lp]
  if lextri?
    then
      lts := zeroSetSplit(lq,false)$lextripack
    else
      lts := zeroSetSplit(lq,true,info?)$TS
  lsts: List ST := []
  for ts in lts repeat
    lsts := concat(squareFree(ts), lsts)
  lsts := removeSuperfluousQuasiComponents(lsts)$quasicomppack
  lr: REALSOL := []
  lv: List Symbol := []
  toSee := [(lr,lv,convert(sts)@(List Q2))$WIP for sts in lsts]
  toSave: List REALSOL := []
  while not empty? toSee repeat
    wip := first toSee; toSee := rest toSee
    lr := wip.reals; lv := wip.vars; lq2 := wip.pols
    (empty? lq2) and (not empty? lr) =>
      toSave := cons(reverse(lr),toSave)
    q2 := first lq2; lq2 := rest lq2
    qrc := convert(q2)$PRC
    if not empty? lr
      then
for r in reverse(lr) for v in reverse(lv) repeat
    qrc := eval(qrc,v,r)
lv := cons((mainVariable(qrc) :: Symbol),lv)
urc: URC := univariate(qrc)@URC
urcRoots := allRootsOf(urc)$RC
for urcRoot in urcRoots repeat
    toSee := cons([cons(urcRoot,lr),lv,lq2]$WIP, toSee)
if check?
    then
        for p in lp repeat
            for realsol in toSave repeat
                prc: PRC := convert(p)@PRC
                for rr in realsol for symb in reverse(ls) repeat
                    prc := eval(prc,symb,rr)
                    not zero? prc =>
                        error "realSolve$ZDSOLVE: bad result"
toSave

realSolve(lp: List(P), info?:Boolean, check?:Boolean): List REALSOL ==
    realSolve(lp,info?,check?,false)
realSolve(lp: List(P), info?:Boolean): List REALSOL ==
    realSolve(lp,info?,false,false)
realSolve(lp: List(P)): List REALSOL ==
    realSolve(lp,false,false,false)
positiveSolve(ts: TS): List REALSOL ==
    lsts: List ST := squareFree(ts)
    lr: REALSOL := []
    lv: List Symbol := []
    toSee := [[lr,lv,convert(sts)@(List Q2)]$WIP for sts in lsts]
    toSave: List REALSOL := []
    while not empty? toSee repeat
        wip := first toSee; toSee := rest toSee
        lr := wip.reals; lv := wip.vars; lq2 := wip.pols
        (empty? lq2) and (not empty? lr) =>
            toSave := cons(reverse(lr),toSave)
        q2 := first lq2; lq2 := rest lq2
        qrc := convert(q2)@PRC
        if not empty? lr
            then
                for r in reverse(lr) for v in reverse(lv) repeat
                    qrc := eval(qrc,v,r)
            lv := cons((mainVariable(qrc) :: Symbol),lv)
        urc: URC := univariate(qrc)@URC
        urcRoots := allRootsOf(urc)$RC
        for urcRoot in urcRoots repeat
            if positive? urcRoot
                then
positiveSolve(lp: List(P), info?: Boolean, lextri?: Boolean): List REALSOL ==
    lts: List TS
    lq: List(Q) := [convert(p)$Q for p in lp]
    if lextri?
        then
            lts := zeroSetSplit(lq,false)$lestripack
        else
            lts := zeroSetSplit(lq,true,info?)$TS
    lsts: List ST := []
    for ts in lts repeat
        lsts := concat(squareFree(ts), lsts)
    lsts := removeSuperfluousQuasiComponents(lsts)$quasicomppack
    lr: REALSOL := []
    lv: List Symbol := []
    toSee := [[lr,lv,convert(sts)@(List Q2)$WIP for sts in lsts]
    toSave: List REALSOL := []
    while not empty? toSee repeat
        wip := first toSee; toSee := rest toSee
        lr := wip.reals; lv := wip.vars; lq2 := wip.pols
        (empty? lq2) and (not empty? lr) =>
            toSave := cons(reverse(lr),toSave)
        q2 := first lq2; lq2 := rest lq2
        qrc := convert(q2)$PRC
        if not empty? lr
            then
                for r in reverse(lr) for v in reverse(lv) repeat
                    qrc := eval(qrc,v,r)
                    lv := cons((mainVariable(qrc) :: Symbol),lv)
        urc: URC := univariate(qrc)$URC
        urcRoots := allRootsOf(urc)$RC
        for urcRoot in urcRoots repeat
            if positive? urcRoot
                then
                    toSee := cons([[urcRoot,lr],lv,lq2]$WIP, toSee)
    toSave

positiveSolve(lp: List(P), info?: Boolean): List REALSOL ==
    positiveSolve(lp, info?, false)

positiveSolve(lp: List(P)): List REALSOL ==
    positiveSolve(lp, false, false)

univariateSolve(ts: TS): List RUR ==
    toSee: List ST := squareFree(ts)
    toSave: List RUR := []
    for st in toSee repeat
        lus: List ST := rur(st,true)$rurpack
for us in lus repeat
  g: U := univariate(select(us,newv)::Q2)$Q2
  lc: LP := [convert(q2)@P for q2 in parts(collectUpper(us,newv)$ST)$ST]
  toSave := cons([g,lc]$RUR, toSave)
  toSave

univariateSolve(lp: List(P), info?:Boolean, check?:Boolean, lextri?: Boolean): List RUR ==
  lts: List TS
  lq: List(Q) := [convert(p)$Q for p in lp]
  if lextri?
    then
      lts := zeroSetSplit(lq,false)$lextripack
    else
      lts := zeroSetSplit(lq,true,info?)$TS
  toSee: List ST := []
  for ts in lts repeat
    toSee := concat(squareFree(ts), toSee)
  toSee := removeSuperfluousQuasiComponents(toSee)$quasicomppack
  toSave: List RUR := []
  if check?
    then
      lq2: List(Q2) := [convert(p)$Q2 for p in lp]
      for st in toSee repeat
        lus: List ST := rur(st,true)$rurpack
        for us in lus repeat
          if check?
            then
              rems: List(Q2) := [removeZero(q2,us)$ST for q2 in lq2]
              not every?(zero?,rems) =>
                output(st::OutputForm)$OutputPackage
                output("Has a bad RUR component:")$OutputPackage
                output(us::OutputForm)$OutputPackage
                error "univariateSolve$ZDSOLVE: bad RUR"
    g: U := univariate(select(us,newv)::Q2)$Q2
    lc: LP := _
      [convert(q2)@P for q2 in parts(collectUpper(us,newv)$ST)$ST]
  toSave := cons([g,lc]$RUR, toSave)
  toSave

univariateSolve(lp: List(P), info?:Boolean, check?:Boolean): List RUR ==
  univariateSolve(lp,info?,check?,false)

univariateSolve(lp: List(P), info?:Boolean): List RUR ==
  univariateSolve(lp,info?,false,false)

univariateSolve(lp: List(P)): List RUR ==
  univariateSolve(lp,false,false,false)
— ZDSOLVE.dotabb —

"ZDSOLVE" [color="#FF4488",href="bookvol10.4.pdf#nameddest=ZDSOLVE"]
"SFRTCAT" [color="#4488FF",href="bookvol10.2.pdf#nameddest=SFRTCAT"]
"ZDSOLVE" -> "SFRTCAT"
Chapter 28

Chunk collections

— algebra —

\getchunk{package AFALGRO AffineAlgebraicSetComputeWithGroebnerBasis}
\getchunk{package AFALGRES AffineAlgebraicSetComputeWithResultant}
\getchunk{package AF AlgebraicFunction}
\getchunk{package INTEGRAL AlgebraicHermiteIntegration}
\getchunk{package INTALG AlgebraicIntegrate}
\getchunk{package INTAF AlgebraicIntegration}
\getchunk{package ALGMANIP AlgebraicManipulations}
\getchunk{package ALGMFACT AlgebraicMultFact}
\getchunk{package ALGPKG AlgebraPackage}
\getchunk{package ALGFECT AlgFactor}
\getchunk{package INTPACK AnnaNumericalIntegrationPackage}
\getchunk{package OPTPACK AnnaNumericalOptimizationPackage}
\getchunk{package ODEPACK AnnaPartialDifferentialEquationPackage}
\getchunk{package ANY1 AnyFunctions1}
\getchunk{package APPRULE ApplyRules}
\getchunk{package APPLYORE ApplyUnivariateSkewPolynomial}
\getchunk{package ASSOCEQ AssociatedEquations}
\getchunk{package PMPRED AttachPredicates}
\getchunk{package AXXSERV AxiomServer}

\getchunk{package BALFACT BalancedFactorisation}
\getchunk{package BOP1 BasicOperatorFunctions1}
\getchunk{package BEZOUT BezoutMatrix}
\getchunk{package BLUPPACK BlowUpPackage}
\getchunk{package BOUNDZRO BoundIntegerRoots}
\getchunk{package BRILL BrillhartTests}

\getchunk{package CARTEN2 CartesianTensorFunctions2}
CHAPTER 28. CHUNK COLLECTIONS

\getchunk{package CHVAR ChangeOfVariable}
\getchunk{package CPIMA CharacteristicPolynomialInMonogenousAlgebra}
\getchunk{package CHARPOL CharacteristicPolynomialPackage}
\getchunk{package IBACHIN ChineseRemainderToolsForIntegralBases}
\getchunk{package CVMP CoerceVectorMatrixPackage}
\getchunk{package COMBF CombinatorialFunction}
\getchunk{package CDEN CommonDenominator}
\getchunk{package COMMONOP CommonOperators}
\getchunk{package CMMUPC CommuteUnivariatePolynomialCategory}
\getchunk{package COMPFCT ComplexFactorization}
\getchunk{package COMPLEX2 ComplexFunctions2}
\getchunk{package CINTSLE ComplexIntegerSolveLinearPolynomialEquation}
\getchunk{package COMPLAT ComplexPattern}
\getchunk{package CPMATCH ComplexPatternMatch}
\getchunk{package CRFP ComplexRootFindingPackage}
\getchunk{package CMPLXRT ComplexRootPackage}
\getchunk{package CTRIGMNP ComplexTrigonometricManipulations}
\getchunk{package ODECONST ConstantLODE}
\getchunk{package COORDSYS CoordinateSystems}
\getchunk{package CRAPACK CRApackage}
\getchunk{package CYCLES CycleIndicators}
\getchunk{package CSTTOOLS CyclicStreamTools}
\getchunk{package CYCLOTOM CyclotomicPolynomialPackage}
\getchunk{package CAD CylindricalAlgebraicDecompositionPackage}
\getchunk{package CADU CylindricalAlgebraicDecompositionUtilities}
\getchunk{package DFINTTLS DefiniteIntegrationTools}
\getchunk{package DEGRED DegreeReductionPackage}
\getchunk{package DTP DesingTreePackage}
\getchunk{package DIOSP DiophantineSolutionPackage}
\getchunk{package DIRPROD2 DirectProductFunctions2}
\getchunk{package DLP DiscreteLogarithmPackage}
\getchunk{package DISPLAY DisplayPackage}
\getchunk{package DDFACT DistinctDegreeFactorize}
\getchunk{package DFSFUN DoubleFloatSpecialFunctions}
\getchunk{package DBLRESP DoubleResultantPackage}
\getchunk{package DRAWCX DrawComplex}
\getchunk{package DRAWHACK DrawNumericHack}
\getchunk{package DROPTO DrawOptionFunctions0}
\getchunk{package DROPT1 DrawOptionFunctions1}
\getchunk{package DO1AGNT d01AgentsPackage}
\getchunk{package DO1WGETS d01WeightsPackage}
\getchunk{package DO2AGNT d02AgentsPackage}
\getchunk{package DO3AGNT d03AgentsPackage}
\getchunk{package EP EigenPackage}
\getchunk{package EP ElementaryFunction}
\getchunk{package DEFINEF ElementaryFunctionDefiniteIntegration}
\getchunk{package LODEEF ElementaryFunctionLODESolver}
\getchunk{package ODEEF ElementaryFunctionODESolver}
\getchunk{package SIGNEF ElementaryFunctionSign}
\getchunk{package EFSTRUC ElementaryFunctionStructurePackage}
\getchunk{package INTEF ElementaryIntegration}
\getchunk{package RDEEF ElementaryRischDE}
\getchunk{package RDEEFS ElementaryRischDESystem}
\getchunk{package ELFUTS EllipticFunctionsUnivariateTaylorSeries}
\getchunk{package EQ2 EquationFunctions2}
\getchunk{package ERROR ErrorFunctions}
\getchunk{package GBEUCLID EuclideanGroebnerBasisPackage}
\getchunk{package EVALCYC EvaluateCycleIndicators}
\getchunk{package ESCONT ExpertSystemContinuityPackage}
\getchunk{package ESCONT1 ExpertSystemContinuityPackage1}
\getchunk{package ESTOOLS ExpertSystemToolsPackage}
\getchunk{package ESTOOLS1 ExpertSystemToolsPackage1}
\getchunk{package ESTOOLS2 ExpertSystemToolsPackage2}
\getchunk{package EXPR2 ExpressionFunctions2}
\getchunk{package EXPSQL ExpressionSolve}
\getchunk{package ES1 ExpressionSpaceFunctions1}
\getchunk{package ES2 ExpressionSpaceFunctions2}
\getchunk{package EXPRODE ExpressionSpaceODESolver}
\getchunk{package OMEXPR ExpressionToOpenMath}
\getchunk{package EXPR2UPS ExpressionToUnivariatePowerSeries}
\getchunk{package EXPRTUBE ExpressionTubePlot}
\getchunk{package EXP3D Export3D}
\getchunk{package E04AGNT e04AgentsPackage}
\getchunk{package FACTFUNC FactoredFunctions}
\getchunk{package FR2 FactoredFunctions2}
\getchunk{package FRUTIL FactoredFunctionUtilities}
\getchunk{package FACUTIL FactoringUtilities}
\getchunk{package FACTEXT FactorisationOverPseudoAlgebraicClosureOfAlgExtOfRationalNumber}
\getchunk{package FACTRN FactorisationOverPseudoAlgebraicClosureOfRationalNumber}
\getchunk{package FGLMICPK FGLMIfCanPackage}
\getchunk{package FORDER FindOrderFinite}
\getchunk{package FAMR2 FiniteAbelianMonoidRingFunctions2}
\getchunk{package FDIV2 FiniteDivisorFunctions2}
\getchunk{package FFACTOR FiniteFieldFactorization}
\getchunk{package FFACTISE FiniteFieldFactorizationWithSizeParseBySideEffect}
\getchunk{package FFF FiniteFieldFunctions}
\getchunk{package FFHOM FiniteFieldHomomorphisms}
\getchunk{package FFPOLY FiniteFieldPolynomialPackage}
\getchunk{package FFPOLY2 FiniteFieldPolynomialPackage2}
\getchunk{package FFSLPE FiniteFieldSolveLinearPolynomialEquation}
\getchunk{package FFSQFR FiniteFieldSquareFreeDecomposition}
\getchunk{package FLAGG2 FiniteLinearAggregateFunctions2}
\getchunk{package FLASORT FiniteLinearAggregateSort}
\getchunk{package FSAGG2 FiniteSetAggregateFunctions2}
\getchunk{package FLOATCP FloatingComplexPackage}
\getchunk{package FLOATRP FloatingRealPackage}
\getchunk{package FCPAK1 FortranCodePackage1}
CHAPTER 28. CHUNK COLLECTIONS

\getchunk{package FOP FortranOutputStackPackage}
\getchunk{package FORT FortranPackage}
\getchunk{package FRIDEB2 FractionalIdealFunctions2}
\getchunk{package FFGF FractionFreeFastGaussianFractions}
\getchunk{package FRACF FractionFunctions2}
\getchunk{package FRNAAF2 FramedNonAssociativeAlgebraFunctions2}
\getchunk{package FSPECF FunctionalSpecialFunction}
\getchunk{package FFCAT2 FunctionFieldCategoryFunctions2}
\getchunk{package FFINT2 FunctionFieldIntegralBasis}
\getchunk{package PMASSFS FunctionSpaceAssertions}
\getchunk{package PMPREDFS FunctionSpaceAttachPredicates}
\getchunk{package FSCINT FunctionSpaceComplexIntegration}
\getchunk{package FS2 FunctionSpaceFunctions2}
\getchunk{package FSINT FunctionSpaceIntegration}
\getchunk{package FSPREMELT FunctionSpacePrimitiveElement}
\getchunk{package FSRED FunctionSpaceReduce}
\getchunk{package SUMFS FunctionSpaceSum}
\getchunk{package FS2EXPXP FunctionSpaceToExponentialExpansion}
\getchunk{package FS2UPS FunctionSpaceToUnivariatePowerSeries}
\getchunk{package FSUPFACT FunctionSpaceUnivariatePolynomialFactor}
\getchunk{package GALFACTU GaloisGroupFactorizationUtilities}
\getchunk{package GALFACT GaloisGroupFactorizer}
\getchunk{package GALPOLL GaloisGroupPolynomialUtilities}
\getchunk{package GAUSSFC GaussFactorizationPackage}
\getchunk{package GHENSEL GeneralHenselPackage}
\getchunk{package GENMFACT GeneralizedMultivariateFactorize}
\getchunk{package CPAFF GeneralPackageForAlgebraicFunctionField}
\getchunk{package GENPOCD GeneralPolynomialGcdPackage}
\getchunk{package GENUPS GenerateUnivariatePowerSeries}
\getchunk{package GENEZ GenExEuclid}
\getchunk{package GENFACT GenUFactorize}
\getchunk{package INTG0 GenusZeroIntegration}
\getchunk{package GOSPER GosperSummationMethod}
\getchunk{package GRDEF GraphicsDefaults}
\getchunk{package GRAY GrayCode}
\getchunk{package GBF GroebnerFactorizationPackage}
\getchunk{package GBINTERN GroebnerInternalPackage}
\getchunk{package GB GroebnerPackage}
\getchunk{package GROEBESOL GroebnerSolve}
\getchunk{package GUESS Guess}
\getchunk{package GUESAN GuessAlgebraicNumber}
\getchunk{package GUESSF GuessFinite}
\getchunk{package GUESSF1 GuessFiniteFunctions}
\getchunk{package GUESST INT GuessInteger}
\getchunk{package GUESSP GuessPolynomial}
\getchunk{package GUESSUP GuessUnivariatePolynomial}
\getchunk{package HB HallBasis}
\getchunk{package HEUGCD HeuGcd}

\getchunk{package IDECOMP IdealDecompositionPackage}
\getchunk{package INCRMAPS IncrementingMaps}
\getchunk{package INFRPDO InfiniteProductCharacteristicZero}
\getchunk{package INFRDPFF InfiniteProductFiniteField}
\getchunk{package INFRDPDF InfiniteProductPrimeField}
\getchunk{package ITFUN2 InfiniteTupleFunctions2}
\getchunk{package ITFUN3 InfiniteTupleFunctions3}
\getchunk{package INFINITY Infinity}
\getchunk{package IALGFAC InnerAlgFactor}
\getchunk{package ICEDEN InnerCommonDenominator}
\getchunk{package IMATRIX InnerMatrixLinearAlgebraFunctions}
\getchunk{package IMATQF InnerMatrixQuotientFieldFunctions}
\getchunk{package INMODGCD InnerModularGcd}
\getchunk{package INMFAC InnerMultFact}
\getchunk{package INBF InnerNormalBasisFieldFunctions}
\getchunk{package INPE InnerNumericEigenPackage}
\getchunk{package INFSP InnerNumericFloatSolvePackage}
\getchunk{package INPSIGN InnerPolySign}
\getchunk{package ISUMP InnerPolySum}
\getchunk{package ITRIGMN InnerTrigonometricManipulations}
\getchunk{package INFORM1 InputFormFunctions1}
\getchunk{package INTBIT IntegerBits}
\getchunk{package COMBINAT IntegerCombinatoricFunctions}
\getchunk{package INTFACT IntegerFactorizationPackage}
\getchunk{package ZLINDEP IntegerLinearDependence}
\getchunk{package INTHEORY IntegerNumberTheoryFunctions}
\getchunk{package PRIMES IntegerPrimesPackage}
\getchunk{package INTRET IntegerRetractions}
\getchunk{package IROOT IntegerRoots}
\getchunk{package INTSLPE IntegerSolveLinearPolynomialEquation}
\getchunk{package IBATOL IntegralBasisTools}
\getchunk{package IRREDFFX IrredPolyOverFiniteField}
\getchunk{package IRSN IrrRepSymNatPackage}
\getchunk{package INVLAPLA InverseLaplaceTransform}
\getchunk{package KERNEL2 KernelFunctions2}
\getchunk{package KOVACIC Kovacic}
CHAPTER 28. CHUNK COLLECTIONS

\getchunk{package LAPLACE LaplaceTransform}
\getchunk{package LAZM3PK LazardSetSolvingPackage}
\getchunk{package LEADCDET LeadingCoefDetermination}
\getchunk{package LEXTRIPK LexTriangularPackage}
\getchunk{package LINDEP LinearDependence}
\getchunk{package LODOF LinearOrdinaryDifferentialOperatorFactorizer}
\getchunk{package LODOOPS LinearOrdinaryDifferentialOperatorsOps}
\getchunk{package LPEFRAC LinearPolynomialEquationByFractions}
\getchunk{package LSMP LinearSystemMatrixPackage}
\getchunk{package LSMP1 LinearSystemMatrixPackage1}
\getchunk{package LSPP LinearSystemPolynomialPackage}
\getchunk{package LOROBP LinGroebnerPackage}
\getchunk{package LOP LinesOpPack}
\getchunk{package LF LiouvillianFunction}
\getchunk{package LIST2 ListFunctions2}
\getchunk{package LIST3 ListFunctions3}
\getchunk{package LIST2MAP ListToMap}
\getchunk{package LPARSPT LocalParametrizationOfSimplePointPackage}
\getchunk{package MKBCFUNC MakeBinaryCompiledFunction}
\getchunk{package MKFLCFN MakeFloatCompiledFunction}
\getchunk{package MKFUNC MakeFunction}
\getchunk{package MKRECORD MakeRecord}
\getchunk{package MKUCFUNC MakeUnaryCompiledFunction}
\getchunk{package MAPHACK1 MappingPackageInternalHacks1}
\getchunk{package MAPHACK2 MappingPackageInternalHacks2}
\getchunk{package MAPHACK3 MappingPackageInternalHacks3}
\getchunk{package MAPPKG1 MappingPackage1}
\getchunk{package MAPPKG2 MappingPackage2}
\getchunk{package MAPPKG3 MappingPackage3}
\getchunk{package MAPPKG4 MappingPackage4}
\getchunk{package MATCAT2 MatrixCategoryFunctions2}
\getchunk{package MCDEN MatrixCommonDenominator}
\getchunk{package MATLIN MatrixLinearAlgebraFunctions}
\getchunk{package MAMA MatrixManipulation}
\getchunk{package MTHING MergeThing}
\getchunk{package MESH MeshCreationRoutinesForThreeDimensions}
\getchunk{package MDDFACT ModularDistinctDegreeFactorizer}
\getchunk{package MHROWRED ModularHermitianRowReduction}
\getchunk{package MRF2 MonoidRingFunctions2}
\getchunk{package MUNUTOOL MonomialExtensionTools}
\getchunk{package MSYSCMD MoreSystemCommands}
\getchunk{package MPCPF MPolyCatPolyFactorizer}
\getchunk{package MPRFF MPolyCatRationalFunctionFactorizer}
\getchunk{package MPC2 MPolyCatFunctions2}
\getchunk{package MPC3 MPolyCatFunctions3}
\getchunk{package MMATFAC MRationalFactorize}
\getchunk{package MFINFFACT MultiFiniteFactorize}
\getchunk{package MMAP MultipleMap}
\getchunk{package ORDCOMP2 OrderedCompletionFunctions2}
\getchunk{package ORDFUNS OrderingFunctions}
\getchunk{package ORTHPOL OrthogonalPolynomialFunctions}
\getchunk{package OUT OutputPackage}

\getchunk{package PAFF PackageForAlgebraicFunctionField}
\getchunk{package PAFFFF PackageForAlgebraicFunctionFieldOverFiniteField}
\getchunk{package PFORP PackageForPoly}
\getchunk{package PADEPAC PadeApproximantPackage}
\getchunk{package PADE PadeApproximants}
\getchunk{package PWFFINTB PAdicWildFunctionFieldIntegralBasis}
\getchunk{package YSTREAM ParadoxicalCombinatorsForStreams}
\getchunk{package PLEQN ParametricLinearEquations}
\getchunk{package PARPC2 ParametricPlaneCurveFunctions2}
\getchunk{package PARSC2 ParametricSpaceCurveFunctions2}
\getchunk{package PARSU2 ParametricSurfaceFunctions2}
\getchunk{package PARAMP ParametrizationPackage}
\getchunk{package PFRPAC PartialFractionPackage}
\getchunk{package PARTPERM PartitionsAndPermutations}
\getchunk{package PATTERN1 PatternFunctions1}
\getchunk{package PATTERN2 PatternFunctions2}
\getchunk{package PATMATCH PatternMatch}
\getchunk{package PMASS PatternMatchAssertions}
\getchunk{package PMFS PatternMatchFunctionSpace}
\getchunk{package PMINS PatternMatchIntegerNumberSystem}
\getchunk{package INTPM PatternMatchIntegration}
\getchunk{package PMSYM PatternMatchSymbol}
\getchunk{package PMKERNEL PatternMatchKernel}
\getchunk{package PMLSAGG PatternMatchListAggregate}
\getchunk{package PMPLCAT PatternMatchPolynomialCategory}
\getchunk{package PMDOWN PatternMatchPushDown}
\getchunk{package PMQFCAT PatternMatchQuotientFieldCategory}
\getchunk{package PATRES2 PatternMatchResultFunctions2}
\getchunk{package PMSYM PatternMatchSymbol}
\getchunk{package PMTOOLS PatternMatchTools}
\getchunk{package PERMAN Permanent}
\getchunk{package PGE PermutationGroupExamples}
\getchunk{package PICOERCE PiCoercions}
\getchunk{package PLOT1 PlotFunctions1}
\getchunk{package PLOTTOOL PlotTools1}
\getchunk{package PTFUNC2 PointFunctions2}
\getchunk{package PTPACK PointPackage}
\getchunk{package PFO PointsOfFiniteOrder}
\getchunk{package PFOQ PointsOfFiniteOrderRational}
\getchunk{package PFOTOOLS PointsOfFiniteOrderTools}
\getchunk{package POLTOPOL PolToPol}
\getchunk{package PQROEB PolyGroebner}
\getchunk{package PAN2EXPR PolynomialAN2Expression}
\getchunk{package POLYCATQ PolynomialCategoryQuotientFunctions}
\getchunk{package PCOMP PolynomialComposition}
\getchunk{package ODERAT RationalLODE}
\getchunk{package RATRET RationalRetractions}
\getchunk{package ODERTRIC RationalRicDE}
\getchunk{package RURPK RationalUnivariateRepresentationPackage}
\getchunk{package POLUTIL RealPolynomialUtilitiesPackage}
\getchunk{package REALSOLV RealSolvePackage}
\getchunk{package REAL0 RealZeroPackage}
\getchunk{package REAL0Q RealZeroPackageQ}
\getchunk{package RMAT2 RectangularMatrixCategoryFunctions2}
\getchunk{package RECOP RecurrenceOperator}
\getchunk{package RDIV ReducedDivisor}
\getchunk{package ODERED ReduceLODE}
\getchunk{package REDORDER ReductionOfOrder}
\getchunk{package RSDCMPK RegularSetDecompositionPackage}
\getchunk{package RSETGCD RegularTriangularSetGcdPackage}
\getchunk{package REPDDB RepeatedDoubling}
\getchunk{package REPSQ RepeatedSquaring}
\getchunk{package REP1 RepresentationPackage1}
\getchunk{package REP2 RepresentationPackage2}
\getchunk{package RESLATC ResolveLatticeCompletion}
\getchunk{package RETSOL RetractSolvePackage}
\getchunk{package RFP RootsFindingPackage}
\getchunk{package SAERFFC SAERationalFunctionAlgFactor}
\getchunk{package FORMULA1 ScriptFormulaFormat1}
\getchunk{package SEGBIND2 SegmentBindingFunctions2}
\getchunk{package SEG2 SegmentFunctions2}
\getchunk{package SAEFACT SimpleAlgebraicExtensionAlgFactor}
\getchunk{package STIMPAN SimplifyAlgebraicNumberConvertPackage}
\getchunk{package SMITH SmithNormalForm}
\getchunk{package SCACHE SortedCache}
\getchunk{package SORTPAK SortPackage}
\getchunk{package SUP2 SparseUnivariatePolynomialFunctions2}
\getchunk{package SPECOUT SpecialOutputPackage}
\getchunk{package SFQCMPK SquareFreeQuasiComponentPackage}
\getchunk{package SRDCMPK SquareFreeRegularSetDecompositionPackage}
\getchunk{package SFRGCD SquareFreeRegularTriangularSetGcdPackage}
\getchunk{package MATSTOR StorageEfficientMatrixOperations}
\getchunk{package STREAM1 StreamFunctions1}
\getchunk{package STREAM2 StreamFunctions2}
\getchunk{package STREAM3 StreamFunctions3}
\getchunk{package STINPROD StreamInfiniteProduct}
\getchunk{package STTAYLOR StreamTaylorSeriesOperations}
\getchunk{package STNSR StreamTensor}
\getchunk{package STTP StreamTranscendentalFunctions}
\getchunk{package STTFNC StreamTranscendentalFunctionsNonCommutative}
\getchunk{package SCPKG StructuralConstantsPackage}
\getchunk{package SHP SturmHabichtPackage}
\getchunk{package SUBRESP SubResultantPackage}
\getchunk{package SUPFRACF SupFractionFactorizer}
\getchunk{package ODESYS SystemODESolver}
\getchunk{package SYSSOLP SystemSolvePackage}
\getchunk{package SGCF SymmetricGroupCombinatoricFunctions}
\getchunk{package SYMFUNC SymmetricFunctions}
\getchunk{package TABLBUMP TableauxBumpers}
\getchunk{package TBCMPPK TabulatedComputationPackage}
\getchunk{package TANEXP TangentExpansions}
\getchunk{package UTSSOL TaylorSolve}
\getchunk{package TEMUTL TemplateUtilities}
\getchunk{package TEX1 TexFormat1}
\getchunk{package TOOLSIGN ToolsForSign}
\getchunk{package DRAW TopLevelDrawFunctions}
\getchunk{package DRAWCURV TopLevelDrawFunctionsForAlgebraicCurves}
\getchunk{package DRAWCFUN TopLevelDrawFunctionsForCompiledFunctions}
\getchunk{package DRAWPT TopLevelDrawFunctionsForPoints}
\getchunk{package TOPSP TopLevelThreeSpace}
\getchunk{package INTHRTR TranscendentalHermiteIntegration}
\getchunk{package INTR TranscendentalIntegration}
\getchunk{package TRMANIP TranscendentalManipulations}
\getchunk{package RETR TranscendentalRischDE}
\getchunk{package RETRIS TranscendentalRischDESystem}
\getchunk{package SOLVETRA TransSolvePackage}
\getchunk{package SOLVESER TransSolvePackageService}
\getchunk{package TRIMAT TriangularMatrixOperations}
\getchunk{package TRIMHIP TrigonometricManipulations}
\getchunk{package TUBETOOL TubePlotTools}
\getchunk{package CLIP TwoDimensionalPlotClipping}
\getchunk{package TWOFAC TwoFactorize}
\getchunk{package UNIFACT UnivariateFactorize}
\getchunk{package UPS1 UnivariateFormalPowerSeriesFunctions}
\getchunk{package ULS2 UnivariateLaurentSeriesFunctions2}
\getchunk{package UPOLYC2 UnivariatePolynomialCategoryFunctions2}
\getchunk{package UPOLCDEN UnivariatePolynomialCommonDenominator}
\getchunk{package UPDECOMP UnivariatePolynomialDecompositionPackage}
\getchunk{package UPDIVP UnivariatePolynomialDivisionPackage}
\getchunk{package UPM2 UnivariatePolynomialFunctions2}
\getchunk{package UPMP UnivariatePolynomialMultiplicationPackage}
\getchunk{package UPSQFREE UnivariatePolynomialSquareFree}
\getchunk{package UPX2 UnivariatePuiseuxSeriesFunctions2}
\getchunk{package OREPCT2 UnivariateSkewPolynomialCategoryOps}
\getchunk{package UTSS2 UnivariateTaylorSeriesFunctions2}
\getchunk{package UTSODE UnivariateTaylorSeriesODESolver}
\getchunk{package UNISEG2 UniversalSegmentFunctions2}
\getchunk{package UDO UserDefinedPartialOrdering}
\getchunk{package UDV UserDefinedVariableOrdering}
\getchunk{package UTSSDET ULTSodetools}
\getchunk{package VECTOR2 VectorFunctions2}
\getitem{package VIEWDEF ViewDefaultsPackage}
\getitem{package VIEW ViewportPackage}

\getitem{package WEIER WeierstrassPreparation}
\getitem{package WFFINTBS WildFunctionFieldIntegralBasis}

\getitem{package XEXPPKG XExponentialPackage}

\getitem{package ZDSOLVE ZeroDimensionalSolvePackage}
Chapter 29

Bibliography
Abstract: A four-parameter fit is developed for the class of integrals known as the exponential integral (real branch). Unlike other fits that are piecewise in nature, the current fit to the exponential integral is valid over the complete domain of the function (compact) and is everywhere accurate to within ±0.0052% when evaluating the first exponential integral, \( E_1 \). To achieve this result, a methodology that makes use of analytically known limiting behaviors at either extreme of the domain is employed. Because the fit accurately captures limiting behaviors of the \( E_1 \) function, more accuracy is retained when the fit is used as part of the scheme to evaluate higher-order exponential integrals, \( E_n \), as compared with the use of brute-force fits to \( E_1 \), which fail to accurately model limiting behaviors. Furthermore, because the fit is compact, no special accommodations are required (as in the case of spliced piecewise fits) to smooth the value, slope, and higher derivatives in the transition region between two piecewise domains. The general methodology employed to develop this fit is outlined, since it may be used for other problems as well.
Chapter 30

Index
Index

-?
  STTAYLOR, 3863

?,?,?
  AF, 14
  COMBF, 208
  INBFF, 1249
  MAPPKG1, 1614
  MATSTOR, 3845
  STTF, 3876
  STTFNC, 3886

???
  INBFF, 1249
  MAPPKG3, 1636
  MAPPKG4, 1643
  PRS, 3527
  STTAYLOR, 3863
  TUBETOOL, 4067

?+?
  MAPPKG4, 1643
  STTAYLOR, 3863
  TUBETOOL, 4067

?.?
  MAPPKG4, 1643
  STTAYLOR, 3863
  TUBETOOL, 4067

?/?
  INBFF, 1249
  MAPPKG4, 1643
  STTAYLOR, 3863

abelianGroup
  PGE, 3346
abs
  FSPECF, 872
accuracyIF
  D02AGNT, 490
aColumn

MAMA, 1703
acos
  EF, 535
  STTF, 3876
  STTFNC, 3886
acosh
  EF, 535
acot
  EF, 535
  STTF, 3876
  STTFNC, 3886
acoth
  EF, 535
acsc
  EF, 535
acsch
  EF, 535
aCubic
  SOLVEFOR, 3471
adaptive
  DROPT0, 421
  GRDEF, 1052
addBadValue
  PATTERN1, 3289
adddiag
  STTAYLOR, 3863
adjoint
  MATLIN, 1652
adjunctionDivisor
  DTP, 324
  GPAFF, 992
  PAFF, 3215
  PAFFF, 3223
AF, 14
???, 14
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>belong?</td>
<td>14</td>
</tr>
<tr>
<td>definingPolynomial</td>
<td>14</td>
</tr>
<tr>
<td>droot</td>
<td>14</td>
</tr>
<tr>
<td>inrootof</td>
<td>14</td>
</tr>
<tr>
<td>iroot</td>
<td>14</td>
</tr>
<tr>
<td>minPoly</td>
<td>14</td>
</tr>
<tr>
<td>operator</td>
<td>14</td>
</tr>
<tr>
<td>rootOf</td>
<td>14</td>
</tr>
<tr>
<td>AFALGGRO, 4</td>
<td></td>
</tr>
<tr>
<td>affineAlgSet, 4</td>
<td></td>
</tr>
<tr>
<td>affineRationalPoints, 4</td>
<td></td>
</tr>
<tr>
<td>affineSingularPoints, 4</td>
<td></td>
</tr>
<tr>
<td>AFALGRES, 9</td>
<td></td>
</tr>
<tr>
<td>affineAlgSet, 9</td>
<td></td>
</tr>
<tr>
<td>affineAlgSetLocal, 9</td>
<td></td>
</tr>
<tr>
<td>affineRationalPoints, 9</td>
<td></td>
</tr>
<tr>
<td>affineSingularPoints, 9</td>
<td></td>
</tr>
<tr>
<td>allPairsAmong, 9</td>
<td></td>
</tr>
<tr>
<td>polyRing2UPUP, 9</td>
<td></td>
</tr>
<tr>
<td>AffineAlgebraicSetComputeWithGroebnerBasis, 4</td>
<td></td>
</tr>
<tr>
<td>affineAlgSet, 9</td>
<td></td>
</tr>
<tr>
<td>ALFALGRES, 9</td>
<td></td>
</tr>
<tr>
<td>affineAlgSetLocal, 9</td>
<td></td>
</tr>
<tr>
<td>ALFALGRES, 9</td>
<td></td>
</tr>
<tr>
<td>affineRationalPoints, 9</td>
<td></td>
</tr>
<tr>
<td>ALFALGRES, 9</td>
<td></td>
</tr>
<tr>
<td>affineSingularPoints, 9</td>
<td></td>
</tr>
<tr>
<td>AIR</td>
<td>14</td>
</tr>
<tr>
<td>AlgebraicIntegrate, 24</td>
<td></td>
</tr>
<tr>
<td>AlgebraicIntegration, 32</td>
<td></td>
</tr>
<tr>
<td>AlgebraicManipulations, 35</td>
<td></td>
</tr>
<tr>
<td>AlgebraicMultFact, 41</td>
<td></td>
</tr>
<tr>
<td>algebraicSet</td>
<td></td>
</tr>
<tr>
<td>PRJALGPK, 3363</td>
<td></td>
</tr>
<tr>
<td>algebraicSort</td>
<td></td>
</tr>
<tr>
<td>QCMPACK, 3567</td>
<td></td>
</tr>
<tr>
<td>SFQCMPK, 3817</td>
<td></td>
</tr>
<tr>
<td>AlgebraPackage, 44</td>
<td></td>
</tr>
<tr>
<td>ALGFAC, 55</td>
<td></td>
</tr>
<tr>
<td>doublyTransitive?, 55</td>
<td></td>
</tr>
<tr>
<td>factor, 55</td>
<td></td>
</tr>
<tr>
<td>split, 55</td>
<td></td>
</tr>
<tr>
<td>AlgFactor, 55</td>
<td></td>
</tr>
<tr>
<td>algint, 55</td>
<td></td>
</tr>
<tr>
<td>INTAF, 32</td>
<td></td>
</tr>
<tr>
<td>alginTEGRATE</td>
<td></td>
</tr>
<tr>
<td>INTALG, 24</td>
<td></td>
</tr>
<tr>
<td>ALGFAC, 41</td>
<td></td>
</tr>
<tr>
<td>ratDenom, 35</td>
<td></td>
</tr>
<tr>
<td>ratPoly, 35</td>
<td></td>
</tr>
<tr>
<td>rootKerSimp, 35</td>
<td></td>
</tr>
<tr>
<td>rootPower, 35</td>
<td></td>
</tr>
<tr>
<td>rootProduct, 35</td>
<td></td>
</tr>
<tr>
<td>rootSimp, 35</td>
<td></td>
</tr>
<tr>
<td>rootSplit, 35</td>
<td></td>
</tr>
<tr>
<td>ALGFAC, 41</td>
<td></td>
</tr>
<tr>
<td>factor, 41</td>
<td></td>
</tr>
<tr>
<td>ALGPKG, 44</td>
<td></td>
</tr>
<tr>
<td>basis, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfCenter, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfCentroid, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfCommutingElements, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfLeftAnnihilator, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfLeftNucleus, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfLeftNucloid, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfMiddleNucleus, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfNucleus, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfRightAnnihilator, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfRightNucleus, 44</td>
<td></td>
</tr>
<tr>
<td>basisOfRightNucloid, 44</td>
<td></td>
</tr>
<tr>
<td>biRank, 44</td>
<td></td>
</tr>
<tr>
<td>doubleRank, 44</td>
<td></td>
</tr>
<tr>
<td>leftRank, 44</td>
<td></td>
</tr>
<tr>
<td>radicalOfLeftTraceForm, 44</td>
<td></td>
</tr>
<tr>
<td>airyAi</td>
<td>369</td>
</tr>
<tr>
<td>DFSFUN, 872</td>
<td></td>
</tr>
<tr>
<td>airyBi</td>
<td>369</td>
</tr>
<tr>
<td>DFSFUN, 872</td>
<td></td>
</tr>
<tr>
<td>algDsolve</td>
<td>3556</td>
</tr>
<tr>
<td>ODEPAL, 3556</td>
<td></td>
</tr>
<tr>
<td>algebraicDecompose</td>
<td></td>
</tr>
<tr>
<td>RSDCMPK, 3728</td>
<td></td>
</tr>
<tr>
<td>SRDCMPK, 3827</td>
<td></td>
</tr>
<tr>
<td>AlgebraicFunction, 14</td>
<td></td>
</tr>
<tr>
<td>AlgebraicHermiteIntegration, 19</td>
<td></td>
</tr>
</tbody>
</table>
rightRank, 44
weakBiRank, 44
aLinear
SOLVEFOR, 3471
allPairsAmong
AFALGRES, 9
alternating
CYCLES, 294
alternatingGroup
PGE, 3346
anfactor
FSUPFACT, 942
AnnaNumericalIntegrationPackage, 59
AnnaNumericalOptimizationPackage, 71
AnnaOrdinaryDifferentialEquationPackage, 81
AnnaPartialDifferentialEquationPackage, 91
anticoord
LGROBP, 1549
antisymmetricTensors
REP1, 3750
ANY1, 97
coerce, 98
retract, 98
retractable?, 98
retractIfCan, 98
AnyFunctions1, 97
API, 106
ApplicationProgramInterface, 106
apply
APPLYORE, 113
OREPCTO, 4122
APPLYORE, 113
apply, 113
ApplyRules, 109
applyRules
APPRULE, 109
applyTransform
BLUPPACK, 170
ApplyUnivariateSkewPolynomial, 113
approxNthRoot
IROOT, 1339
approxSqrt
IROOT, 1339
APPRULE, 109
applyRules, 109
localUnquote, 109
aQuadratic
SOLVEFOR, 3471
aQuartic
SOLVEFOR, 3471
areEquivalent?
REP2, 3758
arg1
MAPHACK2, 1602
arg2
MAPHACK2, 1602
aromberg
NUMQUAD, 3154
aRow
MAMA, 1703
approxNthRoot
IROOT, 1339
approxSqrt
IROOT, 1339
applyRules
APPRULE, 109
assert
PMASS, 3297
PMASSFS, 888
ASSOCEQ, 115
associatedEquations, 115
associatedSystem, 115
axServer, 121
getDatabase, 121
uncouplingMatrices, 115
AssociatedEquations, 115
associatedEquations
ASSOCEQ, 115
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
<th>Module</th>
</tr>
</thead>
<tbody>
<tr>
<td>associatedSystem</td>
<td></td>
<td>ASSOCEQ, 115</td>
</tr>
<tr>
<td>atan</td>
<td></td>
<td>EF, 535</td>
</tr>
<tr>
<td>atan</td>
<td></td>
<td>STTF, 3876</td>
</tr>
<tr>
<td>atan</td>
<td></td>
<td>STTFNC, 3886</td>
</tr>
<tr>
<td>atanh</td>
<td></td>
<td>EF, 535</td>
</tr>
<tr>
<td>atrapezoidal</td>
<td></td>
<td>NUMQUAD, 3154</td>
</tr>
<tr>
<td>att2Result</td>
<td></td>
<td>VIEWDEF, 4161</td>
</tr>
<tr>
<td>AttachPredicates</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>axesColorDefault</td>
<td></td>
<td>119</td>
</tr>
<tr>
<td>AxiomServer</td>
<td></td>
<td>121</td>
</tr>
<tr>
<td>AXSERV</td>
<td></td>
<td>121</td>
</tr>
<tr>
<td>multiServ</td>
<td></td>
<td>121</td>
</tr>
<tr>
<td>axServer</td>
<td></td>
<td>ASSOCEQ, 121</td>
</tr>
<tr>
<td>B1solve</td>
<td></td>
<td>PLEQN, 3257</td>
</tr>
<tr>
<td>badNum</td>
<td></td>
<td>PFOTOOLS, 3383</td>
</tr>
<tr>
<td>badValues</td>
<td></td>
<td>PATTERN1, 3289</td>
</tr>
<tr>
<td>BalancedFactorisation</td>
<td></td>
<td>138</td>
</tr>
<tr>
<td>balancedFactorisation</td>
<td></td>
<td>138</td>
</tr>
<tr>
<td>BALFACT, 138</td>
<td></td>
<td>138</td>
</tr>
<tr>
<td>bandedHessian</td>
<td></td>
<td>MCALCFN, 1768</td>
</tr>
<tr>
<td>bandedJacobian</td>
<td></td>
<td>MCALCFN, 1768</td>
</tr>
<tr>
<td>bandedJacobian</td>
<td></td>
<td>MAMA, 1703</td>
</tr>
<tr>
<td>bandMatrix</td>
<td></td>
<td>RDETR, 4027</td>
</tr>
<tr>
<td>baseRDE</td>
<td></td>
<td>RDETS, 4032</td>
</tr>
<tr>
<td>baseRDEsys</td>
<td></td>
<td>INTFACT, 1293</td>
</tr>
<tr>
<td>BasicMethod</td>
<td></td>
<td>INTFACT, 1293</td>
</tr>
<tr>
<td>BasicOperatorFunctions1</td>
<td></td>
<td>141</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>INBFF, 1249</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basis</td>
<td></td>
<td>ALGPKG, 44</td>
</tr>
<tr>
<td>basic</td>
<td></td>
<td>AF, 14</td>
</tr>
<tr>
<td>basic</td>
<td></td>
<td>COMBF, 208</td>
</tr>
<tr>
<td>basic</td>
<td></td>
<td>EF, 535</td>
</tr>
<tr>
<td>basic</td>
<td></td>
<td>FSPECF, 872</td>
</tr>
<tr>
<td>basic</td>
<td></td>
<td>LF, 1560</td>
</tr>
<tr>
<td>bernoulli</td>
<td></td>
<td>INTHEORY, 1320</td>
</tr>
<tr>
<td>bernoulli</td>
<td></td>
<td>PNTHEORY, 3442</td>
</tr>
<tr>
<td>bernoulliB</td>
<td></td>
<td>NTPOLFN, 3127</td>
</tr>
<tr>
<td>besselI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beauzamyBound</td>
<td></td>
<td>GALFACTU, 948</td>
</tr>
<tr>
<td>belong?</td>
<td></td>
<td>AF, 14</td>
</tr>
<tr>
<td>belong?</td>
<td></td>
<td>COMBF, 208</td>
</tr>
<tr>
<td>belong?</td>
<td></td>
<td>EF, 535</td>
</tr>
<tr>
<td>belong?</td>
<td></td>
<td>FSPECF, 872</td>
</tr>
<tr>
<td>belong?</td>
<td></td>
<td>LF, 1560</td>
</tr>
<tr>
<td>bernoulli</td>
<td></td>
<td>INTHEORY, 1320</td>
</tr>
<tr>
<td>bernoulli</td>
<td></td>
<td>PNTHEORY, 3442</td>
</tr>
<tr>
<td>bernoulliB</td>
<td></td>
<td>NTPOLFN, 3127</td>
</tr>
<tr>
<td>besselI</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
INDEX

DFSFUN, 369
FSPECF, 872
besselJ
DFSFUN, 369
FSPECF, 872
besselK
DFSFUN, 369
FSPECF, 872
besselY
DFSFUN, 369
FSPECF, 872
Beta
DFSFUN, 369
FSPECF, 872
RFDIST, 3613
BEZIER, 162
cubicBezier, 162
linearBezier, 162
quadraticBezier, 162
Bezier, 162
BEZOUT, 165
bezoutDiscriminant, 165
bezoutMatrix, 165
bezoutResultant, 165
brillhartIrreducible?, 180
brillhartTrials, 180
sylvesterMatrix, 165
bezoutDiscriminant
BEZOUT, 165
BezoutMatrix, 165
bezoutMatrix
BEZOUT, 165
bezoutResultant
BEZOUT, 165
binaryFunction
MKBCFUNC, 1582
binomial
COMBF, 208
COMBINAT, 1289
RIDIST, 3616
bipolar
COORDSYS, 268
bipolarCylindrical
COORDSYS, 268
biRank
ALGPKG, 44
biringToPolyRing
BLUPPACK, 170
bitCoef
INTBIT, 1285
bitLength
INTBIT, 1285
bitTruth
INTBIT, 1285
bivariate?
PSETPK, 3453
bivariatePolynomials
PSETPK, 3453
bivariateSLPEBR
PFBR, 3410
blockConcat
MAMA, 1703
blockSplit
MAMA, 1703
blowUp
DTP, 324
BlowUpPackage, 170
blowUpWithExcpDiv
DTP, 324
BLUPPACK, 170
applyTransform, 170
biringToPolyRing, 170
newtonPolySlope, 170
polyRingToBlUpRing, 170
quadTransform, 170
stepBlowUp, 170
bombieriNorm
GALFACTU, 948
BOP1, 141
constantOperator, 141
constantOpIfCan, 141
derivative, 141
evaluate, 141
BoundIntegerRoots, 177
boundOfCauchy
POLUTIL, 3684
BOUNDZERO, 177
integerBound, 177
branchIfCan
QCMPACK, 3567
SFQCMPK, 3817
bright
INDEX

DISPLAY, 345
BRILL, 180
   noLinearFactor?, 180
brillhartIrreducible?
   BEZOUT, 180
BrillhartTests, 180
brillhartTrials
   BEZOUT, 180
bringDown
   FSRED, 907
bsolve
   PLEQN, 3257
btwFact
   GALFACT, 954
BumInSepFFE
   UPSQFREE, 4115
bumprop
   TABLBUMP, 3942
bumptab
   TABLBUMP, 3942
bumptab1
   TABLBUMP, 3942
c02aff
   NAGC02, 2813
c02agf
   NAGC02, 2813
c05adf
   NAGC05, 2836
c05nbf
   NAGC05, 2836
c05pbf
   NAGC05, 2836
c06eaf
   NAGC06, 2901
c06ebf
   NAGC06, 2901
c06ecf
   NAGC06, 2901
c06ekf
   NAGC06, 2901
c06fuf
   NAGC06, 2901
c06gbf
   NAGC06, 2901
c06gcf
   NAGC06, 2901
c06gqf
   NAGC06, 2901
c06gsf
   NAGC06, 2901
cache
   SCACHE, 3805
CAD, 306
   coefficientSet, 306
cyldindricalDecomposition, 306
discriminantSet, 306
principalSubResultantSet, 306
projectionSet, 306
resultantSet, 306
specialise, 306
CADU, 311
   gcdBasis, 311
gcdBasisAdd, 311
   squareFreeBasis, 311
calcRanges
   PLOTTOOL, 3359
cap
   CYCLES, 294
CARTEN2, 186
   map, 186
   reshape, 186
cartesian
   COORDSYS, 268
   CartesianTensorFunctions2, 186
CDEN, 221
   clearDenominator, 221
center
   DISPLAY, 345
central?
   D03AGNT, 512
certainlySubVariety?
   PSETPK, 3453
cfirst
   WEIER, 4172
chainSubResultants
PRS, 3527
changeBase
  PSEUDLIN, 3521
changeName
  D01AGNT, 451
d changeNameToObjf
  E04AGNT, 721
ChangeOfVariable, 188
changeVar
  ODEPRRIC, 3512
characteristicPolynomial
  CHARPOL, 195
  CPIMA, 193
c EP, 516
  NCEP, 3167
  NREP, 3172
CharacteristicPolynomialInMonogenousAlgebra, 193
CharacteristicPolynomialPackage, 195
CHARPOL, 195
  characteristicPolynomial, 195
charpol
  INEP, 1258
chebyshevT
  ORTHPOL, 3207
  PNTHEORY, 3442
chebyshevU
  ORTHPOL, 3207
  PNTHEORY, 3442
check
  MULTSQFR, 1782
checkForZero
  DFINTTLS, 314
checkPrecision
  NAGSP, 2193
checkRur
  IRURPK, 1374
chineseRemainder
  CRAPACK, 273
  IBACHIN, 197
  INTHEORY, 1320
ChineseRemainderToolsForIntegralBases, 197
chiSquare
  RFDIST, 3613
chiSquare1
  RFDIST, 3613
choose
  CHVAR, 188
check
  LFO, 1560
CI
  CINTSLPE, 237
  solveLinearPolynomialEquation, 237
classNumber
  GPAFF, 992
  PAFF, 3215
  PAFFFF, 3223
clearCache
  SCACHE, 3805
clearDenominator
  CDEN, 221
  ICDEN, 1220
  MCODEN, 1649
  UPDCDEN, 4101
clearFortranOutputStack
  FOP, 842
clikeUniv
  WEIER, 4172
CLIP, 4071
clip
  CLIP, 4071
clipBoolean
  DROPT0, 421
clipParametric
  CLIP, 4071
clipPointsDefault
  GRDEF, 1052
clipWithRanges
  CLIP, 4071
CMPLXRT, 258
complexZeros, 258
<table>
<thead>
<tr>
<th>INDEX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cn</td>
<td>ELFUTS, 611</td>
</tr>
<tr>
<td>coefChoose</td>
<td>MULTSQFR, 1782</td>
</tr>
<tr>
<td>coefficientSet</td>
<td>CAD, 306</td>
</tr>
<tr>
<td>coerce</td>
<td>ANY1, 98</td>
</tr>
<tr>
<td></td>
<td>CVMP, 202</td>
</tr>
<tr>
<td></td>
<td>DRAWHACK, 419</td>
</tr>
<tr>
<td></td>
<td>FFHOM, 779</td>
</tr>
<tr>
<td></td>
<td>FORMULA1, 3788</td>
</tr>
<tr>
<td></td>
<td>MAPPKG1, 1614</td>
</tr>
<tr>
<td></td>
<td>NONE1, 3089</td>
</tr>
<tr>
<td></td>
<td>PAN2EXPR, 3396</td>
</tr>
<tr>
<td></td>
<td>PICOERCE, 3355</td>
</tr>
<tr>
<td></td>
<td>RESLATIC, 3775</td>
</tr>
<tr>
<td></td>
<td>RF, 3625</td>
</tr>
<tr>
<td></td>
<td>STTAYLOR, 3863</td>
</tr>
<tr>
<td></td>
<td>TEX1, 3960</td>
</tr>
<tr>
<td></td>
<td>VIEW, 4168</td>
</tr>
<tr>
<td>coerceP</td>
<td>CVMP, 202</td>
</tr>
<tr>
<td></td>
<td>CoerceVectorMatrixPackage, 202</td>
</tr>
<tr>
<td>coleman</td>
<td>SGCF, 3927</td>
</tr>
<tr>
<td>color</td>
<td>PTPACK, 3370</td>
</tr>
<tr>
<td>columns</td>
<td>MAMA, 1703</td>
</tr>
<tr>
<td>COMBF, 207</td>
<td>product, 208</td>
</tr>
<tr>
<td></td>
<td>splitDenominator, 221</td>
</tr>
<tr>
<td></td>
<td>summation, 208</td>
</tr>
<tr>
<td></td>
<td>COMBINAT, 1289</td>
</tr>
<tr>
<td></td>
<td>binomial, 1289</td>
</tr>
<tr>
<td></td>
<td>factorial, 1289</td>
</tr>
<tr>
<td></td>
<td>multinomial, 1289</td>
</tr>
<tr>
<td></td>
<td>partition, 1289</td>
</tr>
<tr>
<td></td>
<td>permutation, 1289</td>
</tr>
<tr>
<td></td>
<td>stirling1, 1289</td>
</tr>
<tr>
<td></td>
<td>stirling2, 1289</td>
</tr>
<tr>
<td></td>
<td>CombinatorialFunction, 207</td>
</tr>
<tr>
<td></td>
<td>combineFeatureCompatibility</td>
</tr>
<tr>
<td></td>
<td>D02AGNT, 490</td>
</tr>
<tr>
<td></td>
<td>commaSeparate</td>
</tr>
<tr>
<td></td>
<td>D01AGNT, 451</td>
</tr>
<tr>
<td></td>
<td>CommonDenominator, 221</td>
</tr>
<tr>
<td></td>
<td>commonDenominator</td>
</tr>
<tr>
<td></td>
<td>COMB, 221</td>
</tr>
<tr>
<td></td>
<td>ICDEN, 1220</td>
</tr>
<tr>
<td></td>
<td>MCDEN, 1649</td>
</tr>
<tr>
<td></td>
<td>UPCDEN, 4101</td>
</tr>
<tr>
<td></td>
<td>COMMONOP, 223</td>
</tr>
<tr>
<td></td>
<td>operator, 223</td>
</tr>
<tr>
<td></td>
<td>CommonOperators, 223</td>
</tr>
<tr>
<td></td>
<td>COMMUPE, 229</td>
</tr>
<tr>
<td></td>
<td>swap, 229</td>
</tr>
<tr>
<td></td>
<td>CommuteUnivariatePolynomialCategory, 229</td>
</tr>
<tr>
<td></td>
<td>comp</td>
</tr>
<tr>
<td></td>
<td>MAPHACK3, 1604</td>
</tr>
<tr>
<td></td>
<td>companionBlocks</td>
</tr>
<tr>
<td></td>
<td>PSEUDLIN, 3521</td>
</tr>
<tr>
<td></td>
<td>compBound</td>
</tr>
<tr>
<td></td>
<td>GENEEZ, 1027</td>
</tr>
<tr>
<td></td>
<td>compdegd</td>
</tr>
<tr>
<td></td>
<td>MULTSQFR, 1782</td>
</tr>
<tr>
<td></td>
<td>COMPFAC, 231</td>
</tr>
<tr>
<td></td>
<td>factor, 231</td>
</tr>
<tr>
<td></td>
<td>compiledFunction</td>
</tr>
<tr>
<td></td>
<td>MKBCFUNC, 1582</td>
</tr>
<tr>
<td></td>
<td>MKUOFUNC, 1597</td>
</tr>
<tr>
<td></td>
<td>complete</td>
</tr>
<tr>
<td></td>
<td>CYCLES, 294</td>
</tr>
<tr>
<td></td>
<td>completeEchelonBasis</td>
</tr>
<tr>
<td></td>
<td>REP2, 3758</td>
</tr>
<tr>
<td></td>
<td>completeEval</td>
</tr>
</tbody>
</table>
FACUTIL, 738
completeHensel
GHENSEL, 984
completeHermite
SMITH, 3799
completeSmith
SMITH, 3799
COMPLEX2, 235
map, 235
complexEigenvalues
NCEP, 3167
complexEigenvectors
NCEP, 3167
complexElementary
CTRIGMNP, 261
TRIGMNIP, 4062
complexExpand
IR2F, 1361
IRRF2F, 1358
ComplexFactorization, 231
complexForm
CTRIGMNP, 261
TRIGMNIP, 4062
ComplexFunctions2, 235
ComplexIntegerSolveLinearPolynomialEquation, 237
complexIntegrate
FSCINT, 893
IRRF2F, 1358
complexLimit
LIMITPS, 3485
LIMITRF, 3639
complexNormalize
CTRIGMNP, 261
TRIGMNIP, 4062
complexNumeric
NUMERIC, 3131
complexNumericIfCan
NUMERIC, 3131
ComplexPattern, 239
ComplexPatternMatch, 241
ComplexRootFindingPackage, 244
ComplexRootPackage, 258
complexRoots
FLOATCP, 830
ComplexTrigonometricManipulations, 260
complexZeros
CMPLXRT, 258
CRFP, 245
COMPLPAT, 239
convert, 239
compose
PCOMP, 3405
STTAYLOR, 3863
computeBasis
LGROBP, 1549
computeCycleEntry
CSTTOOLS, 300
computeCycleLength
CSTTOOLS, 300
computeInt
DFINTTLS, 314
concat
ESTOOLS, 667
STREAM1, 3851
conical
COORDSYS, 268
conjugate
CONSTANT, 3131
const
MAPPKG2, 1625
constant
MAPPKG2, 1625
PFORP, 3232
PMASS, 3297
PMASSFS, 888
constantCoefficientRicDE
ODEPRRIC, 3512
constantIfCan
KERNEL2, 1408
constantKernel
KERNEL2, 1408
constantLeft
MAPPKG3, 1636
ConstantLODE, 264
constantOperator
INDEX

BOP1, 141
constantOpIfCan
BOP1, 141
constantRight
MAPPKG3, 1636
constantToUnaryFunction
EXPRTUBE, 712
constDsolve
ODECONST, 264
content
HEUGCD, 1184
continuedFraction
NCNTFRAC, 3170
contract
IDECOMP, 1190
contractSolve
SOLVERAD, 3600
convert
COMPLPAT, 239
ESTOOLS, 667
RSDCMPK, 3728
SRDCMPK, 3827
ZDSOLVE, 4253
coord
DROPT0, 421
LGROBP, 1549
coordinates
SCPKG, 3893
CoordinateSystems, 268
COORDSYS, 268
bipolar, 268
bipolarCylindrical, 268
cartesian, 268
conical, 268
cylindrical, 268
elliptic, 268
ellipticCylindrical, 268
oblateSpheroidal, 268
parabolic, 268
parabolicCylindrical, 268
paraboloidal, 268
polar, 268
prolateSpheroidal, 268
spherical, 268
toroidal, 268
copies
DISPLAY, 345
copyfirst
POLYVEC, 4144
copyslice
POLYVEC, 4144
cos
EF, 535
STTF, 3876
STTFNC, 3886
cos2sec
TRMANIP, 4012
cosh
EF, 535
cosh2sech
TRMANIP, 4012
cosSinInfo
TUBETOOL, 4067
cot
EF, 535
STTF, 3876
STTFNC, 3886
cot2tan
TRMANIP, 4012
cot2trig
TRMANIP, 4012
coth
EF, 535
coth2tanh
TRMANIP, 4012
coth2trigh
TRMANIP, 4012
countRealRoots
SHP, 3897
countRealRootsMultiple
SHP, 3897
CPIMA, 193
characteristicPolynomial, 193
CPMATCH, 241
patternMatch, 241
CRAPACK, 273
chineseRemainder, 273
modTree, 273
multiEuclideanTree, 273
CRApackage, 273
createGenericMatrix
REP1, 3750
createIrreduciblePoly
  FFPOLY, 788
createLowComplexityNormalBasis
  FFF, 773
createLowComplexityTable
  FFF, 773
createMultiplicationMatrix
  FFF, 773
createMultiplicationTable
  FFF, 773
createNormalPoly
  FFPOLY, 788
createNormalPrimitivePoly
  FFPOLY, 788
createPrimitiveNormalPoly
  FFPOLY, 788
createPrimitivePoly
  FFPOLY, 788
createRandomElement
  REP2, 3758
createThreeSpace
  TOPSP, 3996
createZechTable
  FFF, 773
credPol
  GBINTERN, 1077
crest
  WEIER, 4172
CRFP, 244
criticalZeros, 245
divisorCascade, 245
factor, 245
graeffe, 245	norm, 245
pleskenSplit, 245
reciprocalPolynomial, 245
rootRadius, 245
schwerpunkt, 245
setErrorBound, 245
startPolynomial, 245
critB
  GBINTERN, 1077
critBonD
  GBINTERN, 1077
critM
  GBINTERN, 1077
critMonD1
  GBINTERN, 1077
critMTonD1
  GBINTERN, 1077
critpOrder
  GBINTERN, 1077
critT
  GBINTERN, 1077
cross
  TUBETOOL, 4067
crushedSet
  PSETPK, 3453
csc
  EF, 535
  STTF, 3876
  STTFNC, 3886
csc2sin
  TRMANIP, 4012
csch
  EF, 535
csch2sinh
  TRMANIP, 4012
CSTTOOLS, 300
  computeCycleEntry, 300
  computeCycleLength, 300
  cycleElt, 300
CTRIGMNP, 260
  complexElementary, 261
  complexForm, 261
  complexNormalize, 261
  imag, 261
  real, 261
  trigs, 261
cubic
  SOLVEFOR, 3471
cubicBezier
  BEZIER, 162
cup
  CYCLES, 294
curry
  MAPPKG2, 1625
curryLeft
  MAPPKG3, 1636
curryRight
  MAPPKG3, 1636
curveColorPalette
INDEX

DROPT0, 421
CVMP, 202
c coerce, 202
c coerceP, 202
cycleElts
CSTTOOLS, 300
CycleIndicators, 294
CYCLES, 294
alternating, 294
cap, 294
complete, 294
cup, 294
cyclic, 294
dihedral, 294
elementary, 294
eval, 294
graphs, 294
powerSum, 294
SFunction, 294
skewSFunction, 294
wreath, 294
cyclic
CYCLES, 294
cyclicGroup
PGE, 3346
CycicStreamTools, 300
cyclicSubmodule
REP2, 3758
CYCLOTOM, 303
cyclotomic, 303
cyclotomicDecomposition, 303
cyclotomicFactorization, 303
cyclotomic
CYCLOTOM, 303
NTPOLFN, 3127
PNTHEORY, 3442
cyclotomicDecomposition
CYCLOTOM, 303
cyclotomicFactorization
CYCLOTOM, 303
CyclotomicPolynomialPackage, 303
cylindrical
COORDSYS, 268
CylindricalAlgebraicDecompositionPackage, 306
CylindricalAlgebraicDecompositionUtilities, 311
cylindricalDecomposition
CAD, 306
d01AgentsPackage, 451
D01AGNT, 451
changeName, 451
commaSeparate, 451
df2st, 451
functionIsContinuousAtEndPoints, 451
functionIsOscillatory, 451
gethi, 451
getlo, 451
ldf2lst, 451
problemPoints, 451
rangeIsFinite, 451
sdf2lst, 451
singularitiesOf, 451
d01ajf
NAGD01, 2273
d01akf
NAGD01, 2273
d01alf
NAGD01, 2273
d01amf
NAGD01, 2273
d01anf
NAGD01, 2273
d01apf
NAGD01, 2273
d01aqf
NAGD01, 2273
d01asf
NAGD01, 2273
d01bbf
NAGD01, 2273
d01fcf
NAGD01, 2273
d01gaf
NAGD01, 2273
d01gbf
NAGD01, 2273
d01WeightsPackage, 458
D01WGTS, 458
exprHasAlgebraicWeight, 458
exprHasLogarithmicWeights, 458
exprHasWeightCosWXorSinWX, 458
d02AgentsPackage, 490
D02AGNT, 490
    accuracyIF, 490
    combineFeatureCompatibility, 490
    eval, 490
    expenseOfEvaluationIF, 490
    intermediateResultsIF, 490
    jacobian, 490
    sparsityIF, 490
    stiffnessAndStabilityFactor, 490
    stiffnessAndStabilityOfODEIF, 490
    systemSizeIF, 490

D02BBF
    NAGD02, 2745

D02BHF
    NAGD02, 2745

D02CJF
    NAGD02, 2745

D02EJF
    NAGD02, 2745

D02GAF
    NAGD02, 2745

D02GBF
    NAGD02, 2745

D02KEF
    NAGD02, 2745

D02RAF
    NAGD02, 2745

D03AGNT, 512
    central?, 512
    elliptic?, 512
    subscriptedVariables, 512
    varList, 512

D03EDF
    NAGD03, 2792

D03EUF
    NAGD03, 2792

d03faf
    NAGD03, 2792
dAndcExp
    INBFF, 1249
    DBLRESP, 411
        doubleResultant, 411
    DDFACT, 349
        distdfact, 349
        exptMod, 349

factor, 349
    factorSquareFree, 349
    irreducible?, 349
    separateDegrees, 349
    separateFactors, 349
    trace2PowMod, 349
    tracePowMod, 349

ddFact
    MDDFACT, 1719
decompose
    MONOTool, 1734
    PDECOMP, 3407
    RSDCMPK, 3728
    SRDCMPK, 3827
decomposeFunc
    SOLVESER, 4055
definingPolynomial
    AF, 14
    DefiniteIntegrationTools, 314
    DEFINTEF, 554
        innerint, 554
        integrate, 554
    DEFINTRF, 3628
        integrate, 3628
degOneCoef
    PFORP, 3232

degRED, 321
    expand, 321
    reduce, 321
degree
    FACUTIL, 738
    PFORP, 3232
    POLYVEC, 4144
degreeOfMinimalForm
    PFORP, 3232
degreePartition
    GALFACT, 954
    GALPOLYU, 971
    DegreeReductionPackage, 321
degreeSubResultant
    PRS, 3527
degreeSubResultantEuclidean
    PRS, 3527
denomLODE
    ODEPRIM, 3507
denomRicDE
INDEX

ODEPERRIC, 3512
deriv
STTAYLOR, 3863
derivative
BOP1, 141
desingTree
DTP, 324
GPAFF, 992
PAFF, 3215
PAFFF, 3223
desingTreeAtPoint
DTP, 324
DesingTreePackage, 324
desingTreeWoFullParam
GPAFF, 992
PAFF, 3215
PAFFF, 3223
determinant
MATLIN, 1652
df2ef
ESTOOLS, 667
df2fi
ESTOOLS, 667
df2mf
ESTOOLS, 667
df2st
D01AGNT, 451
ESCONT, 657
ESTOOLS, 667
DFINTTLS, 314
checkForZero, 314
computeInt, 314
ignore?, 314
dffist
ESTOOLS, 667
dfRange
ESTOOLS, 667
DFSFUN, 369
airyAi, 369
airyBi, 369
besselI, 369
besselJ, 369
besselK, 369
besselY, 369
Beta, 369
digamma, 369
E1, 369
Ei, 369
Ei1, 369
Ei2, 369
Ei3, 369
Ei4, 369
Ei5, 369
Ei6, 369
En, 369
fresnelC, 369
fresnelS, 369
Gamma, 369
hypergeometric0F1, 369
logGamma, 369
polygamma, 369
diag
MAPPKG2, 1625
diagonalMatrix
MAMA, 1703
diagonalProduct
IBATOOL, 1347
diff
ODEINT, 3182
DiffAction
FFFG, 852
DiffC
FFFG, 852
differentiate
POLYVEC, 4144
diffHP
GUESS, 1125
GUESSAN, 1162
GUESSF, 1164
GUESSINT, 1169
GUESSP, 1171
GUESSUP, 1174
digamma
DFSFUN, 369
FSPECF, 872
dihedral
CYCLES, 294
dihedralGroup
PGE, 3346
dilog
LF, 1560
dimensionOfIrreducibleRepresentation
INDEX

IRSN, 1394
dimensionsOf
NAGSP, 2193
DiophantineSolutionPackage, 334
diophantineSystem
SMITH, 3799
dioSolve
DIOSP, 334
DIOISP, 334
dioSolve, 334
direction
TOOLSIGN, 3963
DirectProductFunctions2, 339
directSum
LODOOPS, 1530
DIRPROD2, 339
map, 339
reduce, 339
scan, 339
DiscreteLogarithmPackage, 342
discriminant
NFINTBAS, 3116
PRS, 3527
discriminantEuclidean
PRS, 3527
discriminantSet
CAD, 306
DISPLAY, 345
bright, 345
center, 345
copies, 345
newLine, 345
say, 345
sayLength, 345
DisplayPackage, 345
distdfact
DDFACT, 349
distFact
LEADCDET, 1444
DistinctDegreeFactorize, 349
distinguishedCommonRootsOf
RFP, 3781
distinguishedRootsOf
RFP, 3781
divergence
MCALCFN, 1768
divide
PRS, 3527
divideIfCan
UPDIVP, 4107
divisorAtDesingTree
DTP, 324
divisorCascade
CRFP, 245
divisors
INTHEORY, 1320
DLP, 342
shanksDiscLogAlgorithm, 342
dmp2rfi
PLEQN, 3257
dmpToHdmp
POLTOPOL, 3390
dmpToP
POLTOPOL, 3390
dn
ELFUTS, 611
dot
TUBETOOL, 4067
dot2eps
GRAPHVIZ, 1057
dotview
GRAPHVIZ, 1057
double
REPDB, 3745
doubleDisc
PFOTOOLS, 3383
DoubleFloatSpecialFunctions, 369
doubleRank
ALGPKG, 44
doubleResultant
DBLRESP, 411
DoubleResultantPackage, 411
doublyTransitive?
ALGFAC, 55
DRAW, 3965
draw, 3965
makeObject, 3965
draw
DRAW, 3965
DRAWCFS, 3978
DRAWCURV, 3974
DRAWPT, 3993
INDEX

DRAWCFUN, 3978
draw, 3978
makeObject, 3978
recolor, 3978
DrawComplex, 414
drawComplex
  DRAWCX, 414
drawComplexVectorField
    DRAWCX, 414
DRAWCURV, 3974
draw, 3974
drawCurves
  VIEW, 4168
DRAWCX, 414
drawComplex, 414
drawComplexVectorField, 414
setClipValue, 414
setImagSteps, 414
setRealSteps, 414
DRAWHACK, 419
coerce, 419
DrawNumericHack, 419
DrawOptionFunctions0, 421
DrawOptionFunctions1, 426
DRAWPT, 3993
draw, 3993
drawToScale
  GRDEF, 1052
droot
  AF, 14
DROPT0, 421
  adaptive, 421
  clipBoolean, 421
  coord, 421
  curveColorPalette, 421
  pointColorPalette, 421
  ranges, 421
  space, 421
  style, 421
  title, 421
toScale, 421
tubePoints, 421
tubeRadius, 421
units, 421
var1Steps, 421
var2Steps, 421
  viewpoint, 421
DROPT1, 426
  option, 426
DTP, 324
  adjunctionDivisor, 324
  blowUp, 324
  blowUpWithExcpDiv, 324
  desingTree, 324
  desingTreeAtPoint, 324
  divisorAtDesingTree, 324
  fullParamInit, 324
  genus, 324
  genusNeg, 324
  genusTree, 324
  genusTreeNeg, 324
  inBetweenExcpDiv, 324
  initializeParamOfPlaces, 324
  initParLocLeaves, 324
e01baf
  NAGE01, 2339
e01bef
  NAGE01, 2339
e01bff
  NAGE01, 2339
e01bgf
  NAGE01, 2339
e01bhf
  NAGE01, 2339
e01daf
  NAGE01, 2339
e01saf
  NAGE01, 2339
e01baf
  NAGE01, 2339
e01sbf
  NAGE01, 2339
e01sef
  NAGE01, 2339
e01sff
  NAGE01, 2339
e02adf
  NAGE02, 2074
e02aef
  NAGE02, 2074
e02agf
  NAGE02, 2074
e02ahf
NAGE02, 2074
  e02ajf
  NAGE02, 2074
e02akf
  NAGE02, 2074
e02baf
  NAGE02, 2074
e02bbf
  NAGE02, 2074
e02bcf
  NAGE02, 2074
e02bdf
  NAGE02, 2074
e02bef
  NAGE02, 2074
e02daf
  NAGE02, 2074
e02dcf
  NAGE02, 2074
e02dgf
  NAGE02, 2074
e02df
  NAGE02, 2074
e02ddf
  NAGE02, 2074
e02def
  NAGE02, 2074
e02dff
  NAGE02, 2074
e02gaf
  NAGE02, 2074
e02zaf
  NAGE02, 2074
e04AgentsPackage, 721
E04AGNT, 721
  changeNameToObjf, 721
e04df
  NAGE04, 2646
e04ddf
  NAGE04, 2646
e04gcf
  NAGE04, 2646
e04jaf
  NAGE04, 2646
e04mbf
  NAGE04, 2646
e04naf
  NAGE04, 2646
e04ucf
  NAGE04, 2646
e04ycf
  NAGE04, 2646
E1
  DFSFUN, 369
edf2df
  ESTOOLS, 667
edf2ef
  ESTOOLS, 667
edf2ei
  ESTOOLS, 667
edf2fi
  ESTOOLS, 667
EF, 535
  acos, 535
  acosh, 535
  acot, 535
  acoth, 535
  acsc, 535
  acsch, 535
  asec, 535
  asech, 535
  asin, 535
  asinh, 535
  atan, 535
  atanh, 535
  belong?, 535
cos, 535
cosh, 535
cot, 535
coth, 535
csc, 535
csch, 535
INDEX

exp, 535
iiacos, 535
iiacosh, 535
iiacot, 535
iiacoth, 535
iiacsc, 535
iiacsch, 535
iiasec, 535
iiasech, 535
iiasin, 535
iiasinh, 535
iiatan, 535
iiatanh, 535
iicos, 535
iicosh, 535
iicot, 535
iicoth, 535
iicsc, 535
iicsch, 535
iicosh, 535
iiccot, 535
iicoth, 535
iicsc, 535
iicsch, 535
iiexp, 535
iilog, 535
iisec, 535
iisech, 535
iisin, 535
iisinh, 535
iisqrt2, 535
iisqrt3, 535
iiatan, 535
iiatanh, 535
localReal?, 535
log, 535
operator, 535
pi, 535
sec, 535
sech, 535
sin, 535
sinh, 535
specialTrigs, 535
tan, 535
tanh, 535
ef2edf

ESTOOLS, 667
EFSTRUC, 579
normalize, 579
realElementary, 579
rischNormalize, 579

rootNormalize, 579
tanQ, 579
validExponential, 579

Ei

DFSFUN, 369
LF, 1560

Ei1

DFSFUN, 369

Ei2

DFSFUN, 369

Ei3

DFSFUN, 369

Ei4

DFSFUN, 369

Ei5

DFSFUN, 369

Ei6

eigenMatrix

REp, 3582

EigenPackage, 516
eigenvalues

EP, 516
eigenvector

EP, 516
eigenvectors

EP, 516
eisensteinIrreducible?

GALFACT, 954
element

MAMA, 1703
elementary

CYCLES, 294
ElementaryFunction, 535
ElementaryFunctionDefiniteIntegration, 554
ElementaryFunctionLODESolver, 560
ElementaryFunctionODESolver, 567
ElementaryFunctionSign, 574
ElementaryFunctionStructurePackage, 579
ElementaryIntegration, 589
ElementaryRischDE, 599
ElementaryRischDESystem, 608
ELFUTS, 611
cn, 611
dn, 611
sn, 611
sncndn, 611
elliptic
  COORDSYS, 268
elliptic?
  D03AGNT, 512
ellipticCylindrical
  COORDSYS, 268
EllipticFunctionsUnivariateTaylorSeries, 611
En
  DFSFUN, 369
enterInCache
  SCACHE, 3805
EP, 516
  characteristicPolynomial, 516
eigenvalues, 516
eigenvector, 516
eigenvectors, 516
generalizedEigenvector, 516
generalizedEigenvectors, 516
EQ2, 614
  map, 614
EquationFunctions2, 614
erf
  LF, 1560
ERROR, 617
  error, 617
error
  ERROR, 617
ErrorFunctions, 617
ES1, 686
  map, 686
ES2, 688
  map, 688
ESCONT, 657
  df2st, 657
  functionIsFracPolynomial?, 657
gethi, 657
getlo, 657
ldf2lst, 657
polynomialZeros, 657
problemPoints, 657
sdf2lst, 657
  singularitiesOf, 657
zerosOf, 657
ESCONT1, 664
  in?, 664
ESTOOLS, 667
  att2Result, 667
  concat, 667
  convert, 667
  df2ef, 667
df2fi, 667
df2mf, 667
df2st, 667
dffist, 667
dfrange, 667
edf2df, 667
edf2ef, 667
edf2efi, 667
edf2fi, 667
edf2edf, 667
expenseOfEvaluation, 667
f2df, 667
f2st, 667
f2df, 667
gethi, 667
getlo, 667
iflist2Result, 667
in?, 667
isQuotient, 667
ldf2lst, 667
ldf2vmf, 667
mat, 667
measure2Result, 667
numberOfOperations, 667
ocf2ocdf, 667
outputMeasure, 667
pdf2df, 667
pdf2ef, 667
pdf2df, 667
sdf2lst, 667
socf2socdf, 667
vedf2vef, 667
ESTOOLS1, 676
  neglist, 676
ESTOOLS2, 678
  map, 678
euclideanGroebner
  GBEUCLID, 642
EuclideanGroebnerBasisPackage, 642
euclideanNormalForm
  GBEUCLID, 642
euler
<table>
<thead>
<tr>
<th>INDEX</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>INTHEORY, 1320</td>
<td></td>
</tr>
<tr>
<td>PNTHEORY, 3442</td>
<td></td>
</tr>
<tr>
<td>eulerE</td>
<td></td>
</tr>
<tr>
<td>NTPOLFN, 3127</td>
<td></td>
</tr>
<tr>
<td>eulerPhi</td>
<td></td>
</tr>
<tr>
<td>INTHEORY, 1320</td>
<td></td>
</tr>
<tr>
<td>eval</td>
<td></td>
</tr>
<tr>
<td>CHVAR, 188</td>
<td></td>
</tr>
<tr>
<td>CYCLES, 294</td>
<td></td>
</tr>
<tr>
<td>D02AGNT, 490</td>
<td></td>
</tr>
<tr>
<td>EVALCYC, 655</td>
<td></td>
</tr>
<tr>
<td>GPAFF, 992</td>
<td></td>
</tr>
<tr>
<td>PAFF, 3215</td>
<td></td>
</tr>
<tr>
<td>PAFFFF, 3223</td>
<td></td>
</tr>
<tr>
<td>PLPKCRV, 3386</td>
<td></td>
</tr>
<tr>
<td>RF, 3625</td>
<td></td>
</tr>
<tr>
<td>STTAYLOR, 3863</td>
<td></td>
</tr>
<tr>
<td>evalADE</td>
<td></td>
</tr>
<tr>
<td>RECORP, 3707</td>
<td></td>
</tr>
<tr>
<td>evalat</td>
<td></td>
</tr>
<tr>
<td>POLYVEC, 4144</td>
<td></td>
</tr>
<tr>
<td>EVALCYC, 655</td>
<td></td>
</tr>
<tr>
<td>eval, 655</td>
<td></td>
</tr>
<tr>
<td>evalIfCan</td>
<td></td>
</tr>
<tr>
<td>GPAFF, 992</td>
<td></td>
</tr>
<tr>
<td>PAFF, 3215</td>
<td></td>
</tr>
<tr>
<td>PAFFFF, 3223</td>
<td></td>
</tr>
<tr>
<td>evalRec</td>
<td></td>
</tr>
<tr>
<td>RECORP, 3707</td>
<td></td>
</tr>
<tr>
<td>evaluate</td>
<td></td>
</tr>
<tr>
<td>BOP1, 141</td>
<td></td>
</tr>
<tr>
<td>EvaluateCycleIndicators, 655</td>
<td></td>
</tr>
<tr>
<td>evenInfiniteProduct</td>
<td></td>
</tr>
<tr>
<td>INFPROD0, 1202</td>
<td></td>
</tr>
<tr>
<td>INPRODFF, 1204</td>
<td></td>
</tr>
<tr>
<td>INPRODPF, 1208</td>
<td></td>
</tr>
<tr>
<td>STINPROD, 3859</td>
<td></td>
</tr>
<tr>
<td>evenlambert</td>
<td></td>
</tr>
<tr>
<td>STTAYLOR, 3863</td>
<td></td>
</tr>
<tr>
<td>exp</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>STTF, 3876</td>
<td></td>
</tr>
<tr>
<td>STTFNC, 3886</td>
<td></td>
</tr>
<tr>
<td>XEXPPKG, 4184</td>
<td></td>
</tr>
<tr>
<td>EXP3D, 717</td>
<td></td>
</tr>
<tr>
<td>writeObj, 717</td>
<td></td>
</tr>
<tr>
<td>expand</td>
<td></td>
</tr>
<tr>
<td>DEGRED, 321</td>
<td></td>
</tr>
<tr>
<td>IR2F, 1361</td>
<td></td>
</tr>
<tr>
<td>IRRF2F, 1358</td>
<td></td>
</tr>
<tr>
<td>TRMANIP, 4012</td>
<td></td>
</tr>
<tr>
<td>expandLog</td>
<td></td>
</tr>
<tr>
<td>TRMANIP, 4012</td>
<td></td>
</tr>
<tr>
<td>expandPower</td>
<td></td>
</tr>
<tr>
<td>TRMANIP, 4012</td>
<td></td>
</tr>
<tr>
<td>expandTrigProducts</td>
<td></td>
</tr>
<tr>
<td>TRMANIP, 4012</td>
<td></td>
</tr>
<tr>
<td>expenseOfEvaluation</td>
<td></td>
</tr>
<tr>
<td>E04AGNT, 721</td>
<td></td>
</tr>
<tr>
<td>ESTOOLS, 667</td>
<td></td>
</tr>
<tr>
<td>expenseOfEvaluationIF</td>
<td></td>
</tr>
<tr>
<td>D02AGNT, 490</td>
<td></td>
</tr>
<tr>
<td>ExpertSystemContinuityPackage, 657</td>
<td></td>
</tr>
<tr>
<td>ExpertSystemContinuityPackage1, 664</td>
<td></td>
</tr>
<tr>
<td>ExpertSystemToolsPackage, 667</td>
<td></td>
</tr>
<tr>
<td>ExpertSystemToolsPackage1, 676</td>
<td></td>
</tr>
<tr>
<td>ExpertSystemToolsPackage2, 678</td>
<td></td>
</tr>
<tr>
<td>expextendedint</td>
<td></td>
</tr>
<tr>
<td>INTTR, 4001</td>
<td></td>
</tr>
<tr>
<td>expint</td>
<td></td>
</tr>
<tr>
<td>ODEINT, 3182</td>
<td></td>
</tr>
<tr>
<td>expintegrate</td>
<td></td>
</tr>
<tr>
<td>INTTR, 4001</td>
<td></td>
</tr>
<tr>
<td>expintf2dpoly</td>
<td></td>
</tr>
<tr>
<td>INTTR, 4001</td>
<td></td>
</tr>
<tr>
<td>explimitedint</td>
<td></td>
</tr>
<tr>
<td>INTTR, 4001</td>
<td></td>
</tr>
<tr>
<td>explogs2trigs</td>
<td></td>
</tr>
<tr>
<td>ITRIGMNP, 1275</td>
<td></td>
</tr>
<tr>
<td>exponential</td>
<td></td>
</tr>
<tr>
<td>RFDIST, 3613</td>
<td></td>
</tr>
<tr>
<td>exponential1</td>
<td></td>
</tr>
<tr>
<td>RFDIST, 3613</td>
<td></td>
</tr>
<tr>
<td>Export3D, 717</td>
<td></td>
</tr>
<tr>
<td>expPot</td>
<td></td>
</tr>
<tr>
<td>INBFF, 1249</td>
<td></td>
</tr>
<tr>
<td>EXPR2, 680</td>
<td></td>
</tr>
<tr>
<td>map, 680</td>
<td></td>
</tr>
<tr>
<td>EXPR2UPS, 704</td>
<td></td>
</tr>
<tr>
<td>laurent, 704</td>
<td></td>
</tr>
<tr>
<td>puiseux, 704</td>
<td></td>
</tr>
<tr>
<td>series, 704</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

4302

taylor, 704
ExpressionFunctions2, 680
ExpressionSolve, 683
ExpressionSpaceFunctions1, 686
ExpressionSpaceFunctions2, 688
ExpressionSpaceODESolver, 691
ExpressionToOpenMath, 696
ExpressionToUnivariatePowerSeries, 704
ExpressionTubePlot, 712
exprHasAlgebraicWeight
  D01WGTS, 458
exprHasLogarithmicWeights
  D01WGTS, 458
exprHasWeightCosWXorSinWX
  D01WGTS, 458
EXPRODE, 691
  seriesSolve, 691
EXPRSOL, 683
  replaceDiffs, 683
  seriesSolve, 683
exprToGenUPS
  FS2UPS, 925
exprToUPS
  FS2UPS, 925
exprToXXP
  FS2EXPXP, 913
EXPRTUBE, 712
  constantToUnaryFunction, 712
  tubePlot, 712
expt
  REPSQ, 3747
exptMod
  DDFACT, 349
  MDDFACT, 1719
exquo
  PRS, 3527
  STTAYLOR, 3863
extendedcedged
  POLYVEC, 4144
extendedint
  INTRAT, 3655
extendedIntegrate
  INTRF, 3636
extractIfCan
  TBCMPPK, 3947

F
  RFDIST, 3613
  f01brf
  NAGF01, 2452
  f01bsf
  NAGF01, 2452
  f01maf
  NAGF01, 2452
  f01mcf
  NAGF01, 2452
  f01qcf
  NAGF01, 2452
  f01qdf
  NAGF01, 2452
  f01qef
  NAGF01, 2452
  f01rcf
  NAGF01, 2452
  f01rdf
  NAGF01, 2452
  f01ref
  NAGF01, 2452
  f01raf
  NAGF01, 2452
  f01raf
  NAGF01, 2452
  f02aaf
  NAGF02, 1889
  f02abf
  NAGF02, 1889
  f02adf
  NAGF02, 1889
  f02aef
  NAGF02, 1889
  f02aff
  NAGF02, 1889
  f02agf
  NAGF02, 1889
  f02ajf
  NAGF02, 1889
  f02akf
  NAGF02, 1889
  f02awf
  NAGF02, 1889
  f02axf
  NAGF02, 1889
  f02baf
  NAGF02, 1889
  f02bjf
  NAGF02, 1889
INDEX

f02jf
    NAGF02, 1889
f02wef
    NAGF02, 1889
f02xef
    NAGF02, 1889
f04ad
    NAGF04, 2183
f04arf
    NAGF04, 2183
f04asf
    NAGF04, 2183
f04atf
    NAGF04, 2183
f04axf
    NAGF04, 2183
f04d
    NAGF04, 2183
f04g
    NAGF04, 2183
f04maf
    NAGF04, 2183
f04mbf
    NAGF04, 2183
f04mcf
    NAGF04, 2183
f04qaf
    NAGF04, 2183
f07adf
    NAGF07, 2367
f07aef
    NAGF07, 2367
f07ddf
    NAGF07, 2367
f07ef
    NAGF07, 2367
f2df
    ESTOOLS, 667
F2EXPRR
    GUESSF1, 1166
F2FG
    ITRIGMNP, 1275
f2st
    ESTOOLS, 667
FACTTEXT, 741
    factor, 741
factor
    FACTFUNC, 728
    log, 728
    nthRoot, 728
ALGFAC, 55
    ALGMFACT, 41
    COMPFACT, 231
    CRF, 245
    DDFACT, 349
    FACTEXT, 741
    FACTRN, 746
    FFFACTOR, 760
    FFFACTSE, 766
    GALFACT, 954
    GAUSSFAC, 979
    GENMFACT, 988
    GENUFACT, 1032
    IALGFAC, 1217
    INNMFACT, 1239
    INTFACT, 1293
    LODOF, 1526
    MDDFACT, 1719
    MFINFACT, 1754
    MPCPF, 1739
    MPRFF, 1742
    MRATFAC, 1752
    MULTFACT, 1773
    RATFACT, 3622
    RFFACT, 3631
    RFFACTOR, 3634
    SAEFACT, 3795
    SAERFFC, 3786
    SUPFRA, 3910
    UNIFACT, 4086
factor1
    LOD0F, 1526
    factorByRecursion
    PFBR, 3410
    PFBRU, 3418
    factorCantorZassenhaus
    FFFACTOR, 760
    FFFACTSE, 766
    FactoredFunctions, 728
    FactoredFunctions2, 733
    FactoredFunctionUtilities, 735
factorGroebnerBasis
   GBF, 1069
factorial
   COMBF, 208
   COMBINAT, 1289
factorials
   COMBF, 208
FactoringUtilities, 738
FactorisationOverPseudoAlgebraicClosureOfAlgebraicExtensionOfRationalNumber,
   741
FactorisationOverPseudoAlgebraicClosureOfRationalNumber,
   746
factorList
   IBACHIN, 197
factorOfDegree
   GALFACT, 954
   GALPOLYU, 971
factorset
   PLEQN, 3257
factorSFBRlcUnit
   PFBR, 3410
   PFBRU, 3418
factorsOfDegree
   GALPOLYU, 971
factorSqFree
   FACTEXT, 741
   FACTRN, 746
factorSquareFree
   DDFACT, 349
   FFFACTOR, 760
   FFFACTSE, 766
   GALFACT, 954
   RATFACT, 3622
   UNIFACT, 4086
factorSquareFreeByRecursion
   PFBR, 3410
   PFBRU, 3418
factorUsingMusser
   FFFACTOR, 760
   FFFACTSE, 766
factorUsingYun
   FFFACTOR, 760
   FFFACTSE, 766
   FACTRN, 746
   factor, 746
   factorSqFree, 746
FACUTIL, 738
   completeEval, 738
   degree, 738
   lowerPolynomial, 738
   normalDeriv, 738
   raisePolynomial, 738
   ran, 738
   variables, 738
FAMR2, 755
   map, 756
FFDIV2, 758
   map, 758
ffactor
   FSUPFACT, 942
   FFCAT2, 881
   map, 881
FFF, 772
   createLowComplexityNormalBasis, 773
   createLowComplexityMatrix, 773
   createMultiplicationMatrix, 773
   createMultiplicationTable, 773
   createZechTable, 773
   sizeMultiplication, 773
FFFACTOR, 760
   factor, 760
   factorCantorZassenhaus, 760
   factorSquareFree, 760
   factorUsingMusser, 760
   factorUsingYun, 760
   irreducible?, 760
FFFACTSE, 766
   factor, 766
   factorCantorZassenhaus, 766
   factorSquareFree, 766
   factorUsingMusser, 766
   factorUsingYun, 766
   irreducible?, 766
FFFG, 852
   DiffAction, 852
   DiffC, 852
   fffg, 852
INDEX

generalCoefficient, 852
generalInterpolation, 852
interpolate, 852
qShiftAction, 852
qShiftC, 852
ShiftAction, 852
ShiftC, 852

ffg
  FFFG, 852

FFGF, 863
  generalInterpolation, 863

FFHOM, 779
cerce, 779

FFINTBAS, 884
  integralBasis, 884
  localIntegralBasis, 884

FPOLY, 788
  createIrreduciblePoly, 788
  createNormalPoly, 788
  createNormalPrimitivePoly, 788
  createPrimitiveNormalPoly, 788
  createPrimitivePoly, 788
  leastAffineMultiple, 788
  nextIrreduciblePoly, 788
  nextNormalPoly, 788
  nextNormalPrimitivePoly, 788
  nextPrimitiveNormalPoly, 788
  normal?, 788
  numberOfIrreduciblePoly, 788
  numberOfNormalPoly, 788
  numberOfPrimitivePoly, 788
  primitive?, 788
  random, 788
  reducedQPowers, 788

FPOLY2, 809
  rootOfIrreduciblePoly, 809

FFSLPE, 813
  solveLinearPolynomialEquation, 813

FFSQFR, 816
  PolK, 816
  Yun, 816

FG2F
  ITRIGMNP, 1275

FGLMICPK, 750
  fglmIfCan, 750
  groebner, 750
  zeroDimensional?, 750

ESTOOLS, 667

fibonacci
  INTHEORY, 1320
  fillPascalTriangle
    GALUTIL, 975
    FindOrderFinite, 753
    findOrderOfDivisor
      GPAFF, 992
      PAFF, 3215
      PAFFFF, 3223
      FiniteAbelianMonoidRingFunctions2, 755
      finiteBound
        E04AGNT, 721
        FiniteDivisorFunctions2, 758
        FiniteFieldFactorization, 760
        FiniteFieldFactorizationWithSizeParseBySideEffect, 766
        FiniteFieldFunctions, 772
        FiniteFieldHomomorphisms, 779
        FiniteFieldPolynomialPackage, 788
        FiniteFieldPolynomialPackage2, 809
        FiniteFieldSolveLinearPolynomialEquation, 813
        FiniteFieldSquareFreeDecomposition, 816
        FiniteLinearAggregateFunctions2, 819
        FiniteLinearAggregateSort, 823
        finiteSeries2LinSys
          LISYSER, 1536
          finiteSeries2LinSysWOVectorise
            LISYSER, 1536
          finiteSeries2Vector
            LISYSER, 1536
          FiniteSetAggregateFunctions2, 827
          firstExponent
            PFORP, 3232
          firstSubsetGray
            GRAY, 1061
          firstUncouplingMatrix
            PREASSOC, 3497
          fixedDivisor
INDEX

PNTHEORY, 3442
fixedPoint
    MAPPKG1, 1614
fixedPointExquo
    UTSODE, 4128
fixPredicate
    PMDOWN, 3324
FLAGG2, 819
    map, 820
    reduce, 820
    scan, 820
FLASORT, 823
    heapSort, 823
    quickSort, 823
    shellSort, 823
FLOATCP, 830
    complexRoots, 830
    complexSolve, 830
FloatingComplexPackage, 830
FloatingRealPackage, 834
FLOATRP, 834
    realRoots, 834
    solve, 834
FOP, 842
    clearFortranOutputStack, 842
    popFortranOutputStack, 842
    pushFortranOutputStack, 842
    showFortranOutputStack, 842
    topFortranOutputStack, 842
FORDER, 753
    order, 753
FormatArabic
    NUMFMT, 3122
FormatRoman
    NUMFMT, 3122
FORMULA1, 3788
    coerce, 3788
FORT, 845
    linkToFortran, 846
    outputAsFortran, 846
    setLegalFortranSourceExtensions, 846
FortranCodePackage1, 838
fortranCompilerName
    NAGSP, 2193
fortranLinkerArgs
    NAGSP, 2193
FortranOutputStackPackage, 842
FortranPackage, 845
foundZeroes
    RFP, 3781
    fprintINFO
        GBINTERN, 1077
FR2, 733
    map, 733
FRAC2, 867
    map, 867
FractionalIdealFunctions2, 849
FractionFreeFastGaussian, 852
FractionFreeFastGaussianFractions, 863
FractionFunctions2, 867
FramedNonAssociativeAlgebraFunctions2, 869
fresnelC
    DPDFUN, 369
fresnelS
    DPDFUN, 369
FRID1, 849
    map, 849
FRNAAF2, 869
    map, 869
Frobenius
    NORMRETR, 3108
froot
    POLYROOT, 3448
FRUTIL, 735
    mergeFactors, 735
    refine, 735
FS2, 897
    map, 897
FS2EXPXP, 913
    exprToXXP, 913
    localAbs, 913
FS2UPS, 925
    exprToGenUPS, 925
    exprToUPS, 925
    localAbs, 925
FSAGG2, 827
    map, 827
    reduce, 827
    scan, 827
FSCINT, 893
    complexIntegrate, 893
    internalIntegrate, 893
INDEX

internalIntegrate0, 893
FSINT, 899
    integrate, 899
FSPECIALC, 872
    abs, 872
    airyAi, 872
    airyBi, 872
    belong?, 872
    besselI, 872
    besselJ, 872
    besselK, 872
    besselY, 872
    Beta, 872
digamma, 872
    Gamma, 872
    iiabs, 872
    iiAiryAi, 872
    iiAiryBi, 872
    iiBesselI, 872
    iiBesselJ, 872
    iiBesselK, 872
    iiBesselY, 872
    iiBeta, 872
    iipolygamma, 872
    polygamma, 872
FSPECIALMELT, 903
    primitiveElement, 903
FSRED, 907
    bringDown, 907
    newReduc, 907
FSUPFACT, 942
    anfactor, 942
    ffactor, 942
    qfactor, 942
    UP2ifCan, 942
fullDesTree
    PAFF, 3215
    PAFFFF, 3223
fullInfClsPt
    PAFF, 3215
    PAFFFF, 3223
fullParamInit
    DTP, 324

function
    MKFUNC, 1593
    FunctionalSpecialFunction, 872
    FunctionFieldCategoryFunctions2, 881
    FunctionFieldIntegralBasis, 884
    functionIsContinuousAtEndPoints
        D01AGNT, 451
    functionIsFracPolynomial?
        ESCONT, 657
    functionIsOscillatory
        D01AGNT, 451
    FunctionSpaceAssertions, 888
    FunctionSpaceAttachPredicates, 891
    FunctionSpaceComplexIntegration, 893
    FunctionSpaceFunctions2, 897
    FunctionSpaceIntegration, 899
    FunctionSpacePrimitiveElement, 903
    FunctionSpaceReduce, 907
    FunctionSpaceSum, 910
    FunctionSpaceToExponentialExpansion, 913
    FunctionSpaceToUnivariatePowerSeries, 925
    FunctionSpaceUnivariatePolynomialFactor, 942

GALFACT, 953
    btwFact, 954
    degreePartition, 954
    eisensteinIrreducible?, 954
    factor, 954
    factorOfDegree, 954
    factorSquareFree, 954
    henselFact, 954
    makeFR, 954
    modularFactor, 954
    musserTrials, 954
    numberOfFactors, 954
    stopMusserTrials, 954
    tryFunctionalDecomposition, 954
    tryFunctionalDecomposition?, 954
    useEisensteinCriterion, 954
    useEisensteinCriterion?, 954
    useSingleFactorBound, 954
    useSingleFactorBound?, 954

GALFACTU, 948
    beauzamyBound, 948
    bombieriNorm, 948
    height, 948
infinityNorm, 948
length, 948
norm, 948
quadraticNorm, 948
rootBound, 948
singleFactorBound, 948
GaloisGroupFactorizationUtilities, 948
GaloisGroupFactorizer, 953
GaloisGroupPolynomialUtilities, 971
GaloisGroupUtilities, 975
GALPOLYU, 971
degreePartition, 971
factorOfDegree, 971
factorsOfDegree, 971
monic?, 971
reverse, 971
scaleRoots, 971
shiftRoots, 971
unvectorise, 971
GALUTIL, 975
fillPascalTriangle, 975
pascalTriangle, 975
rangePascalTriangle, 975
safeCeiling, 975
safeFloor, 975
safetyMargin, 975
sizePascalTriangle, 975
Gamma
DFSFUN, 369
FSPECF, 872
GAUSSFAC, 979
factor, 979
prime?, 979
sumSquares, 979
GaussianFactorizationPackage, 979
GB, 1114
groebner, 1114
normalForm, 1114
gbasis
GBINTERN, 1077
creditPol, 1077
critB, 1077
critBonD, 1077
critM, 1077
critMonD1, 1077
critMTonD1, 1077
crittpOrder, 1077
critT, 1077
fprindINFO, 1077
gbasis, 1077
hMonic, 1077
lepol, 1077
makeCrit, 1077
minGbasis, 1077
printb, 1077
printdINFO, 1077
printpolINFO, 1077
printshINFO, 1077
redPo, 1077
redPol, 1077
sPol, 1077
updatD, 1077
updatF, 1077
virtualDegree, 1077
gcd
HEUGCD, 1184
MDDFACT, 1719
PGCD, 3426
POLYVEC, 4144
PRS, 3527
gcdBasis
CADU, 311
gcdBasisAdd
CADU, 311
gcdcofact
HEUGCD, 1184
gcdcofactprim
HEUGCD, 1184
gcdPolynomial
GENPGCD, 1007
gedprim
HEUGCD, 1184
gedPrimitive
PGCD, 3426
gderiv
INDEX

STTAYLOR, 3863
GDRAW, 1043
  GospersMethod, 1043
GENEEZ, 1027
  compBound, 1027
  reduction, 1027
  solveid, 1027
  tablePow, 1027
  testModulus, 1027
generalCoefficient
  FFFG, 852
GeneralHenselPackage, 984
generalInfiniteProduct
  INPRODO, 1202
  INPRODFF, 1204
  INPRODPF, 1208
  STINPROD, 3859
generalInterpolation
  FFFG, 852
  FFFGF, 863
generalizedEigenvector
  EP, 516
generalizedEigenvectors
  EP, 516
GeneralizedMultivariateFactorize, 988
generalLambert
  STTAYLOR, 3863
GeneralPackageForAlgebraicFunctionField, 991
GeneralPolynomialGcdPackage, 1007
generalSqFr
  TWOFACT, 4078
generalTwoFactor
  TWOFACT, 4078
generate
  HB, 1180
generateIrredPoly
  IRREDFFX, 1390
GenerateUnivariatePowerSeries, 1021
genericPosition
  GROEBSOL, 1118
GenExEuclid, 1027
GENMFACT, 988
  factor, 988
GENPGCD, 1007
  gcdPolynomial, 1007
  randomR, 1007
GENUFAC, 1032
  factor, 1032
GenUFactorize, 1032
GENUPS, 1021
  laurent, 1021
  puiuex, 1021
  series, 1021
  taylor, 1021
genus
  DTP, 324
  GPAFF, 992
  PAFF, 3215
  PAFFFF, 3223
  genusNeg
  DTP, 324
  GPAFF, 992
  PAFF, 3215
  PAFFFF, 3223
  genusTree
  DTP, 324
  genusTreeNeg
  DTP, 324
GenusZeroIntegration, 1036
geometric
  RIDIST, 3616
getDatabase
  ASSOCEQ, 121
  OPQUERY, 3200
getEq
  RECOP, 3707
getGoodPrime
  PFOTOOLS, 3383
gethi
  D01AGNT, 451
  ESCONT, 657
  ESTOOLS, 667
getlo
  D01AGNT, 451
  ESCONT, 657
  ESTOOLS, 667
getOp
  RECOP, 3707
getOrder
  UDPO, 4135
getShiftRec
  RECOP, 3707
getIndex
UDVO, 4138
GF2FG
ITRIGMNP, 1275
GHENSEL, 984
completeHensel, 984
HenselLift, 984
reduction, 984
GnuDraw, 1043
goodnessOfFit
OPTPACK, 71
goodPoint
CHVAR, 188
goppaCode
PAFF, 3215
PAFFFF, 3223
GOSPER, 1046
GopserMethod, 1046
GopserMethod
GDRAW, 1043
GOSPER, 1046
GopserSummationMethod, 1046
GPAFF, 991
adjunctionDivisor, 992
classNumber, 992
desingTree, 992
desingTreeWoFullParam, 992
eval, 992
evalIfCan, 992
findOrderOfDivisor, 992
genus, 992
genusNeg, 992
homogenize, 992
interpolateForms, 992
interpolateFormsForFact, 992
intersectionDivisor, 992
IBasis, 992
LPolynomial, 992
numberOfPlacesOfDegree, 992
numberOfPlacesDegExtDeg, 992
numberOfRatPlacesExtDeg, 992
parametrize, 992
placesAbove, 992
placesOfDegree, 992
pointDominatesBy, 992
printInfo, 992
rationalPlaces, 992
rationalPoints, 992
reset, 992
setCurve, 992
setSingularPoints, 992
singularPoints, 992
theCurve, 992
ZetaFunction, 992
gradient
MCALCFN, 1768
graeffe
CRFP, 245
gramschmidt
REP, 3582
graphCurves
VIEW, 4168
GraphicsDefaults, 1052
graphs
CYCLES, 294
GRAPHVIZ, 1057
dot2eps, 1057
dotview, 1057
sampleDotGraph, 1057
standardDotHeader, 1057
writeDotGraph, 1057
Graphviz, 1057
GRAY, 1061
firstSubsetGray, 1061
nextSubsetGray, 1061
GrayCode, 1061
GRDEF, 1052
adaptive, 1052
clipPointsDefault, 1052
drawToScale, 1052
maxPoints, 1052
minPoints, 1052
screenResolution, 1052
groebgen
LGROBP, 1549
groebner
FGLMICPK, 750
GB, 1114
INTERGB, 1282
LEXTRIPK, 1517
GroebnerFactorizationPackage, 1068
groebnerFactorize
<table>
<thead>
<tr>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>GBF, 1069</td>
</tr>
<tr>
<td>GroebnerInternalPackage, 1077</td>
</tr>
<tr>
<td>GroebnerPackage, 1114</td>
</tr>
<tr>
<td>GroebnerSolve, 1118</td>
</tr>
<tr>
<td>GROEBSOL, 1118</td>
</tr>
<tr>
<td>genericPosition, 1118</td>
</tr>
<tr>
<td>groebSolve, 1118</td>
</tr>
<tr>
<td>testDim, 1118</td>
</tr>
<tr>
<td>groebSolve</td>
</tr>
<tr>
<td>GROEBSOL, 1118</td>
</tr>
<tr>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>diffHP, 1125</td>
</tr>
<tr>
<td>guess, 1125</td>
</tr>
<tr>
<td>guessADE, 1125</td>
</tr>
<tr>
<td>guessAlg, 1125</td>
</tr>
<tr>
<td>guessBinRat, 1125</td>
</tr>
<tr>
<td>guessExpRat, 1125</td>
</tr>
<tr>
<td>guessHolo, 1125</td>
</tr>
<tr>
<td>guessHP, 1125</td>
</tr>
<tr>
<td>guessPade, 1125</td>
</tr>
<tr>
<td>guessPRec, 1125</td>
</tr>
<tr>
<td>guessRat, 1125</td>
</tr>
<tr>
<td>guessRec, 1125</td>
</tr>
<tr>
<td>shiftHP, 1125</td>
</tr>
<tr>
<td>Guess, 1125</td>
</tr>
<tr>
<td>guess</td>
</tr>
<tr>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>GUESSAN, 1162</td>
</tr>
<tr>
<td>GUESSF, 1164</td>
</tr>
<tr>
<td>GUESSINT, 1169</td>
</tr>
<tr>
<td>GUESSP, 1171</td>
</tr>
<tr>
<td>GUESSUP, 1174</td>
</tr>
<tr>
<td>guessADE</td>
</tr>
<tr>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>GUESSAN, 1162</td>
</tr>
<tr>
<td>GUESSF, 1164</td>
</tr>
<tr>
<td>GUESSINT, 1169</td>
</tr>
<tr>
<td>GUESSP, 1171</td>
</tr>
<tr>
<td>GUESSUP, 1174</td>
</tr>
<tr>
<td>guessAlg</td>
</tr>
<tr>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>GUESSAN, 1162</td>
</tr>
<tr>
<td>GUESSF, 1164</td>
</tr>
<tr>
<td>GUESSINT, 1169</td>
</tr>
<tr>
<td>GUESSP, 1171</td>
</tr>
<tr>
<td>GUESSUP, 1174</td>
</tr>
<tr>
<td>guessAlgebraicNumber, 1162</td>
</tr>
<tr>
<td>GUESSAN, 1162</td>
</tr>
<tr>
<td>diffHP, 1162</td>
</tr>
<tr>
<td>guess, 1162</td>
</tr>
<tr>
<td>guessADE, 1162</td>
</tr>
<tr>
<td>guessAlg, 1162</td>
</tr>
<tr>
<td>guessBinRat, 1162</td>
</tr>
<tr>
<td>guessExpRat, 1162</td>
</tr>
<tr>
<td>guessHolo, 1162</td>
</tr>
<tr>
<td>guessHP, 1162</td>
</tr>
<tr>
<td>guessPade, 1162</td>
</tr>
<tr>
<td>guessPRec, 1162</td>
</tr>
<tr>
<td>guessRat, 1162</td>
</tr>
<tr>
<td>guessRec, 1162</td>
</tr>
<tr>
<td>shiftHP, 1162</td>
</tr>
<tr>
<td>guessBinRat</td>
</tr>
<tr>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>GUESSAN, 1162</td>
</tr>
<tr>
<td>GUESSF, 1164</td>
</tr>
<tr>
<td>GUESSINT, 1169</td>
</tr>
<tr>
<td>GUESSP, 1171</td>
</tr>
<tr>
<td>GUESSUP, 1174</td>
</tr>
<tr>
<td>guessExpRat</td>
</tr>
<tr>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>GUESSAN, 1162</td>
</tr>
<tr>
<td>GUESSF, 1164</td>
</tr>
<tr>
<td>GUESSINT, 1169</td>
</tr>
<tr>
<td>GUESSP, 1171</td>
</tr>
<tr>
<td>GUESSUP, 1174</td>
</tr>
<tr>
<td>GUESSF, 1164</td>
</tr>
<tr>
<td>diffHP, 1164</td>
</tr>
<tr>
<td>guess, 1164</td>
</tr>
<tr>
<td>guessADE, 1164</td>
</tr>
<tr>
<td>guessAlg, 1164</td>
</tr>
<tr>
<td>guessBinRat, 1164</td>
</tr>
<tr>
<td>guessExpRat, 1164</td>
</tr>
<tr>
<td>guessHolo, 1164</td>
</tr>
<tr>
<td>guessHP, 1164</td>
</tr>
<tr>
<td>guessPade, 1164</td>
</tr>
<tr>
<td>guessPRec, 1164</td>
</tr>
<tr>
<td>guessRat, 1164</td>
</tr>
<tr>
<td>guessRec, 1164</td>
</tr>
<tr>
<td>shiftHP, 1164</td>
</tr>
<tr>
<td>GUESSF1, 1166</td>
</tr>
<tr>
<td>F2EXPRR, 1166</td>
</tr>
<tr>
<td>GuessFinite, 1164</td>
</tr>
</tbody>
</table>
GuessFiniteFunctions, 1166
guessHolo
  GUESS, 1125
  GUESSAN, 1162
  GUESSF, 1164
  GUESSINT, 1169
  GUESSP, 1171
  GUESSUP, 1174
guessHP
  GUESS, 1125
  GUESSAN, 1162
  GUESSF, 1164
  GUESSINT, 1169
  GUESSP, 1171
  GUESSUP, 1174
GUESSINT, 1169
diffHP, 1169
guess, 1169
guessADE, 1169
guessAlg, 1169
guessBinRat, 1169
guessExpRat, 1169
guessHolo, 1169
guessHP, 1169
guessPade, 1169
guessPRec, 1169
guessRat, 1169
guessRec, 1169
shiftHP, 1169
GuessInteger, 1169
GUESSP, 1171
diffHP, 1171
guess, 1171
guessADE, 1171
guessAlg, 1171
guessBinRat, 1171
guessExpRat, 1171
guessHolo, 1171
guessHP, 1171
guessPade, 1171
guessPRec, 1171
guessRat, 1171
guessRec, 1171
shiftHP, 1171
guessPade
  GUESS, 1125
GUESSAN, 1162
GUESSF, 1164
GUESSINT, 1169
GUESSP, 1171
GUESSUP, 1174
GuessPolynomial, 1171
guessPRec
  GUESS, 1125
  GUESSAN, 1162
  GUESSF, 1164
  GUESSINT, 1169
  GUESSP, 1171
  GUESSUP, 1174
guessRat
  GUESS, 1125
  GUESSAN, 1162
  GUESSF, 1164
  GUESSINT, 1169
  GUESSP, 1171
  GUESSUP, 1174
guessRec
  GUESS, 1125
  GUESSAN, 1162
  GUESSF, 1164
  GUESSINT, 1169
  GUESSP, 1171
  GUESSUP, 1174
GuessUnivariatePolynomial, 1174
GUESSUP, 1174
diffHP, 1174
guess, 1174
guessADE, 1174
guessAlg, 1174
guessBinRat, 1174
guessExpRat, 1174
guessHolo, 1174
guessHP, 1174
guessPade, 1174
guessPRec, 1174
guessRat, 1174
guessRec, 1174
shiftHP, 1174
hadamard
  UFPS1, 4094
HallBasis, 1180
INDEX

harmonic
  INTHEORY, 1320
hasoln
  PLEQN, 3257
hasSolution?
  LSMP, 1540
  LSMP1, 1543
Hausdorff
  XEXPPKG, 4184
HB, 1180
generate, 1180
inHallBasis?, 1180
lfunc, 1180
hdmToDmp
  POLTOPOL, 3390
hdmToP
  POLTOPOL, 3390
heapSort
  FLASORT, 823
height
  GALFACTU, 948
henselFact
  GALFACT, 954
  UNIFACT, 4086
HenselLift
  GHENSEL, 984
hermite
  PNTHEORY, 3442
  SMITH, 3799
hermiteH
  ORTHPOL, 3207
HermiteIntegrate
  INITERAL, 21
  INTHERTR, 3998
hessian
  MCALCFN, 1768
HEUGCD, 1184
  content, 1184
  gcd, 1184
  gcdcofact, 1184
  gcdcofactprim, 1184
  gcdprim, 1184
  lintgcd, 1184
HenGcd, 1184
hMonic
  GBINTERN, 1077
homogenize
  GPAFF, 992
  PAFF, 3215
  PAFFFE, 3223
  PFORP, 3232
horizConcat
  MAMA, 1703
horizSplit
  MAMA, 1703
htrigs
  TRMANIP, 4012
hue
  PTPACK, 3370
hypergeometric0F1
  DFSFUN, 369
IALGFACT, 1217
  factor, 1217
IBACHIN, 197
  chineseRemainder, 197
  factorList, 197
  listConjugateBases, 197
IBATOOL, 1347
  diagonalProduct, 1347
  divideIfCan, 1347
  idealiser, 1347
  idealiserMatrix, 1347
  leastPower, 1347
  matrixGcd, 1347
  moduleSum, 1347
IBPTOOLS, 1352
  mapBivariate, 1352
  mapMatrixIfCan, 1352
  mapUnivariate, 1352
  mapUnivariateIfCan, 1352
ICDEN, 1220
  clearDenominator, 1220
  commonDenominator, 1220
  splitDenominator, 1220
id
  MAPPKG1, 1614
IdealDecompositionPackage, 1190
idealiser
  IBATOOL, 1347
idealiserMatrix
  IBATOOL, 1347
INDEX

IDECOMP, 1190  
  contract, 1190  
  primaryDecomp, 1190  
  prime?, 1190  
  radical, 1190  
  zeroDimPrimary?, 1190  
  zeroDimPrime?, 1190  
identitySquareMatrix  
  FCPAK1, 838  
iflist2Result  
  ESTOOLS, 667  
ignore?  
  DFIINTTLS, 314  
iabs  
  FSPECF, 872  
iacos  
  EF, 535  
iacosh  
  EF, 535  
iacot  
  EF, 535  
iacoth  
  EF, 535  
iacsc  
  EF, 535  
iacsch  
  EF, 535  
iAiryAi  
  FSPECF, 872  
iAiryBi  
  FSPECF, 872  
iasec  
  EF, 535  
iasech  
  EF, 535  
iasin  
  EF, 535  
iasinh  
  EF, 535  
iatan  
  EF, 535  
iatanh  
  EF, 535  
iBesselI  
  FSPECF, 872  
iBesselJ  
  FSPECF, 872  
inBesselI  
  FSPECF, 872  
inBesselY  
  FSPECF, 872  
inBeta  
  FSPECF, 872  
inbinom  
  COMBF, 208  
inacos  
  EF, 535  
inacosh  
  EF, 535  
inacoth  
  EF, 535  
inacoth  
  EF, 535  
inasec  
  EF, 535  
inasech  
  EF, 535  
inasin  
  EF, 535  
inasinh  
  EF, 535  
inatan  
  EF, 535  
inatanh  
  EF, 535  
inBesselI  
  FSPECF, 872  
inBesselJ  
  FSPECF, 872  
iiBesselK  
  FSPECF, 872  
iiBesselY  
  FSPECF, 872  
iiBeta  
  FSPECF, 872  
iibinom  
  COMBF, 208  
iiacos  
  EF, 535  
iiacos  
  EF, 535  
iiacoth  
  EF, 535  
iiAtan  
  EF, 535  
iiAtan  
  EF, 535  
iiAtan  
  EF, 535  
iiAtan  
  EF, 535  
iidigamma  
  FSPECF, 872  
iidprod  
  COMBF, 208  
iidsum  
  COMBF, 208  
iidexp  
  EF, 535  
iidfact  
  COMBF, 208  
iiGamma  
  FSPECF, 872  
iilog  
  EF, 535  
iiperm  
  COMBF, 208  
iipolygamma  
  FSPECF, 872  
iipow  
  COMBF, 208  
iisec  
  EF, 535  
iisec  
  EF, 535  
iisin  
  EF, 535
<table>
<thead>
<tr>
<th>Index</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>iisinh</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>iiqrt2</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>iiqrt3</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>iitan</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>iitanh</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>imag</td>
<td></td>
</tr>
<tr>
<td>CTRIGMNP, 261</td>
<td></td>
</tr>
<tr>
<td>TRIGMNP, 4062</td>
<td></td>
</tr>
<tr>
<td>images</td>
<td></td>
</tr>
<tr>
<td>AF, 14</td>
<td></td>
</tr>
<tr>
<td>AFALGGRO, 4</td>
<td></td>
</tr>
<tr>
<td>AFALGRES, 9</td>
<td></td>
</tr>
<tr>
<td>ALGFACT, 55</td>
<td></td>
</tr>
<tr>
<td>ALGMANIP, 35</td>
<td></td>
</tr>
<tr>
<td>ALGMFACT, 41</td>
<td></td>
</tr>
<tr>
<td>ALGPKG, 44</td>
<td></td>
</tr>
<tr>
<td>ANY1, 98</td>
<td></td>
</tr>
<tr>
<td>API, 106</td>
<td></td>
</tr>
<tr>
<td>APPLYORE, 113</td>
<td></td>
</tr>
<tr>
<td>APPRULE, 109</td>
<td></td>
</tr>
<tr>
<td>ARRAY12, 3189</td>
<td></td>
</tr>
<tr>
<td>ASSOCEQ, 115</td>
<td></td>
</tr>
<tr>
<td>AXSERV, 121</td>
<td></td>
</tr>
<tr>
<td>BALFACT, 138</td>
<td></td>
</tr>
<tr>
<td>BEZIER, 162</td>
<td></td>
</tr>
<tr>
<td>BEZOUT, 165</td>
<td></td>
</tr>
<tr>
<td>BLUPPACK, 170</td>
<td></td>
</tr>
<tr>
<td>BOP1, 141</td>
<td></td>
</tr>
<tr>
<td>BOUNDRZRO, 177</td>
<td></td>
</tr>
<tr>
<td>BRILL, 180</td>
<td></td>
</tr>
<tr>
<td>CAD, 306</td>
<td></td>
</tr>
<tr>
<td>CADU, 311</td>
<td></td>
</tr>
<tr>
<td>CARTEN2, 186</td>
<td></td>
</tr>
<tr>
<td>CDEN, 221</td>
<td></td>
</tr>
<tr>
<td>CHARPOL, 195</td>
<td></td>
</tr>
<tr>
<td>CHVAR, 188</td>
<td></td>
</tr>
<tr>
<td>CINTSLPE, 237</td>
<td></td>
</tr>
<tr>
<td>CLIP, 4071</td>
<td></td>
</tr>
<tr>
<td>CMPLXRT, 258</td>
<td></td>
</tr>
<tr>
<td>COMBF, 208</td>
<td></td>
</tr>
<tr>
<td>COMBINAT, 1289</td>
<td></td>
</tr>
<tr>
<td>COMMONOP, 223</td>
<td></td>
</tr>
<tr>
<td>COMMUPC, 229</td>
<td></td>
</tr>
<tr>
<td>COMPFAC2, 231</td>
<td></td>
</tr>
<tr>
<td>COMPLEX2, 235</td>
<td></td>
</tr>
<tr>
<td>COMPLPAT, 239</td>
<td></td>
</tr>
<tr>
<td>COORDSYS, 268</td>
<td></td>
</tr>
<tr>
<td>CPIMA, 193</td>
<td></td>
</tr>
<tr>
<td>CPMATCH, 241</td>
<td></td>
</tr>
<tr>
<td>CRAPACK, 273</td>
<td></td>
</tr>
<tr>
<td>CRFP, 244</td>
<td></td>
</tr>
<tr>
<td>CSTTOOLS, 300</td>
<td></td>
</tr>
<tr>
<td>CTRIGMNP, 260</td>
<td></td>
</tr>
<tr>
<td>CVMP, 202</td>
<td></td>
</tr>
<tr>
<td>CYCLES, 294</td>
<td></td>
</tr>
<tr>
<td>CYCLOTO, 303</td>
<td></td>
</tr>
<tr>
<td>D01AGNT, 451</td>
<td></td>
</tr>
<tr>
<td>D01WGETS, 458</td>
<td></td>
</tr>
<tr>
<td>D02AGNT, 490</td>
<td></td>
</tr>
<tr>
<td>D03AGNT, 512</td>
<td></td>
</tr>
<tr>
<td>DBLRESP, 411</td>
<td></td>
</tr>
<tr>
<td>DDFACT, 349</td>
<td></td>
</tr>
<tr>
<td>DEFINTEF, 554</td>
<td></td>
</tr>
<tr>
<td>DEFINTRF, 3628</td>
<td></td>
</tr>
<tr>
<td>DEGRED, 321</td>
<td></td>
</tr>
<tr>
<td>DFINTT, 314</td>
<td></td>
</tr>
<tr>
<td>DFSFUN, 309</td>
<td></td>
</tr>
<tr>
<td>DIOSP, 334</td>
<td></td>
</tr>
<tr>
<td>DIRPROD2, 339</td>
<td></td>
</tr>
<tr>
<td>DISPLAY, 345</td>
<td></td>
</tr>
<tr>
<td>DLP, 342</td>
<td></td>
</tr>
<tr>
<td>DRAW, 3965</td>
<td></td>
</tr>
<tr>
<td>DRAWCFUN, 3978</td>
<td></td>
</tr>
<tr>
<td>DRAWCURV, 3974</td>
<td></td>
</tr>
<tr>
<td>DRAWCX, 414</td>
<td></td>
</tr>
<tr>
<td>DRAWHACK, 419</td>
<td></td>
</tr>
<tr>
<td>DRAWPT, 3993</td>
<td></td>
</tr>
<tr>
<td>DROPT0, 421</td>
<td></td>
</tr>
<tr>
<td>DROPT1, 426</td>
<td></td>
</tr>
<tr>
<td>DTP, 324</td>
<td></td>
</tr>
<tr>
<td>E04AGNT, 721</td>
<td></td>
</tr>
<tr>
<td>EF, 535</td>
<td></td>
</tr>
<tr>
<td>EFSTRUC, 579</td>
<td></td>
</tr>
<tr>
<td>ELFUTS, 611</td>
<td></td>
</tr>
<tr>
<td>EP, 516</td>
<td></td>
</tr>
<tr>
<td>EQ2, 614</td>
<td></td>
</tr>
</tbody>
</table>
INDEX

ERROR, 617
ES1, 686
ES2, 688
ESCONT, 657
ESCONT1, 664
ESTOOLS, 667
ESTOOLS1, 676
ESTOOLS2, 678
EVALCYC, 655
EXP3D, 717
EXPR2, 680
EXPR2UPS, 704
EXPRODE, 691
EXPRSOL, 683
EXPRTUBE, 712
FACTEXT, 741
FACTFUNC, 728
FACTRN, 746
FACUTIL, 738
FAMR2, 756
FCPAK1, 838
FDIV2, 758
FFCAT2, 881
FFF, 772
FFFACTOR, 760
FFFACSE, 766
FFFG, 852
FFFGF, 863
FFHOM, 779
FFINTBAS, 884
FFPOLY, 788
FFPOLY2, 809
FFSLPE, 813
FFSQFR, 816
FGLMICPK, 750
FLAG2, 819
FLASORT, 823
FLOATCP, 830
FLOATRP, 834
FOP, 842
FORDER, 753
FORMULAl, 3788
FORT, 846
FR2, 733
FRAC2, 867
FRIDEAL2, 849
FRNAAF2, 869
FRUTIL, 735
FS2, 897
FS2EXPXP, 913
FS2UPS, 925
FSAGG2, 827
FSINT, 893
FSINT, 899
FSPECF, 872
FSPRMELT, 903
FSRED, 907
FSUPFACT, 942
GALFACT, 953
GALFACTU, 948
GALPOLYU, 971
GALUTIL, 975
GAUSSFAC, 979
GB, 1114
GBEUCLID, 642
GBF, 1068
GBINTERN, 1077
GDRAW, 1043
GENEEZ, 1027
GENMFACT, 988
GENPGCD, 1007
GENUFACT, 1032
GENUPS, 1021
GHENSEL, 984
GOSPER, 1046
GPAFF, 991
GRAPHVIZ, 1057
GRAY, 1061
GRDEF, 1052
GROESBOL, 1118
GUESS, 1125
GUESSAN, 1162
GUESSF, 1164
GUESSF1, 1166
GUESSINT, 1169
GUESSP, 1171
GUESSUP, 1174
HB, 1180
HEUGCD, 1184
IALGFAC, 1217
IBACHIN, 197
IBATOOL, 1347
INDEX

IBPTOOLS, 1352
ICDEN, 1220
IDECOMP, 1190
IMATLIN, 1223
IMATQF, 1229
INBFF, 1249
INCRMAPS, 1199
INEP, 1258
INFINITY, 1215
INFORM1, 1280
INPRODO, 1201
INFSF, 1264
INMODGCD, 1232
INNMFACT, 1239
INPRODF, 1204
INPRODF, 1208
INPSIGN, 1269
INTAF, 32
INTALG, 24
INTBIT, 1285
INTDIVP, 1387
INTEF, 589
INTERGB, 1282
INTFACT, 1293
INTFRSP, 1379
INTG0, 1036
INTHEORY, 1320
INTHERAL, 21
INTHERTR, 3998
INTPACK, 59
INTPAF, 3547
INTPM, 3306
INTRAD, 3655
INTRET, 1337
INTRF, 3636
INTSLPE, 1344
INTTOOLS, 1367
INTTR, 4001
INVLAPLA, 1401
IPRTPK, 1371
IR2, 1355
IR2F, 1361
IROOT, 1339
IRREDFFX, 1390
IRRF2F, 1358
IRSN, 1394
IRURPK, 1374
ISUMP, 1272
ITFUN2, 1210
ITFUN3, 1212
ITRIGMNP, 1275
KERNEL2, 1408
KOVACIC, 1410
LAPLACE, 1414
LAZ3PK, 1440
LEADCDET, 1444
LEXTRIPK, 1517
LF, 1560
LGROBP, 1549
LIMITPS, 3485
LIMITRF, 3639
LINDEP, 1523
LIST2, 1566
LIST2MAP, 1571
LIST3, 1569
LISYSER, 1536
LODEEF, 560
LODOFO, 1526
LODOOPS, 1530
LOP, 1557
LPARSPT, 1575
LPEFRAC, 1534
LSMP, 1539
LSMP1, 1543
LSPP, 1546
MAMA, 1703
MAPHACK1, 1600
MAPHACK2, 1602
MAPHACK3, 1604
MAPPKG1, 1614
MAPPKG2, 1625
MAPPKG3, 1636
MAPPKG4, 1643
MATCAT2, 1646
MATLIN, 1652
MATSTOR, 3845
MCALCF, 1768
MCDEN, 1649
MDDFACT, 1719
MESH, 1715
MFINFACT, 1754
MHROWRED, 1725
| MKBCFUNC, 1582 | NSUP2, 3075 |
| MKFLCFN, 1585 | NTPOLFN, 3127 |
| MKFUNC, 1593 | NUMERIC, 3131 |
| MKRECORD, 1595 | NUMFMT, 3122 |
| MKUCFUNC, 1597 | NUMODE, 3144 |
| MMAP, 1766 | NUMQUAD, 3154 |
| MONOTOOL, 1734 | NUMTUBE, 3175 |
| MPC2, 1747 | OCTC2T2, 3180 |
| MPC3, 1749 | ODECONST, 264 |
| MPCPF, 1739 | ODEEF, 567 |
| MPREF, 1742 | ODEINT, 3182 |
| MRAFFAC, 1752 | ODEPACK, 81 |
| MRF2, 1732 | ODEPAK, 3556 |
| MSYSCMD, 1737 | ODEPRIM, 3507 |
| MTHING, 1712 | ODERAT, 3662 |
| MULTFACT, 1773 | ODERED, 3722 |
| MULTSQFR, 1782 | ODERTRIC, 3671 |
| NAGC02, 2812 | ODESYS, 3913 |
| NAGC05, 2836 | ODETOL, 3186 |
| NAGC06, 2901 | OMEXPR, 696 |
| NAGD01, 2272 | OMPKG, 3194 |
| NAGD02, 2745 | OMSERVER, 3197 |
| NAGD03, 2792 | ONECOMP2, 3191 |
| NAGE01, 2339 | OPQUERY, 3200 |
| NAGE02, 2074 | OPTPAKK, 71 |
| NAGE04, 2646 | ORDCOMP2, 3202 |
| NAGF01, 2452 | ORDFUNS, 3204 |
| NAGF02, 1889 | ORPCTO, 4122 |
| NAGF04, 2183 | ORTHPOL, 3207 |
| NAGF07, 2367 | OUT, 3211 |
| NAGS, 3058 | PADE, 3242 |
| NAGSP, 2193 | PADEPAC, 3240 |
| NCEP, 3166 | PAFF, 3215 |
| NCNTFRAC, 3170 | PAFFF, 3223 |
| NCODIV, 3086 | PAN2EXPR, 3396 |
| NEWTON, 3077 | PARAMEXP, 3277 |
| NFINTBAS, 3116 | PARPC2, 3271 |
| NLINSOL, 3096 | PARSC2, 3273 |
| NODE1, 3091 | PARSU2, 3275 |
| NONE1, 3089 | PARTPERM, 3285 |
| NORMMA, 3105 | PATMATCH, 3294 |
| NORMPK, 3100 | PATRES2, 3330 |
| NORMRETR, 3108 | PATTERN, 3289 |
| NPCOEF, 3111 | PATERN2, 3292 |
| NPOYTHON, 3080 | PCOMP, 3405 |
| NREP, 3172 |
INDEX

PDECOMP, 3407
PDEPACK, 91
PERMAN, 3340
PFBR, 3410
PFBRU, 3418
PFO, 3373
PFOQ, 3380
PFORP, 3232
PFOTOOLS, 3383
PFPC, 3282
PGCD, 3426
PGE, 3346
PGROEB, 3393
PICOERCE, 3355
PINTERP, 3437
PINTERPA, 3440
PLEQN, 3257
PLOT1, 3357
PLOTTOOL, 3359
PLPKCRV, 3386
PMASS, 3297
PMASSFS, 888
PMDOWN, 3324
PMFS, 3300
PMINS, 3303
PMKERNEL, 3314
PMLSAGG, 3318
PMPLCAT, 3320
PMPRED, 119
PMMPREDFS, 891
PMQFCAT, 3327
PMSYM, 3332
PMTOOLS, 3334
PNTHEORY, 3442
POLTOPOL, 3390
POLUTIL, 3684
POLY2, 3424
POLY2UP, 3482
POLYCATQ, 3401
POLYLIFT, 3398
POLYROOT, 3448
POLYVEC, 4144
PREASSOC, 3497
PRIMARR2, 3500
PRIMELT, 3503
PRIMES, 1326

PRINT, 3519
PRJALGPK, 3362
PRS, 3526
PSETPK, 3453
PSEUDLIN, 3521
PSQFR, 3478
PTFUNC2, 3367
PTPACK, 3369
PUSHVAR, 3559
PWFFINTB, 3247
QALGSET2, 3563
QCMPACK, 3567
QFCAT2, 3576
QUATCT2, 3579
RADUTIL, 3608
RANDSRC, 3619
RATFACT, 3622
RATRET, 3669
RDEEF, 599
RDEEFS, 608
RDETR, 4027
RDETRS, 4032
RDIST, 3610
RDIV, 3719
REAL0, 3693
REAL0Q, 3701
REALSOLV, 3690
RECOP, 3707
REDORDER, 3725
REP, 3582
REP1, 3750
REP2, 3758
REPDB, 3744
REPSQ, 3747
RESLATC, 3775
RETSOL, 3777
RF, 3625
RFDIST, 3613
RFFACT, 3631
RFFACTOR, 3634
RFP, 3781
RIDIST, 3616
RINTERP, 3658
RMUCAT2, 3704
RSDCMPK, 3728
RSETGCD, 3736
<table>
<thead>
<tr>
<th>Index Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>RURPK</td>
<td>3679</td>
</tr>
<tr>
<td>SAEFACT</td>
<td>3795</td>
</tr>
<tr>
<td>SAERFFC</td>
<td>3786</td>
</tr>
<tr>
<td>SCACHE</td>
<td>3805</td>
</tr>
<tr>
<td>SCPKG</td>
<td>3893</td>
</tr>
<tr>
<td>SEG2</td>
<td>3792</td>
</tr>
<tr>
<td>SEGBIND2</td>
<td>3790</td>
</tr>
<tr>
<td>SFQCMPK</td>
<td>3816</td>
</tr>
<tr>
<td>SFSGCD</td>
<td>3834</td>
</tr>
<tr>
<td>SGCF</td>
<td>3927</td>
</tr>
<tr>
<td>SHP</td>
<td>3897</td>
</tr>
<tr>
<td>SIGNEF</td>
<td>574</td>
</tr>
<tr>
<td>SIGNRF</td>
<td>3644</td>
</tr>
<tr>
<td>SIMPAN</td>
<td>3797</td>
</tr>
<tr>
<td>SMITH</td>
<td>3799</td>
</tr>
<tr>
<td>SOLVEFOR</td>
<td>3471</td>
</tr>
<tr>
<td>SOLVERAD</td>
<td>3600</td>
</tr>
<tr>
<td>SOLVESER</td>
<td>4055</td>
</tr>
<tr>
<td>SOLVETRA</td>
<td>4043</td>
</tr>
<tr>
<td>SORTPAK</td>
<td>3808</td>
</tr>
<tr>
<td>SPECOUT</td>
<td>3813</td>
</tr>
<tr>
<td>SRDCMPK</td>
<td>3827</td>
</tr>
<tr>
<td>STINPROD</td>
<td>3859</td>
</tr>
<tr>
<td>STNSR</td>
<td>3873</td>
</tr>
<tr>
<td>STREAM1</td>
<td>3851</td>
</tr>
<tr>
<td>STREAM2</td>
<td>3853</td>
</tr>
<tr>
<td>STREAM3</td>
<td>3856</td>
</tr>
<tr>
<td>STTAYLOR</td>
<td>3862</td>
</tr>
<tr>
<td>STTF</td>
<td>3875</td>
</tr>
<tr>
<td>STTFNC</td>
<td>3886</td>
</tr>
<tr>
<td>SUBRESP</td>
<td>3906</td>
</tr>
<tr>
<td>SUMFS</td>
<td>910</td>
</tr>
<tr>
<td>SUMRF</td>
<td>3652</td>
</tr>
<tr>
<td>SUP2</td>
<td>3811</td>
</tr>
<tr>
<td>SUPFRACF</td>
<td>3910</td>
</tr>
<tr>
<td>SYMFUNC</td>
<td>3938</td>
</tr>
<tr>
<td>SYSSOLP</td>
<td>3920</td>
</tr>
<tr>
<td>TABLBUMP</td>
<td>3942</td>
</tr>
<tr>
<td>TANEXP</td>
<td>3951</td>
</tr>
<tr>
<td>TBCMPPK</td>
<td>3947</td>
</tr>
<tr>
<td>TEMUTL</td>
<td>3958</td>
</tr>
<tr>
<td>TEX1</td>
<td>3960</td>
</tr>
<tr>
<td>TOOLSIGN</td>
<td>3962</td>
</tr>
<tr>
<td>TOPSP</td>
<td>3996</td>
</tr>
<tr>
<td>TRIGMNIP</td>
<td>4062</td>
</tr>
<tr>
<td>TRIMAT</td>
<td>4059</td>
</tr>
<tr>
<td>TRMANIP</td>
<td>4012</td>
</tr>
<tr>
<td>TUBETOOL</td>
<td>4067</td>
</tr>
<tr>
<td>TWOFACT</td>
<td>4078</td>
</tr>
<tr>
<td>UDPO</td>
<td>4135</td>
</tr>
<tr>
<td>UDV0</td>
<td>4138</td>
</tr>
<tr>
<td>UFPS1</td>
<td>4094</td>
</tr>
<tr>
<td>ULS2</td>
<td>4096</td>
</tr>
<tr>
<td>UNIFACT</td>
<td>4086</td>
</tr>
<tr>
<td>UNISEG2</td>
<td>4132</td>
</tr>
<tr>
<td>UP2</td>
<td>4110</td>
</tr>
<tr>
<td>UPCDEN</td>
<td>4101</td>
</tr>
<tr>
<td>UPDECOMP</td>
<td>4103</td>
</tr>
<tr>
<td>UPDIVP</td>
<td>4107</td>
</tr>
<tr>
<td>UPMP</td>
<td>4112</td>
</tr>
<tr>
<td>UPOLYC2</td>
<td>4098</td>
</tr>
<tr>
<td>UPSQFREE</td>
<td>4115</td>
</tr>
<tr>
<td>UPXS2</td>
<td>4119</td>
</tr>
<tr>
<td>UTS2</td>
<td>4126</td>
</tr>
<tr>
<td>UTSODE</td>
<td>4128</td>
</tr>
<tr>
<td>UTSODETL</td>
<td>4141</td>
</tr>
<tr>
<td>UTSSOL</td>
<td>3954</td>
</tr>
<tr>
<td>VECTOR2</td>
<td>4158</td>
</tr>
<tr>
<td>VIEW</td>
<td>4168</td>
</tr>
<tr>
<td>VIEWDEF</td>
<td>4161</td>
</tr>
<tr>
<td>WEIER</td>
<td>4172</td>
</tr>
<tr>
<td>WFFINTBS</td>
<td>4177</td>
</tr>
<tr>
<td>XEXPPKG</td>
<td>4184</td>
</tr>
<tr>
<td>YSTREAM</td>
<td>3253</td>
</tr>
<tr>
<td>ZDSOLVE</td>
<td>4253</td>
</tr>
<tr>
<td>ZLINDEP</td>
<td>1304</td>
</tr>
<tr>
<td>IMATLIN</td>
<td>1223</td>
</tr>
<tr>
<td>IMATQF</td>
<td>1229</td>
</tr>
<tr>
<td>inverse</td>
<td>1229</td>
</tr>
<tr>
<td>nullSpace</td>
<td>1229</td>
</tr>
<tr>
<td>rowEchelon</td>
<td>1229</td>
</tr>
<tr>
<td>in?</td>
<td>ESCONT1, 664</td>
</tr>
<tr>
<td>ESTOOLS</td>
<td>667</td>
</tr>
<tr>
<td>inBetweenExcpDiv</td>
<td>DTP, 324</td>
</tr>
<tr>
<td>INBFF</td>
<td>1249</td>
</tr>
<tr>
<td>/**?,</td>
<td>1249</td>
</tr>
<tr>
<td>?*,</td>
<td>1249</td>
</tr>
<tr>
<td>?/?,</td>
<td>1249</td>
</tr>
<tr>
<td>basis</td>
<td>1249</td>
</tr>
<tr>
<td>dAndcExp</td>
<td>1249</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>INEP</td>
<td>1258</td>
</tr>
<tr>
<td>innerint</td>
<td>DEFINTEF, 554</td>
</tr>
<tr>
<td>InnerMatrixLinearAlgebraFunctions</td>
<td>1223</td>
</tr>
<tr>
<td>InnerMatrixQuotientFieldFunctions</td>
<td>1229</td>
</tr>
<tr>
<td>InnerModularGcd</td>
<td>1232</td>
</tr>
<tr>
<td>InnerMultFact</td>
<td>1239</td>
</tr>
<tr>
<td>InnerNormalBasisFieldFunctions</td>
<td>1249</td>
</tr>
<tr>
<td>InnerNumericEigenPackage</td>
<td>1258</td>
</tr>
<tr>
<td>InnerNumericFloatSolvePackage</td>
<td>1264</td>
</tr>
<tr>
<td>InnerPolySign</td>
<td>1269</td>
</tr>
<tr>
<td>InnerPolySum</td>
<td>1272</td>
</tr>
<tr>
<td>innerSolve</td>
<td>INFSP, 1264</td>
</tr>
<tr>
<td>innerSolve1</td>
<td>INFSP, 1264</td>
</tr>
<tr>
<td>InnerTrigonometricManipulations</td>
<td>1275</td>
</tr>
<tr>
<td>INNMFACT</td>
<td>1239</td>
</tr>
<tr>
<td>factor</td>
<td>1239</td>
</tr>
<tr>
<td>INPRODFF</td>
<td>1204</td>
</tr>
<tr>
<td>evenInfiniteProduct</td>
<td>1204</td>
</tr>
<tr>
<td>generalInfiniteProduct</td>
<td>1204</td>
</tr>
<tr>
<td>infiniteProduct</td>
<td>1204</td>
</tr>
<tr>
<td>oddInfiniteProduct</td>
<td>1204</td>
</tr>
<tr>
<td>INPRODPF</td>
<td>1208</td>
</tr>
<tr>
<td>evenInfiniteProduct</td>
<td>1208</td>
</tr>
<tr>
<td>generalInfiniteProduct</td>
<td>1208</td>
</tr>
<tr>
<td>infiniteProduct</td>
<td>1208</td>
</tr>
<tr>
<td>oddInfiniteProduct</td>
<td>1208</td>
</tr>
<tr>
<td>INPSIGN</td>
<td>1269</td>
</tr>
<tr>
<td>signAround</td>
<td>1269</td>
</tr>
<tr>
<td>InputFormFunctions1</td>
<td>1280</td>
</tr>
<tr>
<td>inrootof</td>
<td>AF, 14</td>
</tr>
<tr>
<td>int</td>
<td>ODEINT, 3182</td>
</tr>
<tr>
<td>STTAYLOR</td>
<td>3863</td>
</tr>
<tr>
<td>INTAF</td>
<td>32</td>
</tr>
<tr>
<td>alginnt</td>
<td>32</td>
</tr>
<tr>
<td>INTALG</td>
<td>24</td>
</tr>
<tr>
<td>algintegrate</td>
<td>24</td>
</tr>
<tr>
<td>palginfieldint</td>
<td>24</td>
</tr>
<tr>
<td>palgintegrate</td>
<td>24</td>
</tr>
<tr>
<td>INTBIT</td>
<td>1285</td>
</tr>
<tr>
<td>bitCoef</td>
<td>1285</td>
</tr>
<tr>
<td>bitLength</td>
<td>1285</td>
</tr>
<tr>
<td>bitTruth</td>
<td>1285</td>
</tr>
<tr>
<td>intChoose</td>
<td>MULTSQFR, 1782</td>
</tr>
<tr>
<td>intcompBasis</td>
<td>LGROBP, 1549</td>
</tr>
<tr>
<td>INTDIVP</td>
<td>1387</td>
</tr>
<tr>
<td>intersectionDivisor</td>
<td>1387</td>
</tr>
<tr>
<td>placesOfDegree</td>
<td>1387</td>
</tr>
<tr>
<td>INTEF</td>
<td>589</td>
</tr>
<tr>
<td>Ifextendedint</td>
<td>589</td>
</tr>
<tr>
<td>Ifextlimit</td>
<td>589</td>
</tr>
<tr>
<td>lfinfieldint</td>
<td>589</td>
</tr>
<tr>
<td>lffieldintegrate</td>
<td>589</td>
</tr>
<tr>
<td>llimitedint</td>
<td>589</td>
</tr>
<tr>
<td>integer</td>
<td>INTRET, 1337</td>
</tr>
<tr>
<td>Integer?</td>
<td>INTRET, 1337</td>
</tr>
<tr>
<td>IntegerBits</td>
<td>1285</td>
</tr>
<tr>
<td>integerBound</td>
<td>BOUNDZRO, 177</td>
</tr>
<tr>
<td>IntegerCombinatoricFunctions</td>
<td>1289</td>
</tr>
<tr>
<td>IntegerFactorizationPackage</td>
<td>1293</td>
</tr>
<tr>
<td>IntegerIfCan</td>
<td>INTRET, 1337</td>
</tr>
<tr>
<td>IntegerLinearDependence</td>
<td>1304</td>
</tr>
<tr>
<td>IntegerNumberTheoryFunctions</td>
<td>1320</td>
</tr>
<tr>
<td>IntegerPrimesPackage</td>
<td>1326</td>
</tr>
<tr>
<td>IntegerRetractions</td>
<td>1337</td>
</tr>
<tr>
<td>IntegerRoots</td>
<td>1339</td>
</tr>
<tr>
<td>integers</td>
<td>STTAYLOR, 3863</td>
</tr>
<tr>
<td>IntegerSolveLinearPolynomialEquation</td>
<td>1344</td>
</tr>
<tr>
<td>integral</td>
<td>LF, 1560</td>
</tr>
<tr>
<td>integralBasis</td>
<td>FFINTBAS, 884</td>
</tr>
<tr>
<td>NFINTBAS, 3116</td>
<td>PWFFINTB, 3247</td>
</tr>
<tr>
<td>WFFINTBS, 4177</td>
<td>IntegralBasisPolynomialTools, 1352</td>
</tr>
<tr>
<td>IntegralBasisTools, 1347</td>
<td>integralLastSubResultant</td>
</tr>
<tr>
<td>RSETGCD, 3736</td>
<td>integrate</td>
</tr>
<tr>
<td>-definete</td>
<td>554</td>
</tr>
</tbody>
</table>
INDEX

DEFINTRF, 3628
FSINT, 899
INTPACK, 59
INTRAT, 3655
IRRF2F, 1358
STTAYLOR, 3863
IntegrationResultFunctions2, 1355
IntegrationResultRFToFunction, 1358
IntegrationResultToFunction, 1361
IntegrationTools, 1367
InterfaceGroebnerPackage, 1282
INTERGB, 1282
groebner, 1282
intermediateResultsIF
D02AGNT, 490
internalDecompose
RSDCMPK, 3728
SRDCMPK, 3827
internalInfRittWu?
QCMPACK, 3567
SFQCMPK, 3817
internalIntegrate
FSCINT, 893
INTRF, 3636
internalIntegrate0
FSCINT, 893
InternalPrintPackage, 1371
InternalRationalUnivariateRepresentationPackage, lift, 1036
1374
internalSubPolSet?
QCMPACK, 3567
SFQCMPK, 3817
internalSubQuasiComponent?
QCMPACK, 3567
SFQCMPK, 3817
interpolate
FFF, 852
PINTERP, 3437
RINTERP, 3658
interpolateForms
GPAFF, 992
INTFRSP, 1379
PAFF, 3215
PAFFF, 3223
InterpolateFormsPackage, 1379
interpret
INFORM1, 1280
interpretString
TEMUTL, 3958
interReduce
PSETPK, 3453
intersectionDivisor
GPAFF, 992
INTDIVP, 1387
PAFF, 3215
PAFFF, 3223
IntersectionDivisorPackage, 1387
INTFACT, 1293
BasicMethod, 1293
factor, 1293
PollardSmallFactor, 1293
squareFree, 1293
INTFRSP, 1379
basisOfInterpolateForms, 1379
basisOfInterpolateFormsForFact, 1379
interpolateForms, 1379
interpolateFormsForFact, 1379
INTG0, 1036
lift, 1036
multivariate, 1036
palgextint0, 1036
palgint0, 1036
palglimint0, 1036
palgLODE0, 1036
palgRDE0, 1036
univariate, 1036
INTHEORY, 1320
bernoulli, 1320
chineseRemainder, 1320
divisors, 1320
euler, 1320
eulerPhi, 1320
fibonacci, 1320
harmonic, 1320
jacobi, 1320
legendre, 1320
moebiusMu, 1320
moebiusMu,
numberOfDivisors, 1320
sumOfDivisors, 1320
sumOfKthPowerDivisors, 1320
INTHERAL, 19
    HermiteIntegrate, 21
INTHERTR, 3998
    HermiteIntegrate, 3998
INTPACK, 59
    integrate, 59
    measure, 59
INTPAF, 3547
    palgextint, 3547
    palgint, 3547
    palglimint, 3547
    palgLODE, 3547
    palgRDE, 3547
intPatternMatch
INTTOOLS, 1367
INTPM, 3306
    pmComplexIntegrate, 3306
    pmIntegrate, 3306
    splitConstant, 3306
INTRA, 1367
    extendedInt, 1367
    integrate, 1367
    limitedIntegrate, 1367
INTRET, 1337
    integer, 1337
    integer?, 1337
    integerIfCan, 1337
INTRF, 1367
    extendedIntegrate, 1367
    fieldIntegrate, 1367
    internalIntegrate, 1367
    limitedIntegrate, 1367
INTSLPE, 1344
    solveLinearPolynomialEquation, 1344
INTTOOLS, 1367
    intPatternMatch, 1367
    kmax, 1367
    ksec, 1367
    mkPrim, 1367
    removeConstantTerm, 1367
    union, 1367
    vark, 1367
vselect, 1367
INTTR, 4001
    expextendedInt, 4001
    expIntegrate, 4001
    expIntFieldPoly, 4001
    expLimitedIntegrate, 4001
    monomialIntegrate, 4001
    monomialIntPoly, 4001
    primextendedIntegrate, 4001
    primextIntfrac, 4001
    primIntegrate, 4001
    primIntFieldPoly, 4001
    primLimitedIntegrate, 4001
    tanIntegrate, 4001
inv
    INBFF, 1249
    inverse
    IMATQF, 1229
    MATLIN, 1652
    TABLBUMP, 3942
    inverseColeman
    SGCF, 3927
    inverseLaplace
    INVLAPLA, 1401
    InverseLaplaceTransform, 1401
    invertIfCan
    MATLIN, 1652
    INVLAPLA, 1401
    inverseLaplace, 1401
    invmultisect
    STTAYLOR, 3863
ipow
    COMBF, 208
    iprint
    IPRNTPK, 1372
    IPRNTPK, 1371
    iprint, 1372
IR2, 1355
    map, 1355
    IR2F, 1361
    complexExpand, 1361
    expand, 1361
    split, 1361
IROOT, 1339
    approxNthRoot, 1339
approxSqrt, 1339
perfectNthPower?, 1339
perfectNthRoot, 1339
perfectSqrt, 1339
perfectSquare?, 1339
irroot
AF, 14
IRREDFFX, 1390
generateIrredPoly, 1390
IrredPolyOverFiniteField, 1390
irreducible?
DDFACT, 349
FFFACTOR, 760
FFFACTSE, 766
irreducibleFactors
PSETPK, 3453
irreducibleRepresentation
IRSN, 1394
IRRF2F, 1358
complexExpand, 1358
complexIntegrate, 1358
expand, 1358
integrate, 1358
split, 1358
IrrRepSymNatPackage, 1394
IRSN, 1394
dimensionOfIrreducibleRepresentation, 1394
irreducibleFactors
IRSN, 1394
IrrepFactors
PSETPK, 3453
irreducibleRepresentation
IRSN, 1394
IRRURPK, 1374
checkRur, 1374
rur, 1374
Is
PATMATCH, 3294
is?
PATMATCH, 3294
isAbsolutelyIrreducible?
REP2, 3758
isExpt
POLYCATQ, 3401
isPlus
POLYCATQ, 3401
isPower
POLYCATQ, 3401
isQuotient
ESTOOLS, 667
isTimes
POLYCATQ, 3401
ISUMP, 1272
sum, 1272
iter
MAPACK1, 1600
ITFUN2, 1210
map, 1210
ITFUN3, 1212
map, 1212
ITRIGMNP, 1275
explogs2trigs, 1275
F2FG, 1275
GF2F, 1275
GF2FG, 1275
trigs2explogs, 1275
jacob
INTHEORY, 1320
jacobian
D02AGNT, 490
MCALCFN, 1768
janko2
PGE, 3346
karatsuba
UPMP, 4112
karatsubaOnce
UPMP, 4112
KERNEL2, 1408
constantIFCan, 1408
constantKernel, 1408
KernelFunctions2, 1408
kmax
INTTOOLS, 1367
KOVACIC, 1410
kovacic, 1410
Kovacic, 1410
kovacic
KOVACIC, 1410
KrullNumber
RSDCMPK, 3728
SRDCMPK, 3827
ksec
INTTOOLS, 1367
lagrange
STTAYLOR, 3863
LagrangeInterpolation
PINTERPA, 3440
laguerre
PNTHEORY, 3442
laguerreL
ORTHPOL, 3207
lambert
STTAYLOR, 3863
LAPLACE, 1414
laplace, 1414
laplace
LAPLACE, 1414
LaplaceTransform, 1414
laplacian
MCALCFN, 1768
largest
UDPO, 4135
lastSubResultant
PRS, 3527
lastSubResultantEuclidean
PRS, 3527
laurent
EXPR2UPS, 704
GENUPS, 1021
Lazard
PRS, 3527
Lazard2
PRS, 3527
LazardSetSolvingPackage, 1440
LAZM3PK, 1440
normalizeIfCan, 1440
zeroSetSplit, 1440
lazyGintegrate
STTAYLOR, 3863
lazyIntegrate
STTAYLOR, 3863
lazyIrreducibleFactors
PSETPK, 3453
lazyVariations
POLUTIL, 3684
lBasis
GPAFF, 992
PAFF, 3215
PAFFFT, 3223
lcm
POLYVEC, 4144
ldf2lst
D01AGNT, 451
ESCONT, 657
ESTOOLS, 667
ldf2vmf
ESTOOLS, 667
LEADCDET, 1444
distFact, 1444
polCase, 1444
LeadingCoeffDetermination, 1444
leadingCoefficient, 3430
leadingCoefficientRicDE
ODEPRRIC, 3512
leastAffineMultiple
FFPOLY, 788
leastPower
IBATOOL, 1347
leftDivide
NCODIV, 3086
OREPCTO, 4122
leftExactQuotient
NCODIV, 3086
leftFactor
PDECOMP, 3407
leftFactorIfCan
UPDECOMP, 4103
leftGcd
NCODIV, 3086
leftLcm
NCODIV, 3086
leftQuotient
NCODIV, 3086
leftRank
ALGPKG, 44
leftRemainder
NCODIV, 3086
legendre
INTHEORY, 1320
PNTHEORY, 3442
legendreP
ORTHPOL, 3207
length
GALFACTU, 948
lepol
GBINTERN, 1077
INDEX

less?
    UDPO, 4135
lex
    TABL BUMP, 3942
lexGroebner
    PGROEB, 3393
lexTriangular
    LEXTRIPK, 1517
LexTriangularPackage, 1517
LEXTRIPK, 1517
    fglmIfCan, 1517
groebner, 1517
lexTriangular, 1517
    squareFreeLexTriangular, 1517
zeroDimensional?, 1517
zeroSetSplit, 1517
LF, 1560
    belong?, 1560
Ci, 1560
dilog, 1560
Ei, 1560
erf, 1560
    integral, 1560
li, 1560
    operator, 1560
Si, 1560
lfextendedint
    INTEF, 589
lfextlimint
    INTEF, 589
lfieldint
    INTEF, 589
lfintegrate
    INTEF, 589
lflimitedint
    INTEF, 589
lffunc
    HB, 1180
LGROBP, 1549
    anticoord, 1549
    choosemon, 1549
    computeBasis, 1549
    coord, 1549
groebgen, 1549
    intcompBasis, 1549
    linGenPos, 1549
    minPol, 1549
totolex, 1549
transform, 1549
li
    LF, 1560
lift
    INTEG0, 1036
    MULTSQFR, 1782
limit
    LIMITPS, 3485
    LIMITRF, 3639
limitedint
    INTRAT, 3655
limitedIntegrate
    INTRF, 3636
    LIMITPS, 3484
        complexLimit, 3485
        limit, 3485
    LIMITRF, 3639
        complexLimit, 3639
        limit, 3639
LINDEP, 1523
    linearDependence, 1523
    linearlyDependent?, 1523
    solveLinear, 1523
linear
    SOLVEFOR, 3471
linear?
    E04AGNT, 721
    PSETPK, 3453
linearBezier
    BEZIER, 162
LinearDependence, 1523
    LinearDependence
    LINDEP, 1523
    linearDependenceOverZ
    ZLINDEP, 1304
    linearlyDependent?
    LINDEP, 1523
    linearlyDependentOverZ?
    ZLINDEP, 1304
linearMatrix
    E04AGNT, 721
LinearOrdinaryDifferentialOperatorFactorizer, 1526
LinearOrdinaryDifferentialOperatorsOps, 1530
INDEX

linearPart
   E04AGNT, 721
LinearPolynomialEquationByFractions, 1534
linearPolynomials
   PSETPK, 3453
linears
   MDDFACT, 1719
LinearSystemFromPowerSeriesPackage, 1536
LinearSystemMatrixPackage, 1539
LinearSystemMatrixPackage1, 1543
LinearSystemPolynomialPackage, 1546
lineColorDefault
   VIEWDEF, 4161
LinesOpPack, 1557
linGenPos
   LGROBP, 1549
LinGroebnerPackage, 1549
linkToFortran
   FORT, 846
linSolve
   LSSP, 1546
lintgcd
   HEUGCD, 1184
LiouvillianFunction, 1560
LIST2, 1566
   map, 1566
   reduce, 1566
   scan, 1566
LIST2MAP, 1571
   match, 1571
LIST3, 1569
   map, 1569
listAllMono
   PFORP, 3232
listAllMonoExp
   PFORP, 3232
listConjugateBases
   IABACHIN, 197
listexp
   NPCOEF, 3111
ListFunctions2, 1566
ListFunctions3, 1569
ListToMap, 1571
listVariable
   PFORP, 3232
listYoungTableaus
   SGCF, 3927
   LISYSER, 1536
   finiteSeries2LinSys, 1536
   finiteSeries2LinSysWOVectorise, 1536
   finiteSeries2Vector, 1536
localAbs
   FS2EXPXP, 913
   FS2UPS, 925
localIntegralBasis
   FFINTBAS, 884
   NFINTBAS, 3116
   PWFFINTB, 3247
   WFFINTBS, 4177
localize
   LPARSPT, 1575
LocalParametrizationOfSimplePointPackage, 1575
localParamOfSimplePt
   LPARSPT, 1575
localReal?
   EF, 535
localUnquote
   APPRULE, 109
LODEEF, 560
   solve, 560
LODO2FUN
   UTSODETL, 4141
LODOF, 1526
   factor, 1526
   factor1, 1526
LODOOPS, 1530
   directSum, 1530
   symmetricPower, 1530
   symmetricProduct, 1530
log
   EF, 535
   FACTFUNC, 728
   XEXPPKG, 4184
logGamma
   DFSFUN, 369
lookup
   INBFF, 1249
loopPoints
   TUBETOOL, 4067
LOP, 1557
   quotVecSpaceBasis, 1557
INDEX

reduceLineOverLine, 1557
reduceRow, 1557
reduceRowOnList, 1557
rowEchWoZeroLines, 1557
rowEchWoZeroLinesWOVectorise, 1557
lowerPolynomial
FACUTIL, 738
LowTriBddDenomInv
TRIMAT, 4059
LPARSPT, 1575
localize, 1575
localParamOfSimplePt, 1575
pointDominanteBy, 1575
pointToPlace, 1575
printInfo, 1575
LPEFRAC, 1534
solveLinearPolynomialEquationByFractions, 1534
LPolynomial
GPAFF, 992
PAFF, 3215
PAFFFF, 3223
LSMP, 1539
hasSolution?, 1540
particularSolution, 1540
rank, 1540
solve, 1540
LSMP1, 1543
hasSolution?, 1543
particularSolution, 1543
rank, 1543
solve, 1543
LSPP, 1546
linSolve, 1546
mainVariable
POLYCATQ, 3401
RF, 3625
MakeBinaryCompiledFunction, 1582
makeCrit
GBINTERN, 1077
makeEq
INFSP, 1264
MakeFloatCompiledFunction, 1585
makeFR
GALFACT, 954
MakeFunction, 1593
makeObject
DRAW, 3965
DRAWCFUN, 3978
MakeRecord, 1595
makeRecord
MKRECORD, 1595
MakeUnaryCompiledFunction, 1597
makeYoungTableau
SGCF, 3927
makingStats?
TBCMPPK, 3947
MAMA, 1703
aColumn, 1703
aRow, 1703
bandMatrix, 1703
blockConcat, 1703
blockSplit, 1703
columns, 1703
diagonalMatrix, 1703
element, 1703
horizConcat, 1703
horizSplit, 1703
rows, 1703
subMatrix, 1703
vertConcat, 1703
vertSplit, 1703
map
ARRAY12, 3189
CARTEN2, 186
COMPLEX2, 235
DIRPROD2, 339
EQ2, 614
ES1, 686
ES2, 688
ESTOOLS2, 678
EXPR2, 680
FAMR2, 756
FDIV2, 758
FFCAT2, 881
FLAGG2, 820
FR2, 733
FRAC2, 867
FRIDEAL2, 849
FRNAAF2, 869
FS2, 897

4329
4330

INDEX

FSAGG2, 827
IR2, 1355
ITFUN2, 1210
ITFUN3, 1212
LIST2, 1566
LIST3, 1569
MATCAT2, 1646
MMAP, 1766
MPC2, 1747
MPC3, 1749
MRF2, 1732
NSUP2, 3075
OCTCT2, 3180
ONECOMP2, 3191
ORDCOMP2, 3191
PARPC2, 3271
PARSRC2, 3273
PARSU2, 3275
PATRES2, 3330
PATTERN2, 3292
POLY2, 3424
POLYLIFF, 3398
PRIMARR2, 3500
PTFUNC2, 3367
QFCAT2, 3576
QUATCT2, 3579
RMACAT2, 3704
SEG2, 3792
SEGBIND2, 3790
STREAM2, 3853
STREAM3, 3856
SUP2, 3811
ULS2, 4096
UNISEG2, 4133
UP2, 4110
UPOLYC2, 4099
UPXS2, 4119
UTS2, 4126
VECTOR2, 4158
mapBivariate
IBPTOOLS, 1352
mapdiv
STTAYLOR, 3863
mapExponents
PFORP, 3232
MAPHACK1, 1600
iter, 1600
recr, 1600
MAPHACK2, 1602
arg1, 1602
arg2, 1602
MAPHACK3, 1604
comp, 1604
mapMatrixIfCan
IBPTOOLS, 1352
mapmult
STTAYLOR, 3863
MappingPackage1, 1614
MappingPackage2, 1625
MappingPackage3, 1636
MappingPackage4, 1643
MappingPackageInternalHacks1, 1600
MappingPackageInternalHacks2, 1602
MappingPackageInternalHacks3, 1604
MAPPKG1, 1614
**?, 1614
cocere, 1614
fixedPoint, 1614
id, 1614
nullary, 1614
recr, 1614
MAPPKG2, 1625
const, 1625
constant, 1625
curry, 1625
diag, 1625
MAPPKG3, 1636
**?, 1636
constantLeft, 1636
constantRight, 1636
curryLeft, 1636
curryRight, 1636
twist, 1636
MAPPKG4, 1643
**?, 1643
?+?, 1643
?-?, 1643
?/?, 1643
mapSolve
SOLVEFOR, 3471
mapUnivariate
IBPTOOLS, 1352
INDEX

mapUnivariateIfCan
  IBPTOOLS, 1352
mat
  ESTOOLS, 667
MATCAT2, 1646
  map, 1646
  reduce, 1646
match
  LIST2MAP, 1571
mathieu11
  PGE, 3346
mathieu12
  PGE, 3346
mathieu22
  PGE, 3346
mathieu23
  PGE, 3346
mathieu24
  PGE, 3346
MATLIN, 1652
  adjoint, 1652
  determinant, 1652
e1Column2, 1652
e1Row1, 1652
e1Row2, 1652
fractionFreeGauss, 1652
inverse, 1652
invertIfCan, 1652
minordet, 1652
normalizedDivide, 1652
nullity, 1652
nullSpace, 1652
rank, 1652
rowEchelon, 1652
MatrixCategoryFunctions2, 1646
MatrixCommonDenominator, 1649
matrixGcd
  IBATOOL, 1347
MatrixLinearAlgebraFunctions, 1652
MatrixManipulation, 1703
MATSTOR, 3845
  ???, 3845
copy, 3845
leftScalarTimes, 3845
minus, 3845
plus, 3845
  power, 3845
  rightScalarTimes, 3845
times, 3845
maxPoints
  GRDEF, 1052
maxrank
  PLEQN, 3257
maxrow
  TABLBUMP, 3942
MCALCFN, 1768
  bandedHessian, 1768
  bandedJacobian, 1768
divergence, 1768
  gradient, 1768
hessian, 1768
jacobi, 1768
laplacian, 1768
MCDEN, 1649
  clearDenominator, 1649
  commonDenominator, 1649
  splitDenominator, 1649
MDDFACT, 1719
  ddFact, 1719
  exptMod, 1719
  factor, 1719
gcd, 1719
  linears, 1719
  separateFactors, 1719
measure
  INTPACK, 59
  ODEPACK, 81
  OPTPACK, 71
  PDEPACK, 91
measure2Result
  ESTOOLS, 667
meatAxe
  REP2, 3758
mergeDifference
  MTHING, 1712
mergeFactors
  FRUTIL, 735
Merge Thing, 1712
MESH, 1715
  meshFun2Var, 1715
  meshPar1Var, 1715
  meshPar2Var, 1715
INDEX

INTTR, 4001

monomialIntPoly
INTTR, 4001

monomials
PFORP, 3232

monomRDE
RDETR, 4027

monomRDEsys
RDETRS, 4032

MONOTOOL, 1734

decompose, 1734

normalDenom, 1734

split, 1734

splitSquarefree, 1734

more?
UDPO, 4135

moreAlgebraic?
QCMPACK, 3567
SFQCMKP, 3817

MoreSystemCommands, 1737

MPC2, 1747

map, 1747

reshape, 1747

MPC3, 1749

map, 1749

MPCPF, 1739

factor, 1739

MPolyCatFunctions2, 1747

MPolyCatFunctions3, 1749

MPolyCatPolyFactorizer, 1739

MPolyCatRationalFunctionFactorizer, 1742

MPRFF, 1742

factor, 1742

pushidown, 1742

pushidterm, 1742

pushucoef, 1742

pushuconst, 1742

pushup, 1742

totalfract, 1742

mpsode
UTSODE, 4128

mr
TABLBUMP, 3942

MRATFAC, 1752

factor, 1752

MRationalFactorize, 1752

MRF2, 1732

map, 1732

MSYSMD, 1737

systemCommand, 1737

MTHING, 1712

mergeDifference, 1712

mul
POLYVEC, 4144

mulbybinomial

POLYVEC, 4144

mulbyscalar

POLYVEC, 4144

MULTFACT, 1773

factor, 1773

MultFiniteFactorize, 1754

multiEuclideanTree

CRAPACK, 273

multinomial

COMBINAT, 1289

multiple

PMASS, 3297

PMASSFS, 888

MultipleMap, 1766

multiplicity

NPOLYGON, 3080

PLPKCRV, 3386

multisect

STTAYLOR, 3863

multiserv

AXSERV, 121

MultiVariableCalculusFunctions, 1768

multivariate

INTG0, 1036

POLYCATQ, 3401

RF, 3625

MultivariateFactorize, 1773

MultivariateSquareFree, 1782

MULTSQFR, 1782

check, 1782

coeffChoose, 1782

compdegd, 1782

consnewpol, 1782

intChoose, 1782

lift, 1782

myDegree, 1782

normDeriv2, 1782
nsqfree, 1782
squareFree, 1782
squareFreePrim, 1782
univcase, 1782
musserTrials
GALFACT, 954
myDegree
MULTSQFR, 1782

NAGC02, 2812
c02aff, 2813
c02agf, 2813
NAGC05, 2836
c05adf, 2836
c05nbf, 2836
c05pf, 2836
NAGC06, 2901
c06eaf, 2901
c06ebf, 2901
c06ecf, 2901
c06ekf, 2901
c06fpf, 2901
c06gqf, 2901
c06frf, 2901
c06fuf, 2901
c06gbf, 2901
c06gcg, 2901
c06gqf, 2901
c06gkgf, 2901
NAGD01, 2272
do1ajf, 2273
do1akf, 2273
do1alf, 2273
do1amf, 2273
do1anf, 2273
do1apf, 2273
do1aqf, 2273
do1asf, 2273
do1bbf, 2273
do1bbf, 2273
do1bpf, 2273
do1celf, 2273
do1gaf, 2273
do1gbf, 2273
NAGD02, 2745
do2bbf, 2745
do2bbf, 2745
do2celf, 2745

d02celf, 2745
do2gaf, 2745
do2gbf, 2745
do2kef, 2745
d02raf, 2745

d03edf, 2792
do3eef, 2792
do3faf, 2792
NAGD03, 2792
e01baf, 2339
e01bcf, 2339
e01bff, 2339
e01bgf, 2339
e01bfh, 2339
e01daf, 2339
e01saf, 2339
e01sbf, 2339
e01sef, 2339
e01ssf, 2339
NAGE01, 2339
e02adj, 2074
e02acf, 2074
e02agf, 2074
e02ahf, 2074
e02ajf, 2074
e02akf, 2074
e02bf, 2074
e02bbf, 2074
e02bcf, 2074
e02bdf, 2074
e02bef, 2074
e02daf, 2074
e02dcf, 2074
e02ddf, 2074
e02def, 2074
e02dff, 2074
e02gaf, 2074
e02zaf, 2074
NAGE04, 2646
e04dgg, 2646
e04dfg, 2646
e04gcf, 2646
e04jaf, 2646
e04nbf, 2646
e04naf, 2646
INDEX

4335

e04ncf, 2646
f07fef, 2367
e04ycf, 2646
NagEigenPackage, 1889
NagFittingPackage, 2074
NAGF01, 2452
NagInterpolationPackage, 2339
f01brf, 2452
NagLapack, 2367
f01bsf, 2452
NagLinearEquationSolvingPackage, 2183
f01maf, 2452
NAGLinkSupportPackage, 2193
f01mcf, 2452
NagMatrixOperationsPackage, 2452
f01qcf, 2452
NagOptimisationPackage, 2646
f01qdf, 2452
NagPartialDifferentialEquationsPackage, 2745
f01qef, 2452
NagPolynomialRootsPackage, 2812
f01rcf, 2452
NagRootFindingPackage, 2836
f01rdf, 2452
NAGS, 3058
f01ref, 2452
f01ref, 2452
NAGF02, 1889
s01eaf, 3058
f02aaf, 1889
s13aaf, 3058
f02abf, 1889
s13acf, 3058
f02adf, 1889
s13adbf, 3058
f02aef, 1889
s14aaf, 3058
f02aff, 1889
s14abf, 3058
f02agf, 1889
s14baf, 3058
f02ajf, 1889
s15adf, 3058
f02akf, 1889
s15aef, 3058
f02awf, 1889
s17aaf, 3058
f02axf, 1889
s17adbf, 3058
f02bbf, 1889
s17aef, 3058
f02bjf, 1889
s17aff, 3058
f02jff, 1889
s17agf, 3058
f02wef, 1889
s17ahf, 3058
f02xaf, 1889
s17ajf, 3058
NAGF04, 2183
NAGF07, 2367
f04adf, 2183
s17aaf, 3058
f04arf, 2183
s17adbf, 3058
f04asf, 2183
s17dgvf, 3058
f04atf, 2183
s17dhhf, 3058
f04axf, 2183
s17df, 3058
f04aef, 2183
s18afe, 3058
f04alf, 2183
s18aff, 3058
f04asbf, 2183
s18df, 3058
f04msf, 2183
s18def, 3058
f04mef, 2183
s18def, 3058
f04msef, 2183
s18def, 3058
f07ajaf, 2367
s19aaf, 3058
f07alf, 2367
s19abf, 3058
f07aef, 2367
s19acf, 3058
f07afdf, 2367
s19adbf, 3058
NagSeriesSummationPackage, 2901
NAGSP, 2193
    aspFilename, 2193
    checkPrecision, 2193
    dimensionsOf, 2193
    fortranCompilerName, 2193
    fortranLinkerArgs, 2193
    restorePrecision, 2193
NagSpecialFunctionsPackage, 3058
NCEP, 3166
    characteristicPolynomial, 3167
    complexEigenvalues, 3167
    complexEigenvectors, 3167
NCNTFRAC, 3170
    continuedFraction, 3170
NCODIV, 3086
    leftDivide, 3086
    leftExactQuotient, 3086
    leftGcd, 3086
    leftLcm, 3086
    leftQuotient, 3086
    leftRemainder, 3086
negAndPosEdge
N POLYGON, 3080
neglist
    ESTOOLS1, 676
newLine
    DISPLAY, 345
newReduc
    FSRED, 907
NewSparseUnivariatePolynomialFunctions2, 3058
    Karatsuba
NEWTON, 3077
    newton, 3077
    NewtonInterpolation, 3077
    NewtonPolygon, 3080
    newtonPolygon
    N POLYGON, 3080
    newtonPolySlope
    BLUPPACK, 170
    nextColeman
    SGCF, 3927
    nextIrreduciblePoly
    FFPOLY, 788
    nextLatticePermutation
    SGCF, 3927
    nextNormalPoly
    FFPOLY, 788
    nextNormalPrimitivePoly
    FFPOLY, 788
    nextPartition
    SGCF, 3927
    nextPrime
    PRIMES, 1326
    nextPrimitiveNormalPoly
    FFPOLY, 788
    nextPrimitivePoly
    FFPOLY, 788
    nextsousResultant2
    PRS, 3527
    nextSublist
    PLEQN, 3257
    nextSubsetGray
    GRAY, 1061
    NFINTBAS, 3116
        discriminant, 3116
        integralBasis, 3116
        localIntegralBasis, 3116
    nlde
    STTAYLOR, 3863
    NLINSOL, 3096
        solve, 3096
        solveInField, 3096
    NODE1, 3091
        solve, 3091
    noKaratsuba
    UPMP, 4112
    noLinearFactor?
    BRILL, 180
    NonCommutativeOperatorDivision, 3058
    NONE1, 3089
        coerce, 3089
    NoneFunctions1, 3089
    NonLinearFirstOrderODESolver, 3091
    nonLinearPart
<table>
<thead>
<tr>
<th>INDEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>E04AGNT, 721</td>
</tr>
<tr>
<td>NonLinearSolvePackage, 3096</td>
</tr>
<tr>
<td>nonQsign</td>
</tr>
<tr>
<td>TOOLSIGN, 3963</td>
</tr>
<tr>
<td>norm</td>
</tr>
<tr>
<td>CRFF, 245</td>
</tr>
<tr>
<td>GALFACTU, 948</td>
</tr>
<tr>
<td>INBFF, 1249</td>
</tr>
<tr>
<td>NORMMA, 3105</td>
</tr>
<tr>
<td>normal</td>
</tr>
<tr>
<td>RFDIST, 3613</td>
</tr>
<tr>
<td>normal form, 3430</td>
</tr>
<tr>
<td>normal01</td>
</tr>
<tr>
<td>RFDIST, 3613</td>
</tr>
<tr>
<td>normal?</td>
</tr>
<tr>
<td>FFPOLY, 788</td>
</tr>
<tr>
<td>INBFF, 1249</td>
</tr>
<tr>
<td>normalDenom</td>
</tr>
<tr>
<td>MONOTOOL, 1734</td>
</tr>
<tr>
<td>normalDeriv</td>
</tr>
<tr>
<td>FACUTIL, 738</td>
</tr>
<tr>
<td>normalElement</td>
</tr>
<tr>
<td>INBFF, 1249</td>
</tr>
<tr>
<td>normalForm</td>
</tr>
<tr>
<td>GB, 1114</td>
</tr>
<tr>
<td>PSEUDULIN, 3521</td>
</tr>
<tr>
<td>normalize</td>
</tr>
<tr>
<td>REP, 3582</td>
</tr>
<tr>
<td>NormalizationPackage, 3100</td>
</tr>
<tr>
<td>normalize</td>
</tr>
<tr>
<td>EFSSTRUCT, 579</td>
</tr>
<tr>
<td>NORMPK, 3100</td>
</tr>
<tr>
<td>normalizedAssociate</td>
</tr>
<tr>
<td>NORMPK, 3100</td>
</tr>
<tr>
<td>normalizedDivide</td>
</tr>
<tr>
<td>MATLIN, 1652</td>
</tr>
<tr>
<td>MHROWRED, 1725</td>
</tr>
<tr>
<td>normalizeIfCan</td>
</tr>
<tr>
<td>LAZM3PK, 1440</td>
</tr>
<tr>
<td>normDeriv2</td>
</tr>
<tr>
<td>MULTSQFR, 1782</td>
</tr>
<tr>
<td>normFactors</td>
</tr>
<tr>
<td>NORMRETR, 3108</td>
</tr>
<tr>
<td>NormInMonogenicAlgebra, 3105</td>
</tr>
<tr>
<td>normInvertible?</td>
</tr>
<tr>
<td>NORMPK, 3100</td>
</tr>
<tr>
<td>NORMMA, 3105</td>
</tr>
<tr>
<td>norm, 3105</td>
</tr>
<tr>
<td>NORMPK, 3100</td>
</tr>
<tr>
<td>normalize, 3100</td>
</tr>
<tr>
<td>normalizedAssociate, 3100</td>
</tr>
<tr>
<td>normInvertible?, 3100</td>
</tr>
<tr>
<td>outputArgs, 3100</td>
</tr>
<tr>
<td>recip, 3100</td>
</tr>
<tr>
<td>NORMRETR, 3108</td>
</tr>
<tr>
<td>Frobenius, 3108</td>
</tr>
<tr>
<td>normFactors, 3108</td>
</tr>
<tr>
<td>retractIfCan, 3108</td>
</tr>
<tr>
<td>NormRetractPackage, 3108</td>
</tr>
<tr>
<td>NPCOEF, 3111</td>
</tr>
<tr>
<td>listexp, 3111</td>
</tr>
<tr>
<td>npcoef, 3111</td>
</tr>
<tr>
<td>NPCof, 3111</td>
</tr>
<tr>
<td>npcoef</td>
</tr>
<tr>
<td>NPCOEF, 3111</td>
</tr>
<tr>
<td>NPOLYGON, 3080</td>
</tr>
<tr>
<td>multiplicity, 3080</td>
</tr>
<tr>
<td>negAndPosEdge, 3080</td>
</tr>
<tr>
<td>newtonPolygon, 3080</td>
</tr>
<tr>
<td>slope, 3080</td>
</tr>
<tr>
<td>NREP, 3172</td>
</tr>
<tr>
<td>characteristicPolynomial, 3172</td>
</tr>
<tr>
<td>realEigenvalues, 3172</td>
</tr>
<tr>
<td>realEigenvectors, 3172</td>
</tr>
<tr>
<td>nqsfree</td>
</tr>
<tr>
<td>MULTSQFR, 1782</td>
</tr>
<tr>
<td>NSUP2, 3075</td>
</tr>
<tr>
<td>map, 3075</td>
</tr>
<tr>
<td>ntrhr</td>
</tr>
<tr>
<td>POLYROOT, 3448</td>
</tr>
<tr>
<td>nthRoot</td>
</tr>
<tr>
<td>FACTFUNC, 728</td>
</tr>
<tr>
<td>NTPOLFN, 3127</td>
</tr>
<tr>
<td>bernoulliB, 3127</td>
</tr>
<tr>
<td>cyclotomic, 3127</td>
</tr>
<tr>
<td>eulerE, 3127</td>
</tr>
<tr>
<td>nullary</td>
</tr>
<tr>
<td>MAPPKG1, 1614</td>
</tr>
<tr>
<td>nullity</td>
</tr>
<tr>
<td>MATLIN, 1652</td>
</tr>
<tr>
<td>nullSpace</td>
</tr>
<tr>
<td>IMATQF, 1229</td>
</tr>
<tr>
<td>MATLIN, 1652</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>NumberOfFactors</td>
</tr>
<tr>
<td>Galfact, 954</td>
</tr>
<tr>
<td>NumberOfImproperPartitions</td>
</tr>
<tr>
<td>NumberOfIrreduciblePoly</td>
</tr>
<tr>
<td>NumberOfNormalPoly</td>
</tr>
<tr>
<td>NumberOfOperations</td>
</tr>
<tr>
<td>NumberOfPlacesOfDegree</td>
</tr>
<tr>
<td>PAFF, 3215</td>
</tr>
<tr>
<td>NumberOfPrimitivePoly</td>
</tr>
<tr>
<td>NumberOfValuesNeeded</td>
</tr>
<tr>
<td>NumberOfVariables</td>
</tr>
<tr>
<td>SRDCMPK, 3827</td>
</tr>
<tr>
<td>GPAFF, 992</td>
</tr>
<tr>
<td>PAFFFF, 3223</td>
</tr>
<tr>
<td>GPAFF, 992</td>
</tr>
<tr>
<td>PAFFFF, 3223</td>
</tr>
<tr>
<td>NUMERIC, 3131</td>
</tr>
<tr>
<td>complexNumericIfCan, 3131</td>
</tr>
<tr>
<td>numericIfCan, 3131</td>
</tr>
<tr>
<td>numeric</td>
</tr>
<tr>
<td>NumericContinuousFraction, 3170</td>
</tr>
<tr>
<td>NUMERIC, 3131</td>
</tr>
<tr>
<td>NumericTubePlot, 3175</td>
</tr>
<tr>
<td>FormatArabic, 3122</td>
</tr>
<tr>
<td>ScanArabic, 3122</td>
</tr>
<tr>
<td>ScanFloatIgnoreSpacesIfCan, 3122</td>
</tr>
<tr>
<td>NUMODE, 3144</td>
</tr>
<tr>
<td>rk4a, 3144</td>
</tr>
<tr>
<td>rk4qc, 3144</td>
</tr>
<tr>
<td>arromberg, 3154</td>
</tr>
<tr>
<td>atrapezoidal, 3154</td>
</tr>
<tr>
<td>rombergo, 3154</td>
</tr>
<tr>
<td>simpsono, 3154</td>
</tr>
<tr>
<td>trapezoidalo, 3154</td>
</tr>
<tr>
<td>tube, 3175</td>
</tr>
<tr>
<td>COORDSYS, 268</td>
</tr>
<tr>
<td>ESTOOLS, 667</td>
</tr>
<tr>
<td>map, 3180</td>
</tr>
<tr>
<td>oddInfiniteProduct</td>
</tr>
<tr>
<td>INPRODF, 1204</td>
</tr>
<tr>
<td>STINPROD, 3859</td>
</tr>
<tr>
<td>STTAYLOR, 3863</td>
</tr>
<tr>
<td>STTAYLOR, 3863</td>
</tr>
</tbody>
</table>
ode
  UTSODE, 4128
ode1
  UTSODE, 4128
ode2
  UTSODE, 4128
ODECONST, 264
  constDsolve, 264
ODEEFS, 567
  solve, 567
ODEINT, 3182
  diff, 3182
  expint, 3182
  int, 3182
ODEIntegration, 3182
ODEPACK, 81
  measure, 81
  solve, 81
ODEPAL, 3556
  algDsolve, 3556
ODEPRIM, 3507
  denomLODE, 3507
  indicialEquation, 3507
  indicialEquations, 3507
  splitDenominator, 3507
ODEPRRIC, 3512
  changeVar, 3512
  constantCoefficientRicDE, 3512
  denomRicDE, 3512
  leadingCoefficientRicDE, 3512
  polyRicDE, 3512
  singRicDE, 3512
ODERAT, 3662
  indicialEquationAtInfinity, 3662
  ratDsolve, 3662
ODERED, 3722
  reduceLODE, 3722
ODERTRIC, 3671
  polyRicDE, 3671
  ricDsolve, 3671
  singRicDE, 3671
ODESYS, 3913
  solve, 3913
  solveInField, 3913
  triangulate, 3913
ODETOOLS, 3186
  particularSolution, 3186
  variationOfParameters, 3186
  wronskianMatrix, 3186
ODETools, 3186
OMEXPR, 696
  OMwrite, 697
OMlistCDs
  OMPKG, 3194
OMlistSymbols
  OMPKG, 3194
OMPKG, 3194
  OMlistCDs, 3194
  OMlistSymbols, 3194
  OMread, 3194
  OMreadFile, 3194
  OMreadStr, 3194
OMsupportsCD?, 3194
OMsupportsSymbol?, 3194
OMunhandledSymbol, 3194
OMread
  OMPKG, 3194
  OMreadFile
  OMPKG, 3194
  OMreadStr
    OMPKG, 3194
OMreceive
  OMSERVER, 3197
OMsend
  OMSERVER, 3197
OMserve
  OMSERVER, 3197
OMsupportsCD?
  OMPKG, 3194
OMsupportsSymbol?
  OMPKG, 3194
OMunhandledSymbol
  OMPKG, 3194
OMwrite
  OMEXPR, 697
ONECOMP2, 3191
  map, 3191
OneDimensionalArrayFunctions2, 3189
OnePointCompletionFunctions2, 3191
OpenMathPackage, 3194
OpenMathServerPackage, 3197
OperationsQuery, 3200
operator
   AF, 14
   COMBF, 208
   COMMONOP, 223
   EF, 535
   FSPECF, 872
   LF, 1560
OPQUERY, 3200
   getDatabase, 3200
optAttributes
   E04AGNT, 721
optimize
   OPTPACK, 71
option
   DROPT1, 426
optional
   PMASS, 3297
   PMASSFS, 888
OPTPACK, 71
   goodnessOfFit, 71
   measure, 71
   optimize, 71
ORDCOMP2, 3202
   map, 3202
order
   FORDER, 753
   PFO, 3373
   PFOQ, 3380
   RDIV, 3719
OrderedCompletionFunctions2, 3202
OrderingFunctions, 3204
ORDFUNS, 3204
   pureLex, 3204
   reverseLex, 3204
   totalLex, 3204
OREPCTO, 4122
   apply, 4122
   leftDivide, 4122
   monicLeftDivide, 4122
   monicRightDivide, 4122
   rightDivide, 4122
   times, 4122
OrthogonalPolynomialFunctions, 3207
orthonormalBasis
   REP, 3582
ORTHPOL, 3207
   chebyshevT, 3207
   chebyshevU, 3207
   hermiteH, 3207
   laguerreL, 3207
   legendreP, 3207
OUT, 3211
   output, 3211
   outputList, 3211
output
   OUT, 3211
outputArgs
   NORMPK, 3100
outputAsFortran
   FORT, 846
   SPECOUT, 3813
outputAsScript
   SPECOUT, 3813
outputAsTex
   SPECOUT, 3813
outputList
   OUT, 3211
outputMeasure
   ESTOOLS, 667
OutputPackage, 3211
overset?
   PLEQN, 3257
packageCall
   INFORM1, 1280
PackageForAlgebraicFunctionField, 3215
PackageForAlgebraicFunctionFieldOverFiniteField, 3223
PackageForPoly, 3232
PADE, 3242
   pade, 3242
   pade, 3242
   pade
   PADE, 3242
   PADEPAC, 3240
PadeApproximantPackage, 3240
PadeApproximants, 3242
pade
   PADE, 3242
   PADEPAC, 3240
padecf
INDEX

PADE, 3242
PADEPAC, 3240
pade, 3240
PAAdicWildFunctionFieldIntegralBasis, 3247
PAFF, 3215
adjunctionDivisor, 3215
classNumber, 3215
desingTree, 3215
desingTreeWoFullParam, 3215
eval, 3215
evalIfCan, 3215
findOrderOfDivisor, 3215
fullDesTree, 3215
fullInfClsPt, 3215
genus, 3215
genusNeg, 3215
goppaCode, 3215
homogenize, 3215
interpolateForms, 3223
interpolateFormsForFact, 3223
intersectionDivisor, 3223
lBasis, 3223
LPolynomial, 3223
numberOfPlacesOfDegree, 3223
numberPlacesDegExtDeg, 3223
numberRatPlacesExtDeg, 3223
parametrize, 3223
placesAbove, 3223
placesOfDegree, 3223
pointDominateBy, 3223
projectivePoint, 3223
rationalPlaces, 3223
rationalPoints, 3223
setCurve, 3223
setSingularPoints, 3223
singularPoints, 3223
theCurve, 3223
translateToOrigin, 3223
ZetaFunction, 3223
palgextint
INTPAF, 3547
palgextint0
INTG0, 1036
palginfieldint
INTALG, 24
palgint
INTPAF, 3547
palgint0
INTG0, 1036
palgintegrate
INTALG, 24
palglimint
INTPAF, 3547
palglimint0
INTG0, 1036
palgLODE
INTPAF, 3547
palgLODE0

PAFFFF, 3223
adjunctionDivisor, 3223
classNumber, 3223
desingTree, 3223
desingTreeWoFullParam, 3223
eval, 3223
evalIfCan, 3223
findOrderOfDivisor, 3223
fullDesTree, 3223
fullInfClsPt, 3223
genus, 3223
genusNeg, 3223
goppaCode, 3223
homogenize, 3223
interpolateForms, 3223
interpolateFormsForFact, 3223
intersectionDivisor, 3223
lBasis, 3223
LPolynomial, 3223
numberOfPlacesOfDegree, 3223
numberPlacesDegExtDeg, 3223
numberRatPlacesExtDeg, 3223
parametrize, 3223
placesAbove, 3223
placesOfDegree, 3223
pointDominateBy, 3223
projectivePoint, 3223
rationalPlaces, 3223
rationalPoints, 3223
setCurve, 3223
setSingularPoints, 3223
singularPoints, 3223
theCurve, 3223
translateToOrigin, 3223
ZetaFunction, 3223
INDEX

INTG0, 1036
palgRDE
   INTPAF, 3547
palgRDE0
   INTG0, 1036
PAN2EXPR, 3396
coerce, 3396
parabolic
   COORDSYS, 268
   parabolicCylindrical
      COORDSYS, 268
paraboloidal
   COORDSYS, 268
ParametricalCombinatorsForStreams, 3253
ParametricLinearEquations, 3257
ParametricPlaneCurveFunctions2, 3271
ParametricSpaceCurveFunctions2, 3273
ParametricSurfaceFunctions2, 3275
ParametrizationPackage, 3277
parametrize
   GPAFF, 992
   PAFF, 3215
   PAFFFF, 3223
   PARAMP, 3277
PARAMP, 3277
   parametrize, 3277
ParCond
   PLEQN, 3257
ParCondList
   PLEQN, 3257
PARPC2, 3271
   map, 3271
PARSC2, 3273
   map, 3273
PARSU2, 3275
   map, 3275
partialFraction
   PFRPAC, 3282
PartialFractionPackage, 3282
particularSolution
   LSMTP, 1540
   LSMTP1, 1543
   ODETOOLS, 3186
   SOLVEFOR, 3471
partition
   COMBINAT, 1289
   partitions
      PARPERM, 3285
      PartitionsAndPermutations, 3285
PARPERM, 3285
   conjugate, 3285
   conjugates, 3285
   partitions, 3285
   permutations, 3285
   sequences, 3285
   shuffle, 3285
   shufflein, 3285
pascalTriangle
   GALUTIL, 975
   PATMATCH, 3294
Is, 3294
is?, 3294
PATRES2, 3330
   map, 3330
PATTERN1, 3289
   addBadValue, 3289
   badValues, 3289
   predicate, 3289
   satisfy?, 3289
   suchThat, 3289
PATTERN2, 3292
   map, 3292
PatternFunctions1, 3289
PatternFunctions2, 3292
PatternMatch, 3294
patternMatch
   CPMATCH, 241
   PMDOWN, 3324
   PMFS, 3300
   PMINS, 3303
   PMKERNEL, 3314
   PMLSAGG, 3318
   PMPLCAT, 3320
   PMQFCAT, 3327
   PMSYM, 3332
   PMTOOLS, 3334
PatternMatchAssertions, 3297
PatternMatchFunctionSpace, 3300
PatternMatchIntegerNumberSystem, 3303
PatternMatchIntegration, 3306
PatternMatchKernel, 3314
PatternMatchListAggregate, 3318
INDEX

PatternMatchPolynomialCategory, 3320
PatternMatchPushDown, 3324
PatternMatchQuotientFieldCategory, 3327
PatternMatchResultFunctions2, 3330
PatternMatchSymbol, 3332
patternMatchTimes
  PMTOOLS, 3334
PatternMatchTools, 3334
PCOMP, 3405
    compose, 3405
PDECOMP, 3407
    decompose, 3407
    leftFactor, 3407
    rightFactorCandidate, 3407
PDEPACK, 91
    measure, 91
    solve, 91
pdf2df
  ESTOOLS, 667
pdf2ef
  ESTOOLS, 667
perfectNthPower?
  IROOT, 1339
perfectNthRoot
  IROOT, 1339
perfectSqrt
  IROOT, 1339
perfectSquare?
  IROOT, 1339
PERMAN, 3340
    permanent, 3340
Permanent, 3340
permanent
  PERMAN, 3340
permutation
  COMBF, 208
  COMBINAT, 1289
PermutationGroupExamples, 3346
permutationRepresentation
  REP1, 3750
permutations
  PARTPERM, 3285
PFBR, 3410
    bivariateSLPEBR, 3410
    factorByRecursion, 3410
    factorSFBRlcUnit, 3410
    factorSquareFreeByRecursion, 3410
    randomR, 3410
    solveLinearPolynomialEquationByRecursion, 3410
PFBRU, 3418
    factorByRecursion, 3418
    factorSFBRlcUnit, 3418
    factorSquareFreeByRecursion, 3418
    randomR, 3418
    solveLinearPolynomialEquationByRecursion, 3418
PFO, 3373
    order, 3373
    torsion?, 3373
    torsionIfCan, 3373
PFOQ, 3380
    order, 3380
    torsion?, 3380
    torsionIfCan, 3380
PFORP, 3232
    constant, 3232
    degOneCoef, 3232
    degree, 3232
    degreeOfMinimalForm, 3232
    firstExponent, 3232
    homogenize, 3232
    listAllMono, 3232
    listAllMonoExp, 3232
    listVariable, 3232
    mapExponents, 3232
    minimalForm, 3232
    monomials, 3232
    replaceVarByOne, 3232
    replaceVarByZero, 3232
    subs1stVar, 3232
    subs2ndVar, 3232
    subsInVar, 3232
    totalDegree, 3232
    translate, 3232
    univariate, 3232
PFOTOOLS, 3383
    badNum, 3383
    doubleDisc, 3383
    getGoodPrime, 3383
    mix, 3383
    polyred, 3383
PFRPAC, 3282
  partialFraction, 3282
PGCD, 3426
  gcd, 3426
  gcdPrimitive, 3426
PGE, 3346
  abelianGroup, 3346
  alternatingGroup, 3346
  cyclicGroup, 3346
  dihedralGroup, 3346
  janko2, 3346
  mathieu11, 3346
  mathieu12, 3346
  mathieu22, 3346
  mathieu23, 3346
  mathieu24, 3346
  rubiksGroup, 3346
  symmetricGroup, 3346
  youngGroup, 3346
PGROEB, 3393
  lexGroebner, 3393
  totalGroebner, 3393
phiCoord
  PTPACK, 3370
pi
  EF, 535
PICOERCE, 3355
  coerce, 3355
PiCoercions, 3355
PINTERP, 3437
  interpolate, 3437
PINTERPA, 3440
  LagrangeInterpolation, 3440
placesAbove
  GPAFF, 992
  PAFF, 3215
  PAFFFF, 3223
placesOfDegree
  GPAFF, 992
  INTDIVP, 1387
  PAFF, 3215
  PAFFFF, 3223
PLEQN, 3257
  B1solve, 3257
  bsolve, 3257
  dmp2rfi, 3257
factorset, 3257
hasoln, 3257
inconsistent?, 3257
maxrank, 3257
minrank, 3257
minset, 3257
nextSublist, 3257
overset?, 3257
ParCond, 3257
ParCondList, 3257
pr2dmp, 3257
psolve, 3257
rdregime, 3257
redmat, 3257
redpps, 3257
regime, 3257
se2rfi, 3257
sqfree, 3257
wrregime, 3257
pleskenSplit
  CRFP, 245
plot
  PLOT1, 3357
PLOT1, 3357
  plot, 3357
  plotPolar, 3357
PlotFunctions1, 3357
plotPolar
  PLOT1, 3357
PLOTTOOL, 3359
  calcRanges, 3359
PlotTools, 3359
PLPKCRV, 3386
  eval, 3386
  minimalForm, 3386
  multiplicity, 3386
  pointInIdeal?, 3386
  translateToOrigin, 3386
plusInfinity
  INFINITY, 1215
PMASS, 3297
  assert, 3297
  constant, 3297
  multiple, 3297
  optional, 3297
PMASSFS, 888
INDEX

assert, 888
constant, 888
multiple, 888
optional, 888

pmComplexIntegrate
INTPM, 3306
PMDOWN, 3324
fixPredicate, 3324
patternMatch, 3324
PMFS, 3300
patternMatch, 3300
PMINS, 3303
patternMatch, 3303
pminIntegrate
INTPM, 3306
PMKERNEL, 3314
patternMatch, 3314
PMLSAGG, 3318
patternMatch, 3318
PMPLCAT, 3320
patternMatch, 3320
PMPred, 119
suchThat, 119
PMPredFS, 891
suchThat, 891
PMQFCAT, 3327
patternMatch, 3327
PMSYM, 3332
patternMatch, 3332
PMTOOLS, 3334
patternMatch, 3334
patternMatchTimes, 3334

PNTHEORY, 3442
beroulli, 3442
chebyshevT, 3442
chebyshevU, 3442
cyclotomic, 3442
euler, 3442
fixedDivisor, 3442
hermite, 3442
laguerre, 3442
legendre, 3442
point
TUBETOOL, 4067
pointColorDefault
VIEWDEF, 4161

pointColorPalette
DROPT0, 421
pointDominBy
GPAFF, 992
LPARSPT, 1575
PAFF, 3215
PAFFF, 3223
PointFunctions2, 3367
pointInIdeal?
PLPKCRV, 3386
PointPackage, 3369
pointSizeDefault
VIEWDEF, 4161
PointsOffFiniteOrder, 3373
PointsOffFiniteOrderRational, 3380
PointsOffFiniteOrderTools, 3383
pointToPlace
LPARSPT, 1575
poisson
RIDIST, 3616
pol
INBFF, 1249
polar
COORDSYS, 268
polCase
LEADCDET, 1444
PolK
FFSQFR, 816
PollardSmallFactor
INTFACT, 1293
POLTOPOL, 3390
dmpToHdmp, 3390
dmpToP, 3390
hdmpToDmp, 3390
hdmpToP, 3390
pToDmp, 3390
pToHdmp, 3390
PolToPol, 3390

POLUTIL, 3684
boundOfCauchy, 3684
lazyVariations, 3684
sturmSequence, 3684
sturmVariationsOf, 3684
sylvesterSequence, 3684

POLY2, 3424
map, 3424
<table>
<thead>
<tr>
<th>POLY2UP</th>
<th>3482</th>
</tr>
</thead>
<tbody>
<tr>
<td>univariate</td>
<td>3482</td>
</tr>
<tr>
<td>POLYCATQ</td>
<td>3401</td>
</tr>
<tr>
<td>isExpt</td>
<td>3401</td>
</tr>
<tr>
<td>isPlus</td>
<td>3401</td>
</tr>
<tr>
<td>isPower</td>
<td>3401</td>
</tr>
<tr>
<td>isTimes</td>
<td>3401</td>
</tr>
<tr>
<td>mainVariable</td>
<td>3401</td>
</tr>
<tr>
<td>multivariate</td>
<td>3401</td>
</tr>
<tr>
<td>univariate</td>
<td>3401</td>
</tr>
<tr>
<td>variables</td>
<td>3401</td>
</tr>
<tr>
<td>polygamma</td>
<td></td>
</tr>
<tr>
<td>DFSFUN</td>
<td>369</td>
</tr>
<tr>
<td>FSPECF</td>
<td>872</td>
</tr>
<tr>
<td>PolyGroebner</td>
<td>3393</td>
</tr>
<tr>
<td>POLYLIFT</td>
<td>3398</td>
</tr>
<tr>
<td>map</td>
<td>3398</td>
</tr>
<tr>
<td>PolynomialAN2Expression</td>
<td>3396</td>
</tr>
<tr>
<td>PolynomialCategoryLifting</td>
<td>3398</td>
</tr>
<tr>
<td>PolynomialCategoryQuotientFunctions</td>
<td>3401</td>
</tr>
<tr>
<td>PolynomialComposition</td>
<td>3405</td>
</tr>
<tr>
<td>PolynomialDecomposition</td>
<td>3407</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursion</td>
<td>3410</td>
</tr>
<tr>
<td>PolynomialFactorizationByRecursionUnivariate</td>
<td>3418</td>
</tr>
<tr>
<td>PolynomialFunctions2</td>
<td>3424</td>
</tr>
<tr>
<td>PolynomialGcdPackage</td>
<td>3426</td>
</tr>
<tr>
<td>PolynomialInterpolation</td>
<td>3437</td>
</tr>
<tr>
<td>PolynomialInterpolationAlgorithms</td>
<td>3440</td>
</tr>
<tr>
<td>PolynomialNumberTheoryFunctions</td>
<td>3442</td>
</tr>
<tr>
<td>PolynomialPackageForCurve</td>
<td>3386</td>
</tr>
<tr>
<td>PolynomialRoots</td>
<td>3448</td>
</tr>
<tr>
<td>PolynomialSetUtilitiesPackage</td>
<td>3453</td>
</tr>
<tr>
<td>PolynomialSolveByFormulas</td>
<td>3471</td>
</tr>
<tr>
<td>PolynomialSquareFree</td>
<td>3478</td>
</tr>
<tr>
<td>PolynomialToUnivariatePolynomial</td>
<td>3482</td>
</tr>
<tr>
<td>polynomialZeros</td>
<td></td>
</tr>
<tr>
<td>ESCONT</td>
<td>657</td>
</tr>
<tr>
<td>polyRDE</td>
<td></td>
</tr>
<tr>
<td>RDETR</td>
<td>4027</td>
</tr>
<tr>
<td>polyred</td>
<td></td>
</tr>
<tr>
<td>PFOTOOLS</td>
<td>3383</td>
</tr>
<tr>
<td>polyRicDE</td>
<td></td>
</tr>
<tr>
<td>ODEPRRIC</td>
<td>3512</td>
</tr>
<tr>
<td>ODERTRIC</td>
<td>3671</td>
</tr>
<tr>
<td>polyRing2UPUP</td>
<td></td>
</tr>
<tr>
<td>AFALGRES</td>
<td>9</td>
</tr>
<tr>
<td>polyRingToBlUpRing</td>
<td></td>
</tr>
<tr>
<td>BLUPACK</td>
<td>170</td>
</tr>
<tr>
<td>POLYROOT</td>
<td>3448</td>
</tr>
<tr>
<td>froot</td>
<td>3448</td>
</tr>
<tr>
<td>nthr</td>
<td>3448</td>
</tr>
<tr>
<td>qroot</td>
<td>3448</td>
</tr>
<tr>
<td>rroot</td>
<td>3448</td>
</tr>
<tr>
<td>POLYVEC</td>
<td>4144</td>
</tr>
<tr>
<td>copyfirst</td>
<td>4144</td>
</tr>
<tr>
<td>copyslice</td>
<td>4144</td>
</tr>
<tr>
<td>degree</td>
<td>4144</td>
</tr>
<tr>
<td>differentiate</td>
<td>4144</td>
</tr>
<tr>
<td>divide</td>
<td>4144</td>
</tr>
<tr>
<td>evalat</td>
<td>4144</td>
</tr>
<tr>
<td>extendedgcd</td>
<td>4144</td>
</tr>
<tr>
<td>gcd</td>
<td>4144</td>
</tr>
<tr>
<td>lcm</td>
<td>4144</td>
</tr>
<tr>
<td>mul</td>
<td>4144</td>
</tr>
<tr>
<td>mulbybinomial</td>
<td>4144</td>
</tr>
<tr>
<td>multyscalar</td>
<td>4144</td>
</tr>
<tr>
<td>pow</td>
<td>4144</td>
</tr>
<tr>
<td>remainder</td>
<td>4144</td>
</tr>
<tr>
<td>resultand</td>
<td>4144</td>
</tr>
<tr>
<td>tomodpa</td>
<td>4144</td>
</tr>
<tr>
<td>truncatedmuladd</td>
<td>4144</td>
</tr>
<tr>
<td>truncatedmultiplication</td>
<td>4144</td>
</tr>
<tr>
<td>vectoraddmul</td>
<td>4144</td>
</tr>
<tr>
<td>vectorcombination</td>
<td>4144</td>
</tr>
<tr>
<td>popFortranOutputStack</td>
<td></td>
</tr>
<tr>
<td>FOP</td>
<td>842</td>
</tr>
<tr>
<td>positiveSolve</td>
<td></td>
</tr>
<tr>
<td>ZDSOLVE</td>
<td>4253</td>
</tr>
<tr>
<td>possiblyNewVariety</td>
<td></td>
</tr>
<tr>
<td>PSETPK</td>
<td>3453</td>
</tr>
<tr>
<td>pow</td>
<td></td>
</tr>
<tr>
<td>POLYVEC</td>
<td>4144</td>
</tr>
<tr>
<td>power</td>
<td></td>
</tr>
<tr>
<td>STTAYLOR</td>
<td>3863</td>
</tr>
<tr>
<td>powern</td>
<td></td>
</tr>
<tr>
<td>STTAYLOR</td>
<td>3863</td>
</tr>
<tr>
<td>PowerSeriesLimitPackage</td>
<td>3484</td>
</tr>
<tr>
<td>powersum</td>
<td></td>
</tr>
<tr>
<td>CYCLES</td>
<td>294</td>
</tr>
<tr>
<td>pr2dmp</td>
<td></td>
</tr>
<tr>
<td>PLEQN</td>
<td>3257</td>
</tr>
</tbody>
</table>
INDEX

PREASSOC, 3497
  firstUncouplingMatrix, 3497
PrecomputedAssociatedEquations, 3497
predicate
  PATTERN1, 3289
prepareDecompose
  QCMPACK, 3567
  SFQMPK, 3817
prepareSubResAlgo
  RSETGCD, 3736
prevPrime
  PRIMES, 1326
PRIMARR2, 3500
  map, 3500
  reduce, 3500
  scan, 3500
primaryDecomp
  IDECOMP, 1190
prime?
  GAUSSFAC, 979
  IDECOMP, 1190
  PRIMES, 1326
PRIMELT, 3503
  primitiveElement, 3503
PRIMES, 1326
  nextPrime, 1326
  prevPrime, 1326
  prime?, 1326
  primes, 1326
primes
  PRIMES, 1326
primextendedint
  INTTR, 4001
primextintfrac
  INTTR, 4001
primlimitedintfrac
  INTTR, 4001
primlimitfrac
  INTTR, 4001
primlimitedint
  INTTR, 4001
primlimintfrac
  INTTR, 4001
primlimitedint
  INTTR, 4001
print
  PRIMES, 1326
prinb
  GBINTERN, 1077
principalSubResultantSet
  CAD, 306
prindINFO
  GBINTERN, 1077
prinpolINFO
  GBINTERN, 1077
prinshINFO
  GBINTERN, 1077
PRINT, 3519
  print, 3519
print
  PRINT, 3519
printInfo
  GPAFF, 992
  LPARSPT, 1575
  RSDCMPK, 3728
  SRDCMPK, 3827
printingInfo?
  TBCMPPK, 3947
PrintPackage, 3519
PRJALGPK, 3362
  algebraicSet, 3363
  rationalPoints, 3363
  singularPoints, 3363
  singularPointsWithRestriction, 3363
probablyZeroDim?
  PSETPK, 3453
problemPoints
  D01AGNT, 451
  ESCONT, 657
product
  COMBF, 208
projectionSet
  CAD, 306
ProjectiveAlgebraicSetPackage, 3362
projectivePoint
primitivePart
  SUBRESP, 3906
PrimitiveRatDE, 3507
PrimitiveRatRicDE, 3512
primlimintfrac
PAFF, 3215
PAFFFF, 3223
prolateSpheroidalCOORDSYS, 268
PRS, 3526
?*, 3527
chainSubResultants, 3527
degreeSubResultant, 3527
degreeSubResultantEuclidean, 3527
discriminant, 3527
discriminantEuclidean, 3527
divide, 3527
exquo, 3527
gcd, 3527
indiceSubResultant, 3527
indiceSubResultantEuclidean, 3527
lastSubResultant, 3527
lastSubResultantEuclidean, 3527
Lazard, 3527
Lazard2, 3527
nextSubResultant2, 3527
pseudoDivide, 3527
resultant, 3527
resultantEuclidean, 3527
resultantEuclideanNaïf, 3527
resultantNaïf, 3527
resultantReducit, 3527
resultantReducitEuclidean, 3527
schema, 3527
semiDegreeSubResultantEuclidean, 3527
semiDiscriminantEuclidean, 3527
semiIndiceSubResultantEuclidean, 3527
semiLastSubResultantEuclidean, 3527
semiResultantEuclidean1, 3527
semiResultantEuclidean2, 3527
semiResultantEuclideanNaïf, 3527
semiResultantReducitEuclidean, 3527
semiSubResultantGcdEuclidean1, 3527
semiSubResultantGcdEuclidean2, 3527
subResultantGcd, 3527
subResultantGcdEuclidean, 3527
interReduce, 3453
irreducibleFactors, 3453
lazyIrreducibleFactors, 3453
linear?, 3453
linearPolynomials, 3453
possiblyNewVariety?, 3453
probablyZeroDim?, 3453
quasiMonicPolynomials, 3453
removeIrreducibleRedundantFactors, 3453
removeReducitFactors, 3453
removeReducitFactorsInContents, 3453
removeReducitFactorsInPols, 3453
removeRoughlyReducitFactorsInContents, 3453
removeRoughlyReducitFactorsInPols, 3453
removeSquaresIfCan, 3453
rewriteIdealWithQuasiMonicGenerators, 3453
rewriteSetByReducingWithParticularGenerators, 3453
roughBasicSet, 3453
selectAndPolynomials, 3453
selectOrPolynomials, 3453
selectPolynomials, 3453
squareFreeFactors, 3453
univariate?, 3453
univariatePolynomials, 3453
univariatePolynomialsGcds, 3453
unprotectedRemoveReducitFactors, 3453
PSEUDLIN, 3521
changeBase, 3521
companionBlocks, 3521
normalForm, 3521
pseudoDivide
PRS, 3527
PseudoLinearNormalForm, 3521
PseudoRemainderSequence, 3526
psolve
PLEQN, 3257
PSQFR, 3478
squareFree, 3478
ptFunc
MESH, 1715
PTFUNC2, 3367
map, 3367
pToDmp
   POLTOPOL, 3390
pToHdmp
   POLTOPOL, 3390
PTPACK, 3369
color, 3370
hue, 3370
phiCoord, 3370
rCoord, 3370
thetaCoord, 3370
xCoord, 3370
yCoord, 3370
zCoord, 3370
puiseux
   EXPR2UPS, 704
   GENUPS, 1021
PureAlgebraicIntegration, 3547
PureAlgebraicLODE, 3556
pureLex
   ORDFUNS, 3204
pushdown
   MPRFF, 1742
pushdterm
   MPRFF, 1742
pushFortranOutputStack
   FOP, 842
pushucoef
   MPRFF, 1742
pushuconst
   MPRFF, 1742
pushup
   MPRFF, 1742
PUSHVAR, 3559
PushVariables, 3559
PWFFINTB, 3247
   integralBasis, 3247
   localIntegralBasis, 3247
   reducedDiscriminant, 3247
QALGSET2, 3563
   radicalSimplify, 3563
QCMPACK, 3567
   algebraicSort, 3567
   branchIfCan, 3567
   infRittWu?, 3567
   internalInfRittWu?, 3567
   internalSubPolSet?, 3567
   internalSubQuasiComponent?, 3567
   moreAlgebraic?, 3567
   prepareDecompose, 3567
   removeSuperfluousCases, 3567
   removeSuperfluousQuasiComponents, 3567
   startTable, 3567
   subCase?, 3567
   subPolSet?, 3567
   subQuasiComponent?, 3567
   subTriSet?, 3567
   supDimElseRittWu?, 3567
qfactor
   FSUPFACT, 942
QFCAT2, 3576
   map, 3576
qPot
   INBFF, 1249
qqq
   WEIER, 4172
qroot
   POLYROOT, 3448
qShiftAction
   FFFG, 852
qShiftC
   FFFG, 852
quadratic
   SOLVEFOR, 3471
quadratic?
   E04AGNT, 721
quadraticBezier
   BEZIER, 162
quadraticNorm
   GALFACTU, 948
quadTransform
   BLUPPACK, 170
quartic
   SOLVEFOR, 3471
QuasiAlgebraicSet2, 3563
QuasiComponentPackage, 3567
quasiMonicPolynomials
   PSETPK, 3453
QUATCT2, 3579
   map, 3579
QuaternionCategoryFunctions2, 3579
quickSort
   FLASORT, 823
QuotientFieldCategoryFunctions2, 3576
quotVecSpaceBasis
   LOP, 1557
radical
   IDECOMP, 1190
RadicalEigenPackage, 3582
radicalEigenvalues
   REP, 3582
radicalEigenvectors
   REP, 3582
radicalOfLeftTraceForm
   ALGPKG, 44
radicalRoots
   SOLVERAD, 3600
radicalSimplify
   QALGSET2, 3563
radicalSolve
   SOLVERAD, 3600
RadicalSolvePackage, 3600
radix
   RADUTIL, 3608
RadixUtilities, 3608
radPoly
   CHVAR, 188
RADUTIL, 3608
   radix, 3608
raisePolynomial
   FACUTIL, 738
ran
   FACUTIL, 738
randnum
   RANDSRC, 3619
random
   FFPOLY, 788
   INBFF, 1249
RandomDistributions, 3610
RandomFloatDistributions, 3613
RandomIntegerDistributions, 3616
RandomNumberSource, 3619
randomR
   GENPGCD, 1007
   PFBR, 3410
PFBRU, 3418
RANDSRC, 3619
   randnum, 3619
   reseed, 3619
   seed, 3619
   size, 3619
rangesFinite
   D01AGNT, 451
rangePascalTriangle
   GALUTIL, 975
ranges
   DROPT0, 421
rank
   LSMP, 1540
   LSMP1, 1543
   MATLIN, 1652
ratDenom
   ALGMANIP, 35
ratDsolve
   ODERAT, 3662
RATFACT, 3622
   factor, 3622
   factorSquareFree, 3622
rational
   RATRET, 3669
rational?
   RATRET, 3669
RationalFactorize, 3622
RationalFunction, 3625
RationalFunctionDefiniteIntegration, 3628
RationalFunctionFactor, 3631
RationalFunctionFactorizer, 3634
RationalFunctionIntegration, 3636
RationalFunctionLimitPackage, 3639
RationalFunctionSign, 3644
RationalFunctionSum, 3652
rationalIfCan
   RATRET, 3669
RationalIntegration, 3655
RationalInterpolation, 3658
RationalLODE, 3662
rationalPlaces
   GPAFF, 992
   PAFF, 3215
   PAFFF, 3223
rationalPoints
INDEX

GPAFF, 992
PAFF, 3215
PAFFS, 3223
PRJALGPK, 3363
RationalRetractions, 3669
RationalRicDE, 3671
RationalUnivariateRepresentationPackage, 3679
ratPoly
ALGMANIP, 35
RATRET, 3669
rational, 3669
rational?, 3669
rationalIfCan, 3669
rCoord
PTPACK, 3370
RDDEFS, 599
rischDE, 599
rischDEsys, 599
RDEETR, 4027
baseRDE, 4027
monomRDE, 4027
polyRDE, 4027
RDEETRS, 4032
baseRDEsys, 4032
monomRDEsys, 4032
rdHack1
RDIST, 3610
RDIST, 3610
rdHack1, 3610
uniform, 3610
weighted, 3610
RDIV, 3719
order, 3719
rdregime
PLEQN, 3257
real
CTRIGMNIP, 261
TRIGMNIP, 4062
REAL0, 3693
midpoint, 3693
midpoints, 3693
realZeros, 3693
refine, 3701
real?
TRIGMNIP, 4062
realEigenvalues
NREP, 3172
realEigenvectors
FLOATRP, 834
REALSOLV, 3690
realSolve, 3690
solve, 3690
realSolve
REALSOLV, 3690
ZDSOLVE, 4253
RealSolvePackage, 3690
RealZeroPackage, 3693
RealZeroPackageQ, 3701
realZeros
REAL0, 3693
REALOQ, 3701
recip
NORMPK, 3100
STTAYLOR, 3863
reciprocalPolynomial
CRFP, 245
recolor
DRAWCFUN, 3978
RECOUP, 3707
evalADE, 3707
evalRec, 3707
getEq, 3707
getOp, 3707
getShiftRec, 3707
numberOfValuesNeeded, 3707
shiftInfoRec, 3707
RectangularMatrixCategoryFunctions, 3704
recur
MAPHACK1, 1600
MAPPKG1, 1614
RecurrenceOperator, 3707
redmat
PLEQN, 3257
REDORDER, 3725
    ReduceOrder, 3725
redPo
    GBINTERN, 1077
redPol
    GBINTERN, 1077
redpps
    PLEQN, 3257
reduce
    ARRAY12, 3189
    DEGRED, 321
    DIRPROD2, 339
    FLAGG2, 820
    FSAGG2, 827
    LIST2, 1566
    MATCAT2, 1646
    PRIMARR2, 3500
    RMCAT2, 3704
    STREAM2, 3853
    VECTOR2, 4158
reducedDiscriminant
    PWFFINTB, 3247
ReducedDivisor, 3719
reducedQPowers
    FFPOLY, 788
reduceLineOverLine
    LOP, 1557
ReduceLODE, 3722
reduceLODE
    ODERED, 3722
ReduceOrder
    REDORDER, 3725
reduceRow
    LOP, 1557
reduceRowOnList
    LOP, 1557
reduction
    GENEEZ, 1027
    GHENSEL, 984
    INMODGCD, 1232
ReductionOfOrder, 3725
refine
    FRUTIL, 735
    REAL0, 3693
    REAL0Q, 3701
regime
    PLEQN, 3257
    RegularSetDecompositionPackage, 3728
    RegularTriangularSetGcdPackage, 3736
    removeConstantTerm
        INTTOOLS, 1367
    removeCoshSq
        TRMANIP, 4012
    removeCosSq
        TRMANIP, 4012
    removeIrreducibleRedundantFactors
        PSETPK, 3453
    removeRedundantFactors
        PSETPK, 3453
    removeRedundantFactorsInContents
        PSETPK, 3453
    removeRedundantFactorsInPols
        PSETPK, 3453
    removeRoughlyRedundantFactorsInContents
        PSETPK, 3453
    removeRoughlyRedundantFactorsInPol
        PSETPK, 3453
    removeRoughlyRedundantFactorsInPols
        PSETPK, 3453
    removeSinhSq
        TRMANIP, 4012
    removeSinSq
        TRMANIP, 4012
    removeSquaresIfCan
        PSETPK, 3453
    removeSuperfluousCases
        QCMPACK, 3567
        SFQCMPK, 3817
    removeSuperfluousQuasiComponents
        QCMPACK, 3567
        SFQCMPK, 3817
REP, 3582
    eigenMatrix, 3582
    gramSchmidt, 3582
    normalise, 3582
    orthonormalBasis, 3582
    radicalEigenvalues, 3582
    radicalEigenvectors, 3582
REP1, 3750
    antisymmetricTensors, 3750
    createGenericMatrix, 3750
    permutationRepresentation, 3750
INDEX

symmetricTensors, 3750

tensorProduct, 3750

REP2, 3758

double, 3745

RepeatedDoubling, 3744

RepeatedSquaring, 3747

replaceDiffs

EXPRSOL, 683

replaceVarByOne

PFORP, 3232

replaceVarByZero

PFORP, 3232

RepresentationPackage1, 3750

RepresentationPackage2, 3758

REPSQ, 3747

expt, 3747

repSq

INBFF, 1249

reseed

RANDSRC, 3619

reset

GPAFF, 992

resetVariableOrder

UDVO, 4138

reshape

CARTEN2, 186

MPC2, 1747

RESLATC, 3775

c coerce, 3775

ResolveLatticeCompletion, 3775

restorePrecision

NAGSP, 2193

resultant

POLYVEC, 4144

PRS, 3527

resultantEuclidean

PRS, 3527

resultantEuclideannaif

PRS, 3527

resultantnaif

PRS, 3527

resultantReduit

PRS, 3527

resultantReduitEuclidean

PRS, 3527

resultantSet

CAD, 306

retract

ANY1, 98

retractable?

ANY1, 98

retractIfCan

ANY1, 98

NORMRETR, 3108

RetractSolvePackage, 3777

RETSOL, 3777
	solveRetract, 3778

reverse

GALPOLYU, 971

reverseLex

ORDFUNS, 3204

revert

STTAYLOR, 3863

rewriteIdealWithQuasiMonicGenerators

PSETPK, 3453

rewriteSetByReducingWithParticularGenerators

PSETPK, 3453

RF, 3625

c coerce, 3625

eval, 3625

mainVariable, 3625

multivariate, 3625

univariate, 3625

variables, 3625

RF2UTS

UTSODETL, 4141

RFDIST, 3613

Beta, 3613

chiSquare, 3613

chiSquare1, 3613

exponential, 3613

exponential1, 3613
INDEX

NUMODE, 3144

rk4qc
NUMODE, 3144

RMCAT2, 3704
map, 3704
reduce, 3704

romberg
NUMQUAD, 3154

rombergo
NUMQUAD, 3154

rootBound
gALFACTU, 948

rootKerSimp
ALGMANIP, 35

rootNormalize
EFSTRUC, 579

rootOf
AF, 14

rootOfIrreduciblePoly
FFPOLY2, 809

rootPoly
CHVAR, 188

rootPower
ALGMANIP, 35

rootProduct
ALGMANIP, 35

rootRadius
CRFP, 245

RootsFindingPackage, 3781

rootSimp
ALGMANIP, 35

rootSplit
ALGMANIP, 35

roughBasicSet
PSETPK, 3453

rowEch
MHROWRED, 1725

crowEchelon
IMATQF, 1229
MATLIN, 1652
MHROWRED, 1725

rowEchelonLocal
MHROWRED, 1725

rowEchLocal
MHROWRED, 1725

rowEchWoZeroLines
<table>
<thead>
<tr>
<th>INDEX</th>
<th>4355</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOP, 1557</td>
<td></td>
</tr>
<tr>
<td>rowEchWoZeroLinesWOVectorise row</td>
<td></td>
</tr>
<tr>
<td>rows</td>
<td></td>
</tr>
<tr>
<td>MAMA, 1703</td>
<td></td>
</tr>
<tr>
<td>rroot</td>
<td></td>
</tr>
<tr>
<td>POLYROOT, 3448</td>
<td></td>
</tr>
<tr>
<td>RSDCMPK, 3728</td>
<td></td>
</tr>
<tr>
<td>algebraicDecompose, 3728</td>
<td></td>
</tr>
<tr>
<td>convert, 3728</td>
<td></td>
</tr>
<tr>
<td>decompose, 3728</td>
<td></td>
</tr>
<tr>
<td>internalDecompose, 3728</td>
<td></td>
</tr>
<tr>
<td>KrullNumber, 3728</td>
<td></td>
</tr>
<tr>
<td>numberOfVariables, 3728</td>
<td></td>
</tr>
<tr>
<td>printInfo, 3728</td>
<td></td>
</tr>
<tr>
<td>transcendentalDecompose, 3728</td>
<td></td>
</tr>
<tr>
<td>upDateBranches, 3728</td>
<td></td>
</tr>
<tr>
<td>RSETGCD, 3736</td>
<td></td>
</tr>
<tr>
<td>integralLastSubResultant, 3736</td>
<td></td>
</tr>
<tr>
<td>prepareSubResAlgo, 3736</td>
<td></td>
</tr>
<tr>
<td>startTableGcd, 3736</td>
<td></td>
</tr>
<tr>
<td>startTableInvSet, 3736</td>
<td></td>
</tr>
<tr>
<td>stopTableGcd, 3736</td>
<td></td>
</tr>
<tr>
<td>stopTableInvSet, 3736</td>
<td></td>
</tr>
<tr>
<td>toseInvertible?, 3736</td>
<td></td>
</tr>
<tr>
<td>toseInvertibleSet, 3736</td>
<td></td>
</tr>
<tr>
<td>toseLastSubResultant, 3736</td>
<td></td>
</tr>
<tr>
<td>toseSquareFreePart, 3736</td>
<td></td>
</tr>
<tr>
<td>rubiksGroup</td>
<td></td>
</tr>
<tr>
<td>PGE, 3346</td>
<td></td>
</tr>
<tr>
<td>rur</td>
<td></td>
</tr>
<tr>
<td>IRURPK, 1374</td>
<td></td>
</tr>
<tr>
<td>RURPK, 3679</td>
<td></td>
</tr>
<tr>
<td>RURPK, 3679</td>
<td></td>
</tr>
<tr>
<td>rur, 3679</td>
<td></td>
</tr>
<tr>
<td>s01eaf</td>
<td></td>
</tr>
<tr>
<td>NAGS, 3058</td>
<td></td>
</tr>
<tr>
<td>s13aaf</td>
<td></td>
</tr>
<tr>
<td>NAGS, 3058</td>
<td></td>
</tr>
<tr>
<td>s13acf</td>
<td></td>
</tr>
<tr>
<td>NAGS, 3058</td>
<td></td>
</tr>
<tr>
<td>s13adf</td>
<td></td>
</tr>
<tr>
<td>NAGS, 3058</td>
<td></td>
</tr>
<tr>
<td>s14aaf</td>
<td></td>
</tr>
<tr>
<td>NAGS, 3058</td>
<td></td>
</tr>
</tbody>
</table>
s19aaf  
    NAGS, 3058
s19abf  
    NAGS, 3058
s19acf  
    NAGS, 3058
s19adf  
    NAGS, 3058
s20aef  
    NAGS, 3058
s20adf  
    NAGS, 3058
s21baf  
    NAGS, 3058
s21bbf  
    NAGS, 3058
s21bcf  
    NAGS, 3058
s21bdf  
    NAGS, 3058
SAEFACT, 3795
    factor, 3795
    SAERationalFunctionAlgFactor, 3786
    SAERFFC, 3786
    factor, 3786
safeCeiling  
    GALUTIL, 975
safeFloor  
    GALUTIL, 975
safetyMargin  
    GALUTIL, 975
sampleDotGraph  
    GRAPHVIZ, 1057
satisfy?  
    PATTERN1, 3289
say  
    DISPLAY, 345
sayLength  
    DISPLAY, 345
SCACHE, 3805
    cache, 3805
    clearCache, 3805
    enterInCache, 3805
scaleRoots  
    GALPOLYU, 971
scan  
    ARRAY12, 3189
    DIRPROD2, 339
    FLAGG2, 820
    FSAGG2, 827
    LIST2, 1566
    PRIMARR2, 3500
    STREAM2, 3853
    VECTOR2, 4158
ScanArabic  
    NUMFMT, 3122
ScanFloatIgnoreSpaces  
    NUMFMT, 3122
ScanFloatIgnoreSpacesIfCan  
    NUMFMT, 3122
scanOneDimSubspaces  
    REP2, 3758
ScanRoman  
    NUMFMT, 3122
    schema
    PRS, 3527
    schwerpunkt
    CRFP, 245
SCPKG, 3893
    coordinates, 3893
    structuralConstants, 3893
screenResolution  
    GRDEF, 1052
ScriptFormulaFormat1, 3788
sdf2lst  
    D01AGNT, 451
    ESCONT, 657
    ESTOOLS, 667
se2rfi  
    PLEQN, 3257
sec  
    EF, 535
    STTF, 3876
    STTFNC, 3886
sec2cos  
    TRMANIP, 4012
sech  
    EF, 535
sech2cosh  
    TRMANIP, 4012
seed  
    RANDSRC, 3619
INDEX

SEG2, 3792
  map, 3792
SEGBIND2, 3790
  map, 3790
SegmentBindingFunctions2, 3790
SegmentFunctions2, 3792
selectAndPolynomials
  PSETPK, 3453
selectOrPolynomials
  PSETPK, 3453
selectPolynomials
  PSETPK, 3453
semiDegreeSubResultantEuclidean
  PRS, 3527
semiDiscriminantEuclidean
  PRS, 3527
semiIndiceSubResultantEuclidean
  PRS, 3527
semiLastSubResultantEuclidean
  PRS, 3527
semiResultantEuclidean1
  PRS, 3527
semiResultantEuclidean2
  PRS, 3527
semiResultantEuclideannaif
  PRS, 3527
semiResultantReducitEuclidean
  PRS, 3527
semiSubResultantGcdEuclidean1
  PRS, 3527
semiSubResultantGcdEuclidean2
  PRS, 3527
separateDegrees
  DDFACT, 349
separateFactors
  DDFACT, 349
  MDDFACT, 1719
sequences
  PARTPERM, 3285
series
  EXPR2UPS, 704
  GENUPS, 1021
seriesSolve
  EXPRODE, 691
  EXPRSOL, 683
  UTSSOL, 3954
setClipValue
  DRAW CX, 414
setCurve
  GPAFF, 992
  PAFF, 3215
  PAFFFF, 3223
setErrorBound
  CRFP, 245
setFieldInfo
  INBFF, 1249
setFoundZeroes
  RFP, 3781
setImagSteps
  DRAW CX, 414
setLegalFortranSourceExtensions
  FORT, 846
setOrder
  UDPO, 4135
setRealSteps
  DRAW CX, 414
setSingularPoints
  GPAFF, 992
  PAFF, 3215
  PAFFFF, 3223
setVariableOrder
  UDVO, 4138
SFQCMPK, 3816
  algebraicSort, 3817
  branchIfCan, 3817
  infRittWu?, 3817
  internalInfRittWu?, 3817
  internalSubPolSet?, 3817
  internalSubQuasiComponent?, 3817
  moreAlgebraic?, 3817
  prepareDecompose, 3817
  removeSuperfluousCases, 3817
  removeSuperfluousQuasiComponents, 3817
startTable, 3817
stopTable, 3817
subCase?, 3817
subPolSet?, 3817
subQuasiComponent?, 3817
subTriSet?, 3817
supDimElseRittWu?, 3817
SFRGCD, 3834
startTableGcd, 3835
<table>
<thead>
<tr>
<th>Function/Method</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>startTableInvSet</td>
<td>3835</td>
</tr>
<tr>
<td>stopTableGcd</td>
<td>3835</td>
</tr>
<tr>
<td>stopTableInvSet</td>
<td>3835</td>
</tr>
<tr>
<td>stoseIntegralLastSubResultant</td>
<td>3835</td>
</tr>
<tr>
<td>stoseInternalLastSubResultant</td>
<td>3835</td>
</tr>
<tr>
<td>stoseInvertible?</td>
<td>3835</td>
</tr>
<tr>
<td>stoseInvertible?reg</td>
<td>3835</td>
</tr>
<tr>
<td>stoseInvertible?sqfreg</td>
<td>3835</td>
</tr>
<tr>
<td>stoseInvertibleSetreg</td>
<td>3835</td>
</tr>
<tr>
<td>stoseInvertibleSetsqfreg</td>
<td>3835</td>
</tr>
<tr>
<td>stoseLastSubResultant</td>
<td>3835</td>
</tr>
<tr>
<td>stosePrepareSubResAlgo</td>
<td>3835</td>
</tr>
<tr>
<td>stoseSquareFreePartAlgo</td>
<td>3835</td>
</tr>
<tr>
<td>SFunction</td>
<td>CYCLES, 294</td>
</tr>
<tr>
<td>SGCF</td>
<td>3927</td>
</tr>
<tr>
<td>coleman</td>
<td>3927</td>
</tr>
<tr>
<td>inverseColeman</td>
<td>3927</td>
</tr>
<tr>
<td>listYoungTableaus</td>
<td>3927</td>
</tr>
<tr>
<td>makeYoungTableau</td>
<td>3927</td>
</tr>
<tr>
<td>nextColeman</td>
<td>3927</td>
</tr>
<tr>
<td>nextLatticePermutation</td>
<td>3927</td>
</tr>
<tr>
<td>nextPartition</td>
<td>3927</td>
</tr>
<tr>
<td>numberOfImproperPartitions</td>
<td>3927</td>
</tr>
<tr>
<td>subSet</td>
<td>3927</td>
</tr>
<tr>
<td>unrankImproperPartitions0</td>
<td>3927</td>
</tr>
<tr>
<td>unrankImproperPartitions1</td>
<td>3927</td>
</tr>
<tr>
<td>shade</td>
<td>PTPACK, 3370</td>
</tr>
<tr>
<td>shanksDiscLogAlgorithm</td>
<td>DLP, 342</td>
</tr>
<tr>
<td>shellSort</td>
<td>FLASORT, 823</td>
</tr>
<tr>
<td>ShiftAction</td>
<td>FFFG, 852</td>
</tr>
<tr>
<td>ShiftC</td>
<td>FFFG, 852</td>
</tr>
<tr>
<td>shiftHP</td>
<td>GUESS, 1125</td>
</tr>
<tr>
<td>GUESSAN</td>
<td>1162</td>
</tr>
<tr>
<td>GUESSF</td>
<td>1164</td>
</tr>
<tr>
<td>GUESSINT</td>
<td>1169</td>
</tr>
<tr>
<td>GUESSP</td>
<td>1171</td>
</tr>
<tr>
<td>GUESSUP</td>
<td>1174</td>
</tr>
<tr>
<td>shiftInfoRec</td>
<td>RECOP, 3707</td>
</tr>
<tr>
<td>shiftRoots</td>
<td>GALPOLYU, 971</td>
</tr>
<tr>
<td>showFortranOutputStack</td>
<td>FOP, 842</td>
</tr>
<tr>
<td>SHP</td>
<td>3897</td>
</tr>
<tr>
<td>countRealRoots</td>
<td>3897</td>
</tr>
<tr>
<td>countRealRootsMultiple</td>
<td>3897</td>
</tr>
<tr>
<td>SturmHabicht</td>
<td>3897</td>
</tr>
<tr>
<td>SturmHabichtCoefficients</td>
<td>3897</td>
</tr>
<tr>
<td>SturmHabichtMultiple</td>
<td>3897</td>
</tr>
<tr>
<td>SturmHabichtSequence</td>
<td>3897</td>
</tr>
<tr>
<td>subresultantSequence</td>
<td>3897</td>
</tr>
<tr>
<td>shuffle</td>
<td>PARTPERM, 3285</td>
</tr>
<tr>
<td>shufflein</td>
<td>PARTPERM, 3285</td>
</tr>
<tr>
<td>Si</td>
<td>LF, 1560</td>
</tr>
<tr>
<td>sign</td>
<td>SIGNEF, 574</td>
</tr>
<tr>
<td>SIGNRF</td>
<td>3644</td>
</tr>
<tr>
<td>TOOLSIGN</td>
<td>3963</td>
</tr>
<tr>
<td>signAround</td>
<td>INPSIGN, 1269</td>
</tr>
<tr>
<td>SIGNEF</td>
<td>574</td>
</tr>
<tr>
<td>sign</td>
<td>574</td>
</tr>
<tr>
<td>SIGNRF</td>
<td>3644</td>
</tr>
<tr>
<td>sign</td>
<td>3644</td>
</tr>
<tr>
<td>SIMPAN</td>
<td>3797</td>
</tr>
<tr>
<td>simplify</td>
<td>3797</td>
</tr>
<tr>
<td>SimpleAlgebraicExtensionAlgFactor</td>
<td>3795</td>
</tr>
<tr>
<td>simpleBounds?</td>
<td>E04AGNT, 721</td>
</tr>
<tr>
<td>simplify</td>
<td>SIMPAN, 3797</td>
</tr>
<tr>
<td>TRMANIP</td>
<td>4012</td>
</tr>
<tr>
<td>SimplifyAlgebraicNumberConvertPackage</td>
<td>3797</td>
</tr>
<tr>
<td>simplifyExp</td>
<td>TRMANIP, 4012</td>
</tr>
<tr>
<td>simpson</td>
<td>NUMQUAD, 3154</td>
</tr>
<tr>
<td>simpsonano</td>
<td>NUMQUAD, 3154</td>
</tr>
</tbody>
</table>
INDEX

sin
EF, 535
STTF, 3876
STTFNC, 3886
sin2csc
TRMANIP, 4012
sincos
STTF, 3876
singleFactorBound
GALFACTU, 948
singRicDE
ODEPRRIC, 3512
ODERTRIC, 3671
singularitiesOf
D01AGNT, 451
ESCONI, 657
singularPoints
PAFF, 992
PAFF, 3215
PAFFFF, 3223
PRJALGPK, 3363
singularPointsWithRestriction
PRJALGPK, 3363
sinh
EF, 535
sinh2csch
TRMANIP, 4012
sinhcosh
STTF, 3876
size
RANDSRC, 3619
sizeMultiplication
FFF, 773
sizePascalTriangle
GALUTIL, 975
skewSFunction
CYCLES, 294
slex
TABLBUMP, 3942
slope
NPOLYGON, 3080
SMITH, 3799
completeHermite, 3799
completeSmith, 3799
diophantineSystem, 3799
hermite, 3799
smith, 3799
smith
SMITH, 3799
SmithNormalForm, 3799
sn
ELFUTS, 611
snccdn
ELFUTS, 611
socf2socdf
ESTOOLS, 667
solve
FLOATRP, 834
LODEEF, 560
LSMP, 1540
LSMP1, 1543
NLINSOL, 3096
NODEI, 3091
ODEEF, 567
ODEPACK, 81
ODESYS, 3913
PDEPACK, 91
REALSOLV, 3690
SOLVEFOR, 3471
SOLVENETRA, 4043
SYSSOLP, 3920
solve1
INEP, 1258
SOLVEFOR, 3471
aCubic, 3471
aLinear, 3471
aQuadratic, 3471
aQuartic, 3471
cubic, 3471
linear, 3471
mapSolve, 3471
particularSolution, 3471
quadratic, 3471
quartic, 3471
solve, 3471
solveid
GENEEZ, 1027
solveInField
NLINSOL, 3096
ODESYS, 3913
solveLinear
LINDEP, 1523
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>solveLinearlyOverQ</td>
<td>1304</td>
</tr>
<tr>
<td>ZLINDEP,</td>
<td></td>
</tr>
<tr>
<td>solveLinearPolynomialEquation</td>
<td>237</td>
</tr>
<tr>
<td>CINTSLPE,</td>
<td>813</td>
</tr>
<tr>
<td>FFSLPE,</td>
<td></td>
</tr>
<tr>
<td>INTSLPE,</td>
<td>1344</td>
</tr>
<tr>
<td>solveLinearPolynomialEquationByFractions</td>
<td>1534</td>
</tr>
<tr>
<td>LPEFRAC,</td>
<td></td>
</tr>
<tr>
<td>solveLinearPolynomialEquationByRecursion</td>
<td>3410</td>
</tr>
<tr>
<td>PFBAD,</td>
<td>3418</td>
</tr>
<tr>
<td>PFBRU,</td>
<td></td>
</tr>
<tr>
<td>SOLVERAD,</td>
<td>3600</td>
</tr>
<tr>
<td>contractSolve,</td>
<td>3600</td>
</tr>
<tr>
<td>radicalRoots,</td>
<td>3600</td>
</tr>
<tr>
<td>radicalSolve,</td>
<td>3600</td>
</tr>
<tr>
<td>solveRetract</td>
<td>3778</td>
</tr>
<tr>
<td>RETSOL,</td>
<td></td>
</tr>
<tr>
<td>SOLVESER,</td>
<td>4055</td>
</tr>
<tr>
<td>decomposeFunc,</td>
<td>4055</td>
</tr>
<tr>
<td>unvectorise,</td>
<td>4055</td>
</tr>
<tr>
<td>SOLVETRA,</td>
<td>4043</td>
</tr>
<tr>
<td>solve,</td>
<td>4043</td>
</tr>
<tr>
<td>sortConstraints</td>
<td>721</td>
</tr>
<tr>
<td>E04AGNT,</td>
<td></td>
</tr>
<tr>
<td>SortedCache,</td>
<td>3805</td>
</tr>
<tr>
<td>SortPackage,</td>
<td>3808</td>
</tr>
<tr>
<td>SORTPAK,</td>
<td>3808</td>
</tr>
<tr>
<td>bubbleSort,</td>
<td>3808</td>
</tr>
<tr>
<td>insertionSort,</td>
<td>3808</td>
</tr>
<tr>
<td>space,</td>
<td>421</td>
</tr>
<tr>
<td>DROPT0,</td>
<td></td>
</tr>
<tr>
<td>SparseUnivariatePolynomialFunctions2,</td>
<td>3811</td>
</tr>
<tr>
<td>sparsityIF,</td>
<td></td>
</tr>
<tr>
<td>D02AGNT,</td>
<td>490</td>
</tr>
<tr>
<td>specialise</td>
<td></td>
</tr>
<tr>
<td>CAD,</td>
<td>306</td>
</tr>
<tr>
<td>SpecialOutputPackage,</td>
<td>3813</td>
</tr>
<tr>
<td>specialTrigs</td>
<td></td>
</tr>
<tr>
<td>EF,</td>
<td>535</td>
</tr>
<tr>
<td>SPECOUT,</td>
<td>3813</td>
</tr>
<tr>
<td>outputAsFortran,</td>
<td>3813</td>
</tr>
<tr>
<td>outputAsScript,</td>
<td>3813</td>
</tr>
<tr>
<td>outputAsTex,</td>
<td>3813</td>
</tr>
<tr>
<td>spherical</td>
<td></td>
</tr>
<tr>
<td>COORDSYS,</td>
<td>268</td>
</tr>
<tr>
<td>split,</td>
<td></td>
</tr>
<tr>
<td>ALGFACKT,</td>
<td>55</td>
</tr>
<tr>
<td>IR2F,</td>
<td>1361</td>
</tr>
<tr>
<td>IRRF2F,</td>
<td>1358</td>
</tr>
<tr>
<td>MONOTOOL,</td>
<td>1734</td>
</tr>
<tr>
<td>REP2,</td>
<td>3758</td>
</tr>
<tr>
<td>splitConstant,</td>
<td></td>
</tr>
<tr>
<td>INTPM,</td>
<td>3306</td>
</tr>
<tr>
<td>splitDenominator</td>
<td></td>
</tr>
<tr>
<td>COMBF,</td>
<td>221</td>
</tr>
<tr>
<td>ICDEN,</td>
<td>1220</td>
</tr>
<tr>
<td>MCDEN,</td>
<td>1649</td>
</tr>
<tr>
<td>ODEPRIM,</td>
<td>3507</td>
</tr>
<tr>
<td>UPCDEN,</td>
<td>4101</td>
</tr>
<tr>
<td>splitLinear</td>
<td></td>
</tr>
<tr>
<td>E04AGNT,</td>
<td>721</td>
</tr>
<tr>
<td>splitSquarefree</td>
<td></td>
</tr>
<tr>
<td>MONOTOOL,</td>
<td>1734</td>
</tr>
<tr>
<td>sPol,</td>
<td></td>
</tr>
<tr>
<td>GBINTERN,</td>
<td>1077</td>
</tr>
<tr>
<td>sqfree,</td>
<td></td>
</tr>
<tr>
<td>PLEQN,</td>
<td>3257</td>
</tr>
<tr>
<td>squareFree</td>
<td></td>
</tr>
<tr>
<td>INTFACT,</td>
<td>1293</td>
</tr>
<tr>
<td>MULTSQFR,</td>
<td>1782</td>
</tr>
<tr>
<td>PSQFR,</td>
<td>3478</td>
</tr>
<tr>
<td>SUPFRACF,</td>
<td>3910</td>
</tr>
<tr>
<td>UPSQFREE,</td>
<td>4115</td>
</tr>
<tr>
<td>ZDSOLVE,</td>
<td>4253</td>
</tr>
<tr>
<td>squareFreeBasis</td>
<td></td>
</tr>
<tr>
<td>CADU,</td>
<td>311</td>
</tr>
<tr>
<td>squareFreeFactors</td>
<td></td>
</tr>
<tr>
<td>PSETPK,</td>
<td>3453</td>
</tr>
<tr>
<td>squareFreeLexTriangular</td>
<td></td>
</tr>
<tr>
<td>LEPTRIJK,</td>
<td>1517</td>
</tr>
<tr>
<td>squareFreePart</td>
<td></td>
</tr>
<tr>
<td>UPSQFREE,</td>
<td>4115</td>
</tr>
<tr>
<td>squareFreePrim</td>
<td></td>
</tr>
<tr>
<td>MULTSQFR,</td>
<td>1782</td>
</tr>
<tr>
<td>SquareFreeQuasiComponentPackage,</td>
<td>3816</td>
</tr>
<tr>
<td>SquareFreeRegularSetDecompositionPackage,</td>
<td>3827</td>
</tr>
<tr>
<td>SquareFreeRegularTriangularSetGcdPackage,</td>
<td>3834</td>
</tr>
<tr>
<td>SRDCMPK,</td>
<td>3827</td>
</tr>
<tr>
<td>algebraicDecompose,</td>
<td></td>
</tr>
<tr>
<td>convert,</td>
<td>3827</td>
</tr>
</tbody>
</table>
INDEX

decompose, 3827
internalDecompose, 3827
KrullNumber, 3827
numberOfVariables, 3827
printInfo, 3827
transcendentalDecompose, 3827
upDateBranches, 3827
standardBasisOfCyclicSubmodule
REP2, 3758
standardDotHeader
GRAPHVIZ, 1057
startPolynomial
CRFP, 245
stepBlowUp
BLUPPACK, 170
stFunc1
UTSODE, 4128
stFunc2
UTSODE, 4128
stFuncN
UTSODE, 4128
stiffnessAndStabilityFactor
D02AGNT, 490
stiffnessAndStabilityOfODEIF
D02AGNT, 490
STINPROD, 3859
evenInfiniteProduct, 3859
generalInfiniteProduct, 3859
infiniteProduct, 3859
oddInfiniteProduct, 3859
stirling1
COMBINAT, 1289
stirling2
COMBINAT, 1289
STNSR, 3873
tensorMap, 3873
stopMusserTrials
GALFACT, 954
StorageEfficientMatrixOperations, 3845
stoseIntegralLastSubResultant
SFRGCD, 3835
stoseInternalLastSubResultant
SFRGCD, 3835
stoseInvertible?
SFRGCD, 3835
stoseInvertible?reg
SFRGCD, 3835
stoseInvertible?sqfreg
SFRGCD, 3835
stoseLastSubResultant
SFRGCD, 3835
stosePrepareSubResAlgo
SFRGCD, 3835
stoseSquareFreePart
SFRGCD, 3835
STREAM1, 3851
concat, 3851
STREAM2, 3853
map, 3853
reduce, 3853
scan, 3853
STREAM3, 3856
map, 3856
StreamFunctions1, 3851
StreamFunctions2, 3853
StreamFunctions3, 3856
StreamInfiniteProduct, 3859
StreamTaylorSeriesOperations, 3862
StreamTensor, 3873
StreamTranscendentalFunctions, 3875
StreamTranscendentalFunctionsNonCommutative, 3886
stripCommentsAndBlanks
TEMUTL, 3958
structuralConstants
SCPKG, 3893
StructuralConstantsPackage, 3893
sts2stst
WEIER, 4172
STTAYLOR, 3862
?. 3863
*?, 3863
+?, 3863
-?, 3863
/?, 3863
addiag, 3863
coerce, 3863
compose, 3863
deriv, 3863
eval, 3863
evenlambert, 3863
equivo, 3863
gderiv, 3863
generalLambert, 3863
int, 3863
terms, 3863
integrate, 3863
invmultisect, 3863
lagrange, 3863
lambert, 3863
lazyGintegrate, 3863
lazyIntegrate, 3863
mapdiv, 3863
mapmult, 3863
monom, 3863
multisect, 3863
nlde, 3863
oddintegers, 3863
oddLambert, 3863
power, 3863
powers, 3863
recip, 3863
revert, 3863
STTF, 3875
STTFNC, 3886
acos, 3886
acot, 3886
acsc, 3886
asec, 3886
asin, 3886
atan, 3886
cos, 3886
cot, 3886
csc, 3886
exp, 3886
sec, 3886
sin, 3886
tan, 3886
SturmHabicht
SHP, 3897
SturmHabichtCoefficients
SHP, 3897
SturmHabichtMultiple
SHP, 3897
SturmHabichtPackage, 3897
SturmHabichtSequence
SHP, 3897
sturmSequence
POLUTIL, 3684
sturmVariationsOf
POLUTIL, 3684
style
DROPT0, 421
subCase?
QCMPACK, 3567
SFQCMPK, 3817
subMatrix
MAMA, 1703
subPolSet?
QCMPACK, 3567
SFQCMPK, 3817
subQuasiComponent?
QCMPACK, 3567
SFQCMPK, 3817
SUBRESP, 3906
primitivePart, 3906
subresultantVector, 3906
subResultantGcd
PRS, 3527
subResultantGcdEuclidean
PRS, 3527
INDEX

SubResultantPackage, 3906
subresultantSequence
   SHP, 3897
subresultantVector
   SUBRESP, 3906
subs1stVar
   PFORP, 3232
subs2ndVar
   PFORP, 3232
subscriptedVariables
   D03AGNT, 512
subSet
   SGCF, 3927
subsInVar
   PFORP, 3232
subTriSet?
   QCMPACK, 3567
   SFQCMPK, 3817
suchThat
   PATTERN1, 3289
   PMPRED, 119
   PMPREDFS, 891
sum
   ISUMP, 1272
   SUMFS, 910
   SUMRF, 3652
SUMFS, 910
sum, 910
summation
   COMBF, 208
sumOfDivisors
   INTHEORY, 1320
sumOfKthPowerDivisors
   INTHEORY, 1320
sumOfSquares
   E04AGNT, 721
SUMRF, 3652
sum, 3652
sumSquares
   GAUSSFAC, 979
SUP2, 3811
map, 3811
supDimElseRittWu?
   QCMPACK, 3567
   SFQCMPK, 3817
SUPFRACF, 3910
factor, 3910
squareFree, 3910
SupFractionFactorizer, 3910
swap
   COMMUPC, 229
sylvesterMatrix
   BEZOUT, 165
sylvesterSequence
   POLUTIL, 3684
SYMFUNC, 3938
symFunc, 3938
symFunc
   SYMFUNC, 3938
SymmetricFunctions, 3938
symmetricGroup
   PGE, 3346
SymmetricGroupCombinatoricFunctions, 3927
symmetricPower
   LODOOPS, 1530
symmetricProduct
   LODOOPS, 1530
symmetricTensors
   REP1, 3750
SYSSOLP, 3920
solve, 3920
triangularSystems, 3920
systemCommand
   MSYSCMD, 1737
SystemODESolver, 3913
systemSizeIF
   D02AGNT, 490
SystemSolvePackage, 3920
t
   RFDIST, 3613
tab
   TABLBUMP, 3942
tab1
   TABLBUMP, 3942
TABLBUMP, 3942
bat, 3942
bat1, 3942
bumptab, 3942
bumptab1, 3942
inverse, 3942
lex, 3942
maxrow, 3942
mr, 3942
slex, 3942
tab, 3942
tab1, 3942
untab, 3942
TableauxBumpers, 3942
tablePow
GENEEZ, 1027
TabulatedComputationPackage, 3947
tan
EF, 535
STTF, 3876
STTFNC, 3886
tan2cot
TRMANIP, 4012
tan2trig
TRMANIP, 4012
tanAn
TANEXP, 3951
TANEXP, 3951
tanAn, 3951
tanNa, 3951
tanSum, 3951
TangentExpansions, 3951
tanh
EF, 535
tanh2coth
TRMANIP, 4012
tanh2trigh
TRMANIP, 4012
tanintegrate
INTTR, 4001
tanNa
TANEXP, 3951
tanQ
EFSTRUC, 579
tanSum
TANEXP, 3951
taylor
EXPR2UPS, 704
GENUPS, 1021
TaylorSolve, 3954
TBCMPPK, 3947
clearTable, 3947
extractIfCan, 3947
initTable, 3947
insert, 3947
makingStats?, 3947
printInfo, 3947
printingInfo?, 3947
printStats, 3947
startStats, 3947
usingTable?, 3947
TemplateUtilities, 3958
TEMUTL, 3958
interpretString, 3958
stripCommentsAndBlanks, 3958
tensorMap
STNSR, 3873
tensorProduct
REP1, 3750
testDim
GROEBSOL, 1118
testModulus
GENEEZ, 1027
TEX1, 3960
coerce, 3960
TexFormat1, 3960
theCurve
GPAFF, 992
PAFF, 3215
PAFFFF, 3223
thetaCoord
PTPACK, 3370
times
OREPCTO, 4122
title
DROPT0, 421
tomodpa
POLYVEC, 4144
ToolsForSign, 3962
TOOLSIGN, 3962
direction, 3963
nonQsign, 3963
sign, 3963
topFortranOutputStack
FOP, 842
TopLevelDrawFunctions, 3965
TopLevelDrawFunctionsForAlgebraicCurves, 3974
INDEX

TopLevelDrawFunctionsForCompiledFunctions, TranscendentalRischDE, 4027
TopLevelDrawFunctionsForPoints, 3993
TopLevelThreeSpace, 3996
TOPSP, 3996
createThreeSpace, 3996
toroidal

COORDSYS, 268
torsion?
PFO, 3373
PFOQ, 3380
torsionIfCan
PFO, 3373
PFOQ, 3380
toScale
DROPT0, 421
toseInvertible?
RSETGCD, 3736
toseInvertibleSet
RSETGCD, 3736
toseLastSubResultant
RSETGCD, 3736
toseSquareFreePart
RSETGCD, 3736
totalDegree
PFORP, 3232
totalfract
MPRFF, 1742
totalGroebner
PGROEB, 3393
totalLex
ORDFUNS, 3204
totolex
LGROBP, 1549
trace
INBFF, 1249
trace2PowMod
DDFACT, 349
tracePowMod
DDFACT, 349
transcendentalDecompose
RSDCMPK, 3728
SRDCMPK, 3827
TranscendentalHermiteIntegration, 3998
TranscendentalIntegration, 4001
TranscendentalManipulations, 4012
TranscendentalRischDESystem, 4032
transform
LGROBP, 1549
translate
PFORP, 3232
translateToOrigin
PAFFFF, 3223
PLPKCRV, 3386
TransSolvePackage, 4043
TransSolvePackageService, 4055
trapezoidal
NUMQUAD, 3154
trapezoidal
NUMQUAD, 3154
triangSolve
ZDSOLVE, 4253
TriangularMatrixOperations, 4059
triangularSystems
SYSSOLP, 3920
triangulate
ODESYS, 3913
TRIGMNIP, 4062
complexElementary, 4062
complexForm, 4062
complexNormalize, 4062
imag, 4062
real, 4062
real?, 4062
trigs, 4062
TrigonometricManipulations, 4062
trigs
CTRIGMNP, 261
TRIGMNIP, 4062
trigs2explogs
ITRIGMNP, 1275
TRIMAT, 4059
LowTriBddDenomInv, 4059
UpTriBddDenomInv, 4059
TRMANIP, 4012
cos2sec, 4012
cosh2sech, 4012
cot2tan, 4012
cot2trig, 4012
coth2tanh, 4012
coth2trigh, 4012
csc2sin, 4012
csch2sinh, 4012
expand, 4012
expandLog, 4012
expandPower, 4012
expandTrigProducts, 4012
htrigs, 4012
removeCoshSq, 4012
removeCosSq, 4012
removeSinhSq, 4012
removeSinSq, 4012
sec2cos, 4012
sech2cosh, 4012
simplify, 4012
simplifyExp, 4012
simplifyLog, 4012
sin2csc, 4012
sinh2csch, 4012
tan2cot, 4012
tan2trig, 4012
tanh2coth, 4012
tanh2trigh, 4012
twist
MAPPKG3, 1636
TwoDimensionalPlotClipping, 4071
TWOFACT, 4078
generalSqFr, 4078
genertwoFactor, 4078
twoFactor, 4078
twoFactor
TWOFACT, 4078
TwoFactorize, 4078
U32VectorPolynomialOperations, 4144
UDPO, 4135
geterOrder, 4135
largest, 4135
less?, 4135
more?, 4135
setOrer, 4135
userOrdered?, 4135
UDVO, 4138
getVerableOrder, 4138
resetVerableOrder, 4138
setVerableOrder, 4138
UFPS1, 4094
hadamard, 4094
ULS2, 4096
map, 4096
unaryFunction
MKUCFUNC, 1597
uncouplingMatrices
ASSOCEQ, 115
UNIFACT, 4086
factor, 4086
factorSquareFree, 4086
henselFact, 4086
uniform
RDIST, 3610
RFDIST, 3613
RIDIST, 3616
truncatedmuladd
POLYVEC, 4144
truncatedmultiplication
POLYVEC, 4144
tryFunctionalDecomposition
GALFACT, 954
tryFunctionalDecomposition?
GALFACT, 954
tube
NUMTUBE, 3175
tubePlot
EXPRTUBE, 712
TubePlotTools, 4067
tubePoints
DROPT0, 421
tubePointsDefault
VIEWDEF, 4161
tubeRadius
DROPT0, 421
tubeRadiusDefault
VIEWDEF, 4161
TUBETOOL, 4067
?*, 4067
?+, 4067
?.?, 4067
cosSinInfo, 4067
cross, 4067
dot, 4067
loopPoints, 4067
point, 4067
unitVector, 4067
INDEX

uniform01
  RFDIST, 3613
union
  INTTOOLS, 1367
UNISEG2, 4132
  map, 4133
units
  DROPT0, 421
unitsColorDefault
  VIEWDEF, 4161
unitVector
  TUBETOOL, 4067
univariate
  INTG0, 1036
  PFORP, 3232
  POLY2UP, 3482
  POLYCATQ, 3401
  RF, 3625
univariate?
  PSETPK, 3453
UnivariateFactorize, 4086
UnivariateFormalPowerSeriesFunctions, 4094
UnivariateLaurentSeriesFunctions2, 4098
UnivariatePolynomialCategoryFunctions2, 4099
UnivariatePolynomialCommonDenominator, 4101
UnivariatePolynomialDecompositionPackage, 4103
UnivariatePolynomialDivisionPackage, 4107
UnivariatePolynomialFunctions2, 4110
UnivariatePolynomialMultiplicationPackage, 4112
univariatePolynomials
  PSETPK, 3453
univariatePolynomialsGcds
  PSETPK, 3453
UnivariatePolynomialSquareFree, 4115
UnivariatePuiseuxSeriesFunctions2, 4119
UnivariateSkewPolynomialCategoryOps, 4122
univariateSolve
  ZDSOLVE, 4253
UnivariateTaylorSeriesFunctions2, 4126
UnivariateTaylorSeriesODESolver, 4128
univcase
  MULTSQFR, 1782
UniversalSegmentFunctions2, 4132
unprotectedRemoveRedundantFactors
  PSETPK, 3453
unrankImproperPartitions0
  SGCF, 3927
unrankImproperPartitions1
  SGCF, 3927
untab
  TABLBUMP, 3942
unvectorise
  GALPOLYU, 971
  SOLVESER, 4055
UP2, 4110
  map, 4110
UP2ifCan
  FSUPFACT, 942
UPCDEN, 4101
clearDenominator, 4101
commonDenominator, 4101
splitDenominator, 4101
updatD
  GBINTERN, 1077
upDateBranches
UnivariatePolynomialCategoryFunctions2, 4098
GBINTERN, 1077
UnivariatePolynomialCommonDenominator, 4103
UnivariatePolynomialDecompositionPackage, 4103
  leftFactorIfCan, 4103
  monicCompleteDecompose, 4103
UnivariatePolynomialDivisionPackage, 4107
  monicDecomposeIfCan, 4103
  monicRightFactorIfCan, 4103
UnivariatePolynomialFunctions2, 4110
  rightFactorIfCan, 4103
UPDIVP, 4107
divideIfCan, 4107
UnivariatePolynomialMultiplicationPackage, 4112
  UPMP, 4112
  karatsuba, 4112
  karatsubaOnce, 4112
  noKaratsuba, 4112
UPSQFREE, 4115
  BumInSepFFE, 4115
  squareFree, 4115
  squareFreePart, 4115
univariateSolve
  UpTriBddDenomInv
  TRIMAT, 4059
UPXS2, 4119
  map, 4119
useEisensteinCriterion
GALFACT, 954
useEisensteinCriterion?
GALFACT, 954
UserDefinedPartialOrdering, 4135
UserDefinedVariableOrdering, 4138
userOrdered?
UDPO, 4135
useSingleFactorBound
GALFACT, 954
useSingleFactorBound?
GALFACT, 954
usingTable?
TBCMPPK, 3947
UTS2, 4126
map, 4126
UTS2UP
UTSODETL, 4141
UTSODE, 4128
fixedPointExquo, 4128
mpsode, 4128
ode, 4128
ode1, 4128
ode2, 4128
stFunc1, 4128
stFunc2, 4128
stFuncN, 4128
UTSODETL, 4141
LODO2FUN, 4141
RF2UTS, 4141
UTS2UP, 4141
UTSodetools, 4141
UTSSOL, 3954
seriesSolve, 3954
validExponential
EFSTRUC, 579
var1Steps
DROPT0, 421
var1StepsDefault
VIEWDEF, 4161
var2Steps
DROPT0, 421
var2StepsDefault
VIEWDEF, 4161
variables
E04AGNT, 721
FACUTIL, 738
POLYCATQ, 3401
RF, 3625
variationOfParameters
ODETOOLS, 3186
vark
INTTOOLS, 1367
varList
D03AGNT, 512
E04AGNT, 721
varselect
INTTOOLS, 1367
VECTOR2, 4158
map, 4158
reduce, 4158
scan, 4158
vectoraddmul
POLYVEC, 4144
vectorcombination
POLYVEC, 4144
VectorFunctions2, 4158
vedf2vef
ESTOOLS, 667
vertConcat
MAMA, 1703
vertSplit
MAMA, 1703
VIEW, 4168
coerce, 4168
drawCurves, 4168
graphCurves, 4168
VIEWDEF, 4161
axesColorDefault, 4161
lineColorDefault, 4161
pointColorDefault, 4161
pointSizeDefault, 4161
tubePointsDefault, 4161
tubeRadiusDefault, 4161
unitsColorDefault, 4161
var1StepsDefault, 4161
var2StepsDefault, 4161
viewDefaults, 4161
viewPosDefault, 4161
viewSizeDefault, 4161
viewWriteAvailable, 4161
FCPAK1, 838
zeroVector
FCPAK1, 838
ZetaFunction
GPAFF, 992
PAFF, 3215
PAFFFF, 3223
ZLINDEP, 1304
  linearDependenceOverZ, 1304
  linearlyDependentOverZ?, 1304
  solveLinearlyOverQ, 1304