

REDUCE

User's Manual

Free Version

Snapshot of September 30, 2014

Anthony C. Hearn

[**http://reduce-algebra.sourceforge.net/**](http://reduce-algebra.sourceforge.net/)

September 30, 2014

Copyright ©2004–2014 Anthony C. Hearn. All rights reserved.

Registered system holders may reproduce all or any part of this publication for internal purposes, provided that the source of the material is clearly acknowledged, and the copyright notice is retained.

Contents

Abstract	21
1 Introductory Information	25
2 Structure of Programs	29
2.1 The REDUCE Standard Character Set	29
2.2 Numbers	30
2.3 Identifiers	31
2.4 Variables	32
2.5 Strings	33
2.6 Comments	33
2.7 Operators	34
3 Expressions	37
3.1 Scalar Expressions	37
3.2 Integer Expressions	38
3.3 Boolean Expressions	39
3.4 Equations	40
3.5 Proper Statements as Expressions	41
4 Lists	43
4.1 Operations on Lists	43
4.1.1 LIST	44
4.1.2 FIRST	44

4.1.3	SECOND	44
4.1.4	THIRD	44
4.1.5	REST	44
4.1.6	. (Cons) Operator	44
4.1.7	APPEND	44
4.1.8	REVERSE	45
4.1.9	List Arguments of Other Operators	45
4.1.10	Caveats and Examples	45
5	Statements	47
5.1	Assignment Statements	48
5.1.1	Set Statement	48
5.2	Group Statements	49
5.3	Conditional Statements	49
5.4	FOR Statements	50
5.5	WHILE ... DO	52
5.6	REPEAT ... UNTIL	53
5.7	Compound Statements	53
5.7.1	Compound Statements with GO TO	55
5.7.2	Labels and GO TO Statements	55
5.7.3	RETURN Statements	56
6	Commands and Declarations	59
6.1	Array Declarations	59
6.2	Mode Handling Declarations	60
6.3	END	61
6.4	BYE Command	61
6.5	SHOWTIME Command	61
6.6	DEFINE Command	61
7	Built-in Prefix Operators	63
7.1	Numerical Operators	63

CONTENTS	3
7.1.1 ABS	63
7.1.2 CEILING	64
7.1.3 CONJ	64
7.1.4 FACTORIAL	64
7.1.5 FIX	64
7.1.6 FLOOR	65
7.1.7 IMPART	65
7.1.8 MAX/MIN	65
7.1.9 NEXTPRIME	65
7.1.10 RANDOM	66
7.1.11 RANDOM_NEW_SEED	66
7.1.12 REPART	66
7.1.13 ROUND	67
7.1.14 SIGN	67
7.2 Mathematical Functions	67
7.3 Bernoulli Numbers and Euler Numbers	71
7.4 Fibonacci Numbers and Fibonacci Polynomials	71
7.5 Motzkin numbers	72
7.6 CONTINUED_FRACTION Operator	72
7.7 DF Operator	73
7.7.1 Switches influencing differentiation	73
7.7.2 Adding Differentiation Rules	74
7.8 INT Operator	75
7.8.1 Options	76
7.8.2 Advanced Use	76
7.8.3 References	77
7.9 LENGTH Operator	77
7.10 MAP Operator	77
7.11 MKID Operator	79
7.12 The Pochhammer Notation	79

7.13	PF Operator	79
7.14	SELECT Operator	80
7.15	SOLVE Operator	82
7.15.1	Handling of Undetermined Solutions	83
7.15.2	Solutions of Equations Involving Cubics and Quartics	84
7.15.3	Other Options	86
7.15.4	Parameters and Variable Dependency	87
7.16	Even and Odd Operators	89
7.17	Linear Operators	90
7.18	Non-Commuting Operators	91
7.19	Symmetric and Antisymmetric Operators	91
7.20	Declaring New Prefix Operators	92
7.21	Declaring New Infix Operators	93
7.22	Creating/Removing Variable Dependency	93
8	Display and Structuring of Expressions	95
8.1	Kernels	95
8.2	The Expression Workspace	96
8.3	Output of Expressions	97
8.3.1	LINELENGTH Operator	98
8.3.2	Output Declarations	98
8.3.3	Output Control Switches	99
8.3.4	WRITE Command	102
8.3.5	Suppression of Zeros	104
8.3.6	FORTRAN Style Output Of Expressions	104
8.3.7	Saving Expressions for Later Use as Input	107
8.3.8	Displaying Expression Structure	108
8.4	Changing the Internal Order of Variables	109
8.5	Obtaining Parts of Algebraic Expressions	110
8.5.1	COEFF Operator	110
8.5.2	COEFFN Operator	111

8.5.3	PART Operator	111
8.5.4	Substituting for Parts of Expressions	112
9	Polynomials and Rationals	113
9.1	Controlling the Expansion of Expressions	114
9.2	Factorization of Polynomials	114
9.3	Cancellation of Common Factors	116
9.3.1	Determining the GCD of Two Polynomials	117
9.4	Working with Least Common Multiples	118
9.5	Controlling Use of Common Denominators	118
9.6	REMAINDER Operator	118
9.7	RESULTANT Operator	119
9.8	DECOMPOSE Operator	120
9.9	INTERPOL operator	121
9.10	Obtaining Parts of Polynomials and Rationals	121
9.10.1	DEG Operator	121
9.10.2	DEN Operator	122
9.10.3	LCOF Operator	122
9.10.4	LPOWER Operator	123
9.10.5	LTERM Operator	123
9.10.6	MAINVAR Operator	123
9.10.7	NUM Operator	124
9.10.8	REDUCT Operator	124
9.10.9	TOTALDEG Operator	125
9.11	Polynomial Coefficient Arithmetic	125
9.11.1	Rational Coefficients in Polynomials	125
9.11.2	Real Coefficients in Polynomials	126
9.11.3	Modular Number Coefficients in Polynomials	127
9.11.4	Complex Number Coefficients in Polynomials	128
9.12	ROOT_VAL Operator	128

10 Assigning and Testing Algebraic Properties	131
10.1 REALVALUED Declaration and Check	131
10.2 Declaring Expressions Positive or Negative	132
11 Substitution Commands	135
11.1 SUB Operator	135
11.2 LET Rules	136
11.2.1 FOR ALL ... LET	138
11.2.2 FOR ALL ... SUCH THAT ... LET	139
11.2.3 Removing Assignments and Substitution Rules	140
11.2.4 Overlapping LET Rules	140
11.2.5 Substitutions for General Expressions	141
11.3 Rule Lists	143
11.4 Asymptotic Commands	149
12 File Handling Commands	151
12.1 IN Command	151
12.2 OUT Command	152
12.3 SHUT Command	152
13 Commands for Interactive Use	155
13.1 Referencing Previous Results	155
13.2 Interactive Editing	156
13.3 Interactive File Control	157
14 Matrix Calculations	159
14.1 MAT Operator	159
14.2 Matrix Variables	159
14.3 Matrix Expressions	160
14.4 Operators with Matrix Arguments	161
14.4.1 DET Operator	161
14.4.2 MATEIGEN Operator	162

14.4.3	TP Operator	163
14.4.4	Trace Operator	163
14.4.5	Matrix Cofactors	163
14.4.6	NULLSPACE Operator	163
14.4.7	RANK Operator	164
14.5	Matrix Assignments	165
14.6	Evaluating Matrix Elements	165
15	Procedures	167
15.1	Procedure Heading	168
15.2	Procedure Body	169
15.3	Using LET Inside Procedures	170
15.4	LET Rules as Procedures	171
15.5	REMEMBER Statement	173
16	User Contributed Packages	175
16.1	ALGINT: Integration of square roots	176
16.2	APPLYSYM: Infinitesimal symmetries of differential equations	177
16.2.1	Introduction and overview of the symmetry method	177
16.2.2	Applying symmetries with APPLYSYM	183
16.2.3	Solving quasilinear PDEs	192
16.2.4	Transformation of DEs	196
	Bibliography	198
16.3	ARNUM: An algebraic number package	201
16.4	ASSERT: Dynamic Verification of Assertions on Function Types	207
16.5	ASSIST: Useful utilities for various applications	208
16.6	AVECTOR: A vector algebra and calculus package	208
16.6.1	Introduction	208
16.6.2	Vector declaration and initialisation	208
16.6.3	Vector algebra	209
16.6.4	Vector calculus	210

16.6.5	Volume and Line Integration	212
16.6.6	Defining new functions and procedures	214
16.6.7	Acknowledgements	214
16.7	BIBASIS: A Package for Calculating Boolean Involutive Bases . .	215
16.7.1	Introduction	215
16.7.2	Boolean Ring	215
16.7.3	Pommaret Involutive Algorithm	216
16.7.4	BIBASIS Package	217
16.7.5	Examples	218
	Bibliography	221
16.8	BOOLEAN: A package for boolean algebra	222
16.8.1	Introduction	222
16.8.2	Entering boolean expressions	222
16.8.3	Normal forms	223
16.8.4	Evaluation of a boolean expression	224
16.9	CALI: A package for computational commutative algebra	226
16.10	CAMAL: Calculations in celestial mechanics	227
16.10.1	Introduction	227
16.10.2	How CAMAL Worked	228
16.10.3	Towards a CAMAL Module	231
16.10.4	Integration with REDUCE	233
16.10.5	The Simple Experiments	234
16.10.6	A Medium-Sized Problem	235
16.10.7	Conclusion	237
16.11	CDIFF: A package for the geometry of Differential Equations . .	240
16.12	CHANGEVR: Change of Independent Variable(s) in DEs	241
16.12.1	Introduction	241
16.12.2	How to Use CHANGEVR	242
16.12.3	AN EXAMPLE. The 2-dim. Laplace Equation	243
16.12.4	ANOTHER EXAMPLE. An Euler Equation	244

16.13COMPACT: Package for compacting expressions	245
16.14CRACK: Solving overdetermined systems of PDEs or ODEs	246
16.15CVIT: Fast calculation of Dirac gamma matrix traces	247
16.16DEFINT: A definite integration interface	256
16.16.1 Introduction	256
16.16.2 Integration between zero and infinity	256
16.16.3 Integration over other ranges	257
16.16.4 Using the definite integration package	258
16.16.5 Integral Transforms	260
16.16.6 Additional Meijer G-function Definitions	262
16.16.7 The print_conditions function	263
16.16.8 Acknowledgements	264
Bibliography	264
16.17DESIR: Differential linear homogeneous equation solutions in the neighborhood of irregular and regular singular points	266
16.17.1 INTRODUCTION	266
16.17.2 FORMS OF SOLUTIONS	267
16.17.3 INTERACTIVE USE	268
16.17.4 DIRECT USE	269
16.17.5 USEFUL FUNCTIONS	269
16.17.6 LIMITATIONS	273
16.18DFPART: Derivatives of generic functions	274
16.18.1 Generic Functions	274
16.18.2 Partial Derivatives	275
16.18.3 Substitutions	277
16.19DUMMY: Canonical form of expressions with dummy variables .	279
16.19.1 Introduction	279
16.19.2 Dummy variables and dummy summations	280
16.19.3 The Operators and their Properties	282
16.19.4 The Function CANONICAL	283
16.19.5 Bibliography	284

16.20EXCALC: A differential geometry package	286
16.20.1 Introduction	286
16.20.2 Declarations	287
16.20.3 Exterior Multiplication	288
16.20.4 Partial Differentiation	289
16.20.5 Exterior Differentiation	290
16.20.6 Inner Product	292
16.20.7 Lie Derivative	293
16.20.8 Hodge-* Duality Operator	293
16.20.9 Variational Derivative	295
16.20.10 Handling of Indices	296
16.20.1 Metric Structures	299
16.20.1 Riemannian Connections	303
16.20.1 Ordering and Structuring	304
16.20.1 Summary of Operators and Commands	305
16.20.1 Examples	307
16.21FIDE: Finite difference method for partial differential equations	318
16.22FPS: Automatic calculation of formal power series	319
16.22.1 Introduction	319
16.22.2 REDUCE operator FPS	319
16.22.3 REDUCE operator SimpleDE	321
16.22.4 Problems in the current version	321
Bibliography	321
16.23GCREF: A Graph Cross Referencer	323
16.23.1 Basic Usage	323
16.23.2 Shell Script "gcref"	323
16.23.3 Redering with yED	323
16.24GENTRAN: A code generation package	325
16.24.1 INTRODUCTION	326
16.24.2 Interactive Code Generation	328

16.24.3 Template Processing	352
16.24.4 Output Redirection	364
16.24.5 Modification of the Code Generation Process	375
16.24.6 Examples	381
16.24.7 Symbolic Mode Functions	393
16.24.8 Translatable REDUCE Expressions & Statements	409
16.24.9 List of Commands, Switches, & Variables	424
16.24.10 The Programs M1 . F and M2 . F.	426
16.25 GNUPLOT: Display of functions and surfaces	442
16.25.1 Introduction	442
16.25.2 Command PLOT	442
16.25.3 Paper output	447
16.25.4 Mesh generation for implicit curves	447
16.25.5 Mesh generation for surfaces	448
16.25.6 GNUPLOT operation	448
16.25.7 Saving GNUPLOT command sequence	448
16.25.8 Direct Call of GNUPLOT	449
16.25.9 Examples	449
16.26 GROEBNER: A Gröbner basis package	455
16.27 GUARDIAN: Guarded Expressions in Practice	455
16.27.1 Introduction	456
16.27.2 An outline of our method	457
16.27.3 Examples	466
16.27.4 Outlook	468
16.27.5 Conclusions	471
Bibliography	471
16.28 IDEALS: Arithmetic for polynomial ideals	473
16.28.1 Introduction	473
16.28.2 Initialization	473
16.28.3 Bases	473

16.28.4 Algorithms	474
16.28.5 Examples	475
16.29INEQ: Support for solving inequalities	476
16.30INVBASE: A package for computing involutive bases	478
16.30.1 Introduction	478
16.30.2 The Basic Operators	479
Bibliography	482
16.31LAPLACE: Laplace transforms	483
16.32LIE: Functions for the classification of real n-dimensional Lie al- gebras	485
16.33LIMITS: A package for finding limits	489
16.33.1 Normal entry points	489
16.33.2 Direction-dependent limits	489
16.33.3 Diagnostic Functions	489
16.34LINALG: Linear algebra package	491
16.34.1 Introduction	491
16.34.2 Getting started	493
16.34.3 What's available	493
16.34.4 Fast Linear Algebra	519
16.34.5 Acknowledgments	520
Bibliography	520
16.35LPDO: Linear Partial Differential Operators	521
16.35.1 Introduction	521
16.35.2 Operators	522
16.35.3 Shapes of F-elements	523
16.35.4 Commands	524
16.36MODSR: Modular solve and roots	531
16.37NCPOLY: Non-commutative polynomial ideals	532
16.37.1 Introduction	532
16.37.2 Setup, Cleanup	532
16.37.3 Left and right ideals	534

16.37.4 Gröbner bases	534
16.37.5 Left or right polynomial division	535
16.37.6 Left or right polynomial reduction	535
16.37.7 Factorization	536
16.37.8 Output of expressions	537
16.38 NORMFORM: Computation of matrix normal forms	539
16.38.1 Introduction	539
16.38.2 smithex	540
16.38.3 smithex_int	541
16.38.4 frobenius	542
16.38.5 ratjordan	543
16.38.6 jordansymbolic	545
16.38.7 jordan	547
16.38.8 arnum	550
16.38.9 modular	551
Bibliography	552
16.39 NUMERIC: Solving numerical problems	553
16.39.1 Syntax	553
16.39.2 Minima	554
16.39.3 Roots of Functions/ Solutions of Equations	555
16.39.4 Integrals	556
16.39.5 Ordinary Differential Equations	557
16.39.6 Bounds of a Function	559
16.39.7 Chebyshev Curve Fitting	560
16.39.8 General Curve Fitting	561
16.39.9 Function Bases	562
16.40 ODESOLVE: Ordinary differential equations solver	564
16.40.1 Introduction	564
16.40.2 Installation	565
16.40.3 User interface	566

16.40.4 Output syntax	572
16.40.5 Solution techniques	572
16.40.6 Extension interface	577
16.40.7 Change log	580
16.40.8 Planned developments	580
Bibliography	581
16.41 ORTHOVEC: Manipulation of scalars and vectors	583
16.41.1 Introduction	583
16.41.2 Initialisation	584
16.41.3 Input-Output	585
16.41.4 Algebraic Operations	585
16.41.5 Differential Operations	588
16.41.6 Integral Operations	590
16.41.7 Test Cases	591
16.42 PHYSOP: Operator calculus in quantum theory	594
16.42.1 Introduction	594
16.42.2 The NONCOM2 Package	594
16.42.3 The PHYSOP package	595
16.42.4 Known problems in the current release of PHYSOP	603
16.42.5 Compilation of the packages	604
16.42.6 Final remarks	605
16.42.7 Appendix: List of error and warning messages	605
16.43 PM: A REDUCE pattern matcher	607
16.44 RANDPOLY: A random polynomial generator	607
16.44.1 Introduction	607
16.44.2 Basic use of <code>randpoly</code>	608
16.44.3 Advanced use of <code>randpoly</code>	609
16.44.4 Subsidiary functions: <code>rand</code> , <code>proc</code> , <code>random</code>	610
16.44.5 Examples	612
16.44.6 Appendix: Algorithmic background	614

16.45REACTEQN: Support for chemical reaction equation systems . . .	617
16.46REDLOG: Extend REDUCE to a computer logic system	621
16.47RESET: Code to reset REDUCE to its initial state	621
16.48RESIDUE: A residue package	622
16.49RLFI: REDUCE L ^A T _E X formula interface	626
16.49.1 APPENDIX: Summary and syntax	628
16.50ROOTS: A REDUCE root finding package	631
16.50.1 Introduction	631
16.50.2 Root Finding Strategies	631
16.50.3 Top Level Functions	632
16.50.4 Switches Used in Input	634
16.50.5 Internal and Output Use of Switches	635
16.50.6 Root Package Switches	636
16.50.7 Operational Parameters and Parameter Setting.	636
16.50.8 Avoiding truncation of polynomials on input	637
16.51RSOLVE: Rational/integer polynomial solvers	638
16.51.1 Introduction	638
16.51.2 The user interface	638
16.51.3 Examples	639
16.51.4 Tracing	639
16.52SCOPE: REDUCE source code optimization package	640
16.53SETS: A basic set theory package	641
16.53.1 Introduction	641
16.53.2 Infix operator precedence	642
16.53.3 Explicit set representation and <code>mkset</code>	642
16.53.4 Union and intersection	643
16.53.5 Symbolic set expressions	643
16.53.6 Set difference	644
16.53.7 Predicates on sets	645
16.53.8 Installation	648

16.53.9 Possible future developments	648
16.54SPDE: Finding symmetry groups of PDE's	650
16.54.1 Description of the System Functions and Variables	650
16.54.2 How to Use the Package	652
16.54.3 Test File	659
16.55SPECFN: Package for special functions	661
16.56SPECFN2: Package for special special functions	662
16.56.1 REDUCE operator HYPERGEOMETRIC	663
16.56.2 Extending the HYPERGEOMETRIC operator	663
16.56.3 REDUCE operator meijerg	664
16.57SUM: A package for series summation	665
16.58SYMMETRY: Operations on symmetric matrices	667
16.58.1 Introduction	667
16.58.2 Operators for linear representations	667
16.58.3 Display Operators	669
16.58.4 Storing a new group	669
Bibliography	671
16.59TAYLOR: Manipulation of Taylor series	672
16.59.1 Basic Use	672
16.59.2 Caveats	676
16.59.3 Warning messages	676
16.59.4 Error messages	677
16.59.5 Comparison to other packages	679
16.60TPS: A truncated power series package	680
16.60.1 Introduction	680
16.60.2 PS Operator	680
16.60.3 PSEXPLIM Operator	682
16.60.4 PSORDLIM Operator	682
16.60.5 PSTERM Operator	683
16.60.6 PSORDER Operator	683

16.60.7 PSSETORDER Operator	683
16.60.8 PSDEPVAR Operator	683
16.60.9 PSEXPANSIONPT operator	684
16.60.10 PSFUNCTION Operator	684
16.60.11 PSCHANGEVAR Operator	684
16.60.12 PSREVERSE Operator	684
16.60.13 PSCOMPOSE Operator	685
16.60.14 PSSUM Operator	686
16.60.15 PSCOPY Operator	687
16.60.16 PSTRUNCATE Operator	687
16.60.17 Arithmetic Operations	688
16.60.18 Differentiation	688
16.60.19 Restrictions and Known Bugs	688
16.61 TRI: TeX REDUCE interface	690
16.62 TRIGSIMP: Simplification and factorization of trigonometric and hyperbolic functions	691
16.62.1 Introduction	691
16.62.2 Simplifying trigonometric expressions	691
16.62.3 Factorizing trigonometric expressions	695
16.62.4 GCDs of trigonometric expressions	696
16.62.5 Further Examples	696
Bibliography	700
16.63 TURTLE: Turtle Graphics Interface for REDUCE	701
16.63.1 Turtle Graphics	701
16.63.2 Implementation	701
16.63.3 Turtle Functions	702
16.63.4 Examples	707
16.63.5 References	713
16.64 WU: Wu algorithm for polynomial systems	715
16.65 XCOLOR: Color factor in some field theories	717
16.66 XIDEAL: Gröbner Bases for exterior algebra	719

16.66.1 Description	719
16.66.2 Declarations	720
16.66.3 Operators	721
16.66.4 Switches	723
16.66.5 Examples	723
Bibliography	726
16.67 ZEILBERG: Indefinite and definite summation	727
16.67.1 Introduction	727
16.67.2 Gosper Algorithm	727
16.67.3 Zeilberger Algorithm	728
16.67.4 REDUCE operator GOSPER	729
16.67.5 REDUCE operator EXTENDED_GOSPER	732
16.67.6 REDUCE operator SUMRECURSION	732
16.67.7 REDUCE operator EXTENDED_SUMRECURSION	735
16.67.8 REDUCE operator HYPERRECURSION	736
16.67.9 REDUCE operator HYPERSUM	738
16.67.10 REDUCE operator SUMTOHYPER	740
16.67.11 Simplification Operators	740
16.67.12 Tracing	742
16.67.13 Global Variables and Switches	744
16.67.14 Messages	745
Bibliography	746
16.68 ZTRANS: Z-transform package	748
16.68.1 Z-Transform	748
16.68.2 Inverse Z-Transform	748
16.68.3 Input for the Z-Transform	748
16.68.4 Input for the Inverse Z-Transform	749
16.68.5 Application of the Z-Transform	750
16.68.6 EXAMPLES	750
Bibliography	756

CONTENTS	19
17 Symbolic Mode	757
17.1 Symbolic Infix Operators	759
17.2 Symbolic Expressions	759
17.3 Quoted Expressions	759
17.4 Lambda Expressions	759
17.5 Symbolic Assignment Statements	760
17.6 FOR EACH Statement	761
17.7 Symbolic Procedures	761
17.8 Standard Lisp Equivalent of Reduce Input	762
17.9 Communicating with Algebraic Mode	762
17.9.1 Passing Algebraic Mode Values to Symbolic Mode	763
17.9.2 Passing Symbolic Mode Values to Algebraic Mode	765
17.9.3 Complete Example	766
17.9.4 Defining Procedures for Intermode Communication	766
17.10Rlisp '88	768
17.11References	768
18 Calculations in High Energy Physics	769
18.1 High Energy Physics Operators	769
18.1.1 . (Cons) Operator	769
18.1.2 G Operator for Gamma Matrices	770
18.1.3 EPS Operator	771
18.2 Vector Variables	771
18.3 Additional Expression Types	772
18.3.1 Vector Expressions	772
18.3.2 Dirac Expressions	772
18.4 Trace Calculations	773
18.5 Mass Declarations	773
18.6 Example	774
18.7 Extensions to More Than Four Dimensions	775

19 REDUCE and Rlisp Utilities	777
19.1 The Standard Lisp Compiler	777
19.2 Fast Loading Code Generation Program	778
19.3 The Standard Lisp Cross Reference Program	779
19.3.1 Restrictions	780
19.3.2 Usage	780
19.3.3 Options	780
19.4 Prettyprinting Reduce Expressions	780
19.5 Prettyprinting Standard Lisp S-Expressions	781
20 Maintaining REDUCE	783
A Reserved Identifiers	787
B Changes since Version 3.8	791

Abstract

This document provides the user with a description of the algebraic programming system REDUCE. The capabilities of this system include:

1. expansion and ordering of polynomials and rational functions,
2. substitutions and pattern matching in a wide variety of forms,
3. automatic and user controlled simplification of expressions,
4. calculations with symbolic matrices,
5. arbitrary precision integer and real arithmetic,
6. facilities for defining new functions and extending program syntax,
7. analytic differentiation and integration,
8. factorization of polynomials,
9. facilities for the solution of a variety of algebraic equations,
10. facilities for the output of expressions in a variety of formats,
11. facilities for generating numerical programs from symbolic input,
12. Dirac matrix calculations of interest to high energy physicists.

Acknowledgment

The production of this version of the manual has been the result of the contributions of a large number of individuals who have taken the time and effort to suggest improvements to previous versions, and to draft new sections. Particular thanks are due to Gerry Rayna, who provided a draft rewrite of most of the first half of the manual. Other people who have made significant contributions have included John Fitch, Martin Griss, Stan Kameny, Jed Marti, Herbert Melenk, Don Morrison, Arthur Norman, Eberhard Schröder, Larry Seward and Walter Tietze. Finally, Richard Hitt produced a \TeX version of the REDUCE 3.3 manual, which has been a useful guide for the production of the \LaTeX version of this manual.

Chapter 1

Introductory Information

REDUCE is a system for carrying out algebraic operations accurately, no matter how complicated the expressions become. It can manipulate polynomials in a variety of forms, both expanding and factoring them, and extract various parts of them as required. REDUCE can also do differentiation and integration, but we shall only show trivial examples of this in this introduction. Other topics not considered include the use of arrays, the definition of procedures and operators, the specific routines for high energy physics calculations, the use of files to eliminate repetitious typing and for saving results, and the editing of the input text.

Also not considered in any detail in this introduction are the many options that are available for varying computational procedures, output forms, number systems used, and so on.

REDUCE is designed to be an interactive system, so that the user can input an algebraic expression and see its value before moving on to the next calculation. For those systems that do not support interactive use, or for those calculations, especially long ones, for which a standard script can be defined, REDUCE can also be used in batch mode. In this case, a sequence of commands can be given to REDUCE and results obtained without any user interaction during the computation.

In this introduction, we shall limit ourselves to the interactive use of REDUCE, since this illustrates most completely the capabilities of the system. When REDUCE is called, it begins by printing a banner message like:

```
REDUCE 3.8, 15-Jul-2003 ...
```

where the version number and the system release date will change from time to time. It then prompts the user for input by:

```
1:
```

You can now type a REDUCE statement, terminated by a semicolon to indicate the end of the expression, for example:

```
(x+y+z) ^ 2;
```

This expression would normally be followed by another character (a Return on an ASCII keyboard) to “wake up” the system, which would then input the expression, evaluate it, and return the result:

$$x^2 + 2 * x * y + 2 * x * z + y^2 + 2 * y * z + z^2$$

Let us review this simple example to learn a little more about the way that REDUCE works. First, we note that REDUCE deals with variables, and constants like other computer languages, but that in evaluating the former, a variable can stand for itself. Expression evaluation normally follows the rules of high school algebra, so the only surprise in the above example might be that the expression was expanded. REDUCE normally expands expressions where possible, collecting like terms and ordering the variables in a specific manner. However, expansion, ordering of variables, format of output and so on is under control of the user, and various declarations are available to manipulate these.

Another characteristic of the above example is the use of lower case on input and upper case on output. In fact, input may be in either mode, but output is usually in lower case. To make the difference between input and output more distinct in this manual, all expressions intended for input will be shown in lower case and output in upper case. However, for stylistic reasons, we represent all single identifiers in the text in upper case.

Finally, the numerical prompt can be used to reference the result in a later computation.

As a further illustration of the system features, the user should try:

```
for i:= 1:40 product i;
```

The result in this case is the value of 40!,

```
815915283247897734345611269596115894272000000000
```

You can also get the same result by saying

```
factorial 40;
```

Since we want exact results in algebraic calculations, it is essential that integer arithmetic be performed to arbitrary precision, as in the above example. Further-

more, the `FOR` statement in the above is illustrative of a whole range of combining forms that `REDUCE` supports for the convenience of the user.

Among the many options in `REDUCE` is the use of other number systems, such as multiple precision floating point with any specified number of digits — of use if roundoff in, say, the 100th digit is all that can be tolerated.

In many cases, it is necessary to use the results of one calculation in succeeding calculations. One way to do this is via an assignment for a variable, such as

```
u := (x+y+z) ^ 2;
```

If we now use `U` in later calculations, the value of the right-hand side of the above will be used.

The results of a given calculation are also saved in the variable `WS` (for `WorkSpace`), so this can be used in the next calculation for further processing.

For example, the expression

```
df (ws, x) ;
```

following the previous evaluation will calculate the derivative of $(x+y+z)^2$ with respect to `X`. Alternatively,

```
int (ws, y) ;
```

would calculate the integral of the same expression with respect to `y`.

`REDUCE` is also capable of handling symbolic matrices. For example,

```
matrix m(2,2) ;
```

declares `m` to be a two by two matrix, and

```
m := mat((a,b),(c,d)) ;
```

gives its elements values. Expressions that include `M` and make algebraic sense may now be evaluated, such as $1/m$ to give the inverse, $2*m - u*m^2$ to give us another matrix and `det(m)` to give us the determinant of `M`.

`REDUCE` has a wide range of substitution capabilities. The system knows about elementary functions, but does not automatically invoke many of their well-known properties. For example, products of trigonometrical functions are not converted automatically into multiple angle expressions, but if the user wants this, he can say, for example:

```
(sin(a+b)+cos(a+b))*(sin(a-b)-cos(a-b))
```

where $\cos(\tilde{x}) * \cos(\tilde{y}) = (\cos(x+y) + \cos(x-y)) / 2,$
 $\cos(\tilde{x}) * \sin(\tilde{y}) = (\sin(x+y) - \sin(x-y)) / 2,$
 $\sin(\tilde{x}) * \sin(\tilde{y}) = (\cos(x-y) - \cos(x+y)) / 2;$

where the tilde in front of the variables X and Y indicates that the rules apply for all values of those variables. The result of this calculation is

$$-(\cos(2*A) + \sin(2*B))$$

See also the user-contributed packages ASSIST (chapter 16.5), CAMAL (chapter 16.10) and TRIGSIMP (chapter 16.62).

Another very commonly used capability of the system, and an illustration of one of the many output modes of REDUCE, is the ability to output results in a FORTRAN compatible form. Such results can then be used in a FORTRAN based numerical calculation. This is particularly useful as a way of generating algebraic formulas to be used as the basis of extensive numerical calculations.

For example, the statements

```
on fort;
df(log(x)*(sin(x)+cos(x))/sqrt(x),x,2);
```

will result in the output

```
ANS=(-4.*LOG(X)*COS(X)*X**2-4.*LOG(X)*COS(X)*X+3.*
. LOG(X)*COS(X)-4.*LOG(X)*SIN(X)*X**2+4.*LOG(X)*
. SIN(X)*X+3.*LOG(X)*SIN(X)+8.*COS(X)*X-8.*COS(X)-8.*
. *SIN(X)*X-8.*SIN(X))/(4.*SQRT(X)*X**2)
```

These algebraic manipulations illustrate the algebraic mode of REDUCE. REDUCE is based on Standard Lisp. A symbolic mode is also available for executing Lisp statements. These statements follow the syntax of Lisp, e.g.

```
symbolic car '(a);
```

Communication between the two modes is possible.

With this simple introduction, you are now in a position to study the material in the full REDUCE manual in order to learn just how extensive the range of facilities really is. If further tutorial material is desired, the seven REDUCE Interactive Lessons by David R. Stoutemyer are recommended. These are normally distributed with the system.

Chapter 2

Structure of Programs

A REDUCE program consists of a set of functional commands which are evaluated sequentially by the computer. These commands are built up from declarations, statements and expressions. Such entities are composed of sequences of numbers, variables, operators, strings, reserved words and delimiters (such as commas and parentheses), which in turn are sequences of basic characters.

2.1 The REDUCE Standard Character Set

The basic characters which are used to build REDUCE symbols are the following:

1. The 26 letters a through z
2. The 10 decimal digits 0 through 9
3. The special characters `_ ! " $ % ' () * + , - . / : ; < > = { } <blank>`

With the exception of strings and characters preceded by an exclamation mark, the case of characters is ignored: depending of the underlying LISP they will all be converted internally into lower case or upper case: ALPHA, Alpha and alpha represent the same symbol. Most implementations allow you to switch this conversion off. The operating instructions for a particular implementation should be consulted on this point. For portability, we shall limit ourselves to the standard character set in this exposition.

2.2 Numbers

There are several different types of numbers available in REDUCE. Integers consist of a signed or unsigned sequence of decimal digits written without a decimal point, for example:

-2, 5396, +32

In principle, there is no practical limit on the number of digits permitted as exact arithmetic is used in most implementations. (You should however check the specific instructions for your particular system implementation to make sure that this is true.) For example, if you ask for the value of 2^{2000} you get it displayed as a number of 603 decimal digits, taking up several lines of output on an interactive display. It should be borne in mind of course that computations with such long numbers can be quite slow.

Numbers that aren't integers are usually represented as the quotient of two integers, in lowest terms: that is, as rational numbers.

In essentially all versions of REDUCE it is also possible (but not always desirable!) to ask REDUCE to work with floating point approximations to numbers again, to any precision. Such numbers are called *real*. They can be input in two ways:

1. as a signed or unsigned sequence of any number of decimal digits with an embedded or trailing decimal point.
2. as in 1. followed by a decimal exponent which is written as the letter E followed by a signed or unsigned integer.

e.g. 32. +32.0 0.32E2 and 320.E-1 are all representations of 32.

The declaration `SCIENTIFIC_NOTATION` controls the output format of floating point numbers. At the default settings, any number with five or less digits before the decimal point is printed in a fixed-point notation, e.g., 12345.6. Numbers with more than five digits are printed in scientific notation, e.g., 1.234567E+5. Similarly, by default, any number with eleven or more zeros after the decimal point is printed in scientific notation. To change these defaults, `SCIENTIFIC_NOTATION` can be used in one of two ways. `SCIENTIFIC_NOTATION m`, where m is a positive integer, sets the printing format so that a number with more than m digits before the decimal point, or m or more zeros after the decimal point, is printed in scientific notation. `SCIENTIFIC_NOTATION {m,n}`, with m and n both positive integers, sets the format so that a number with more than m digits before the decimal point, or n or more zeros after the decimal point is printed in scientific notation.

CAUTION: The unsigned part of any number may *not* begin with a decimal point, as this causes confusion with the `CONS (.)` operator, i.e., **NOT ALLOWED:** .5 -.23 +.12; use 0.5 -0.23 +0.12 instead.

2.3 Identifiers

Identifiers in REDUCE consist of one or more alphanumeric characters (i.e. alphabetic letters or decimal digits) the first of which must be alphabetic. The maximum number of characters allowed is implementation dependent, although twenty-four is permitted in most implementations. In addition, the underscore character (`_`) is considered a letter if it is *within* an identifier. For example,

```
a az p1 q23p a_very_long_variable
```

are all identifiers, whereas

```
_a
```

is not.

A sequence of alphanumeric characters in which the first is a digit is interpreted as a product. For example, `2ab3c` is interpreted as `2*ab3c`. There is one exception to this: If the first letter after a digit is `E`, the system will try to interpret that part of the sequence as a real number, which may fail in some cases. For example, `2E12` is the real number 2.0×10^{12} , `2e3c` is `2000.0*C`, and `2ebc` gives an error.

Special characters, such as `-`, `*`, and blank, may be used in identifiers too, even as the first character, but each must be preceded by an exclamation mark in input. For example:

```
light!-years      d!*!*n          good! morning
!$sign            !5goldrings
```

CAUTION: Many system identifiers have such special characters in their names (especially `*` and `=`). If the user accidentally picks the name of one of them for his own purposes it may have catastrophic consequences for his REDUCE run. Users are therefore advised to avoid such names.

Identifiers are used as variables, labels and to name arrays, operators and procedures.

Restrictions

The reserved words listed in another section may not be used as identifiers. No spaces may appear within an identifier, and an identifier may not extend over a line of text. (Hyphenation of an identifier, by using a reserved character as a hyphen before an end-of-line character is possible in some versions of REDUCE).

2.4 Variables

Every variable is named by an identifier, and is given a specific type. The type is of no concern to the ordinary user. Most variables are allowed to have the default type, called *scalar*. These can receive, as values, the representation of any ordinary algebraic expression. In the absence of such a value, they stand for themselves.

Reserved Variables

Several variables in REDUCE have particular properties which should not be changed by the user. These variables include:

CATALAN Catalan's constant, defined as

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)^2}.$$

E Intended to represent the base of the natural logarithms. $\log(e)$, if it occurs in an expression, is automatically replaced by 1. If ROUNDED is on, E is replaced by the value of E to the current degree of floating point precision.

EULER_GAMMA Euler's constant, also available as $-\psi(1)$.

GOLDEN_RATIO The number $\frac{1+\sqrt{5}}{2}$.

I Intended to represent the square root of -1 . i^2 is replaced by -1 , and appropriately for higher powers of I . This applies only to the symbol I used on the top level, not as a formal parameter in a procedure, a local variable, nor in the context `for i:= ...`.

INFINITY Intended to represent ∞ in limit and power series calculations for example, as well as in definite integration. Note however that the current system does *not* do proper arithmetic on ∞ . For example, `infinity + infinity` is `2*infinity`.

KHINCHIN Khinchin's constant, defined as

$$\prod_{n=1}^{\infty} \left(1 + \frac{1}{n(n+2)}\right)^{\log_2 n}.$$

NIL In REDUCE (algebraic mode only) taken as a synonym for zero. Therefore `NIL` cannot be used as a variable.

PI	Intended to represent the circular constant. With <code>ROUNDED</code> on, it is replaced by the value of π to the current degree of floating point precision.
T	Must not be used as a formal parameter or local variable in procedures, since conflict arises with the symbolic mode meaning of T as <i>true</i> .

Other reserved variables, such as `LOW_POWER`, described in other sections, are listed in Appendix A.

Using these reserved variables inappropriately will lead to errors.

There are also internal variables used by `REDUCE` that have similar restrictions. These usually have an asterisk in their names, so it is unlikely a casual user would use one. An example of such a variable is `K!*` used in the asymptotic command package.

Certain words are reserved in `REDUCE`. They may only be used in the manner intended. A list of these is given in the section “Reserved Identifiers”. There are, of course, an impossibly large number of such names to keep in mind. The reader may therefore want to make himself a copy of the list, deleting the names he doesn’t think he is likely to use by mistake.

2.5 Strings

Strings are used in `WRITE` statements, in other output statements (such as error messages), and to name files. A string consists of any number of characters enclosed in double quotes. For example:

```
"A String".
```

Lower case characters within a string are not converted to upper case.

The string `" "` represents the empty string. A double quote may be included in a string by preceding it by another double quote. Thus `"a""b"` is the string `a"b`, and `" "" "` is the string consisting of the single character `"`.

2.6 Comments

Text can be included in program listings for the convenience of human readers, in such a way that `REDUCE` pays no attention to it. There are two ways to do this:

1. Everything from the word `COMMENT` to the next statement terminator, normally `;` or `$`, is ignored. Such comments can be placed anywhere a blank could properly appear. (Note that `END` and `>>` are *not* treated as `COMMENT` delimiters!)
2. Everything from the symbol `%` to the end of the line on which it appears is ignored. Such comments can be placed as the last part of any line. Statement terminators have no special meaning in such comments. Remember to put a semicolon before the `%` if the earlier part of the line is intended to be so terminated. Remember also to begin each line of a multi-line `%` comment with a `%` sign.

2.7 Operators

Operators in `REDUCE` are specified by name and type. There are two types, infix and prefix. Operators can be purely abstract, just symbols with no properties; they can have values assigned (using `:=` or simple `LET` declarations) for specific arguments; they can have properties declared for some collection of arguments (using more general `LET` declarations); or they can be fully defined (usually by a procedure declaration).

Infix operators have a definite precedence with respect to one another, and normally occur between their arguments. For example:

<code>a + b - c</code>	(spaces optional)
<code>x<y and y=z</code>	(spaces required where shown)

Spaces can be freely inserted between operators and variables or operators and operators. They are required only where operator names are spelled out with letters (such as the `AND` in the example) and must be unambiguously separated from another such or from a variable (like `Y`). Wherever one space can be used, so can any larger number.

Prefix operators occur to the left of their arguments, which are written as a list enclosed in parentheses and separated by commas, as with normal mathematical functions, e.g.,

```
cos (u)
df (x^2, x)
q (v+w)
```

Unmatched parentheses, incorrect groupings of infix operators and the like, naturally lead to syntax errors. The parentheses can be omitted (replaced by a space following the operator name) if the operator is unary and the argument is a single symbol or begins with a prefix operator name:

<code>cos y</code>	means $\cos(y)$
<code>cos (-y)</code>	– parentheses necessary
<code>log cos y</code>	means $\log(\cos(y))$
<code>log cos (a+b)</code>	means $\log(\cos(a+b))$

but

<code>cos a*b</code>	means $(\cos a)*b$
<code>cos -y</code>	is erroneous (treated as a variable “cos” minus the variable y)

A unary prefix operator has a precedence higher than any infix operator, including unary infix operators. In other words, REDUCE will always interpret `cos y + 3` as $(\cos y) + 3$ rather than as $\cos(y + 3)$.

Infix operators may also be used in a prefix format on input, e.g., `+(a, b, c)`. On output, however, such expressions will always be printed in infix form (i.e., `a + b + c` for this example).

A number of prefix operators are built into the system with predefined properties. Users may also add new operators and define their rules for simplification. The built in operators are described in another section.

Built-In Infix Operators

The following infix operators are built into the system. They are all defined internally as procedures.

$\langle \text{infix operator} \rangle \longrightarrow$ where | := | or | and | member | memq |
= | neq | eq | >= | > | <= | < |
+ | - | * | / | ^ | ** | .

These operators may be further divided into the following subclasses:

$\langle \text{assignment operator} \rangle \longrightarrow$:=
 $\langle \text{logical operator} \rangle \longrightarrow$ or | and | member | memq
 $\langle \text{relational operator} \rangle \longrightarrow$ = | neq | eq | >= | > | <= | < |
 $\langle \text{substitution operator} \rangle \longrightarrow$ where
 $\langle \text{arithmetic operator} \rangle \longrightarrow$ + | - | * | / | ^ | **
 $\langle \text{construction operator} \rangle \longrightarrow$.

MEMQ and EQ are not used in the algebraic mode of REDUCE. They are explained in the section on symbolic mode. WHERE is described in the section on substitutions.

In previous versions of REDUCE, *not* was also defined as an infix operator. In the present version it is a regular prefix operator, and interchangeable with *null*.

For compatibility with the intermediate language used by REDUCE, each special character infix operator has an alternative alphanumeric identifier associated with it. These identifiers may be used interchangeably with the corresponding special character names on input. This correspondence is as follows:

<code>:=</code>	<code>setq</code>	(the assignment operator)
<code>=</code>	<code>equal</code>	
<code>>=</code>	<code>geq</code>	
<code>></code>	<code>greaterp</code>	
<code><=</code>	<code>leq</code>	
<code><</code>	<code>lessp</code>	
<code>+</code>	<code>plus</code>	
<code>-</code>	<code>difference</code>	(if unary, minus)
<code>*</code>	<code>times</code>	
<code>/</code>	<code>quotient</code>	(if unary, recip)
<code>^</code> or <code>**</code>	<code>expt</code>	(raising to a power)
<code>.</code>	<code>cons</code>	

Note: `NEQ` is used to mean *not equal*. There is no special symbol provided for it.

The above operators are binary, except `NOT` which is unary and `+` and `*` which are nary (i.e., taking an arbitrary number of arguments). In addition, `-` and `/` may be used as unary operators, e.g., `/2` means the same as `1/2`. Any other operator is parsed as a binary operator using a left association rule. Thus `a/b/c` is interpreted as `(a/b)/c`. There are two exceptions to this rule: `:=` and `.` are right associative. Example: `a:=b:=c` is interpreted as `a:=(b:=c)`. Unlike ALGOL and PASCAL, `^` is left associative. In other words, `a^b^c` is interpreted as `(a^b)^c`.

The operators `<`, `<=`, `>`, `>=` can only be used for making comparisons between numbers. No meaning is currently assigned to this kind of comparison between general expressions.

Parentheses may be used to specify the order of combination. If parentheses are omitted then this order is by the ordering of the precedence list defined by the right-hand side of the *<infix operator>* table at the beginning of this section, from lowest to highest. In other words, `WHERE` has the lowest precedence, and `.` (the dot operator) the highest.

Chapter 3

Expressions

REDUCE expressions may be of several types and consist of sequences of numbers, variables, operators, left and right parentheses and commas. The most common types are as follows:

3.1 Scalar Expressions

Using the arithmetic operations $+$ $-$ $*$ $/$ $^$ (power) and parentheses, scalar expressions are composed from numbers, ordinary “scalar” variables (identifiers), array names with subscripts, operator or procedure names with arguments and statement expressions.

Examples:

```
x
x^3 - 2*y/(2*z^2 - df(x,z))
(p^2 + m^2)^(1/2)*log(y/m)
a(5) + b(i,q)
```

The symbol $**$ may be used as an alternative to the caret symbol ($^$) for forming powers, particularly in those systems that do not support a caret symbol.

Statement expressions, usually in parentheses, can also form part of a scalar expression, as in the example

```
w + (c:=x+y) + z .
```

When the algebraic value of an expression is needed, REDUCE determines it, starting with the algebraic values of the parts, roughly as follows:

Variables and operator symbols with an argument list have the algebraic values

they were last assigned, or if never assigned stand for themselves. However, array elements have the algebraic values they were last assigned, or, if never assigned, are taken to be 0.

Procedures are evaluated with the values of their actual parameters.

In evaluating expressions, the standard rules of algebra are applied. Unfortunately, this algebraic evaluation of an expression is not as unambiguous as is numerical evaluation. This process is generally referred to as “simplification” in the sense that the evaluation usually but not always produces a simplified form for the expression.

There are many options available to the user for carrying out such simplification. If the user doesn’t specify any method, the default method is used. The default evaluation of an expression involves expansion of the expression and collection of like terms, ordering of the terms, evaluation of derivatives and other functions and substitution for any expressions which have values assigned or declared (see assignments and `LET` statements). In many cases, this is all that the user needs.

The declarations by which the user can exercise some control over the way in which the evaluation is performed are explained in other sections. For example, if a real (floating point) number is encountered during evaluation, the system will normally convert it into a ratio of two integers. If the user wants to use real arithmetic, he can effect this by the command `on rounded;`. Other modes for coefficient arithmetic are described elsewhere.

If an illegal action occurs during evaluation (such as division by zero) or functions are called with the wrong number of arguments, and so on, an appropriate error message is generated.

3.2 Integer Expressions

These are expressions which, because of the values of the constants and variables in them, evaluate to whole numbers.

Examples:

$$2, \quad 37 * 999, \quad (x + 3)^2 - x^2 - 6*x$$

are obviously integer expressions.

$$j + k - 2 * j^2$$

is an integer expression when J and K have values that are integers, or if not integers are such that “the variables and fractions cancel out”, as in

$$k - 7/3 - j + 2/3 + 2*j^2.$$

3.3 Boolean Expressions

A boolean expression returns a truth value. In the algebraic mode of REDUCE, boolean expressions have the syntactical form:

$$\langle expression \rangle \langle relational operator \rangle \langle expression \rangle$$

or

$$\langle boolean operator \rangle (\langle arguments \rangle)$$

or

$$\langle boolean expression \rangle \langle logical operator \rangle \langle boolean expression \rangle.$$

Parentheses can also be used to control the precedence of expressions.

In addition to the logical and relational operators defined earlier as infix operators, the following boolean operators are also defined:

EVENP (U)	determines if the number U is even or not;
FIXP (U)	determines if the expression U is integer or not;
FREEOF (U, V)	determines if the expression U does not contain the kernel V anywhere in its structure;
NUMBERP (U)	determines if U is a number or not;
ORDP (U, V)	determines if U is ordered ahead of V by some canonical ordering (based on the expression structure and an internal ordering of identifiers);
PRIMEP (U)	true if U is a prime object, i.e., any object other than 0 and plus or minus 1 which is only exactly divisible by itself or a unit.

Examples:

```
j<1
x>0 or x=-2
numberp x
fixp x and evenp x
numberp x and x neq 0
```

Boolean expressions can only appear directly within IF, FOR, WHILE, and UNTIL statements, as described in other sections. Such expressions cannot be used in place of ordinary algebraic expressions, or assigned to a variable.

NB: For those familiar with symbolic mode, the meaning of some of these operators is different in that mode. For example, NUMBERP is true only for integers and reals in symbolic mode.

When two or more boolean expressions are combined with AND, they are evaluated one by one until a *false* expression is found. The rest are not evaluated. Thus

```
numberp x and numberp y and x>y
```

does not attempt to make the $x > y$ comparison unless X and Y are both verified to be numbers.

Similarly, evaluation of a sequence of boolean expressions connected by OR stops as soon as a *true* expression is found.

NB: In a boolean expression, and in a place where a boolean expression is expected, the algebraic value 0 is interpreted as *false*, while all other algebraic values are converted to *true*. So in algebraic mode a procedure can be written for direct usage in boolean expressions, returning say 1 or 0 as its value as in

```
procedure polynomialp(u,x);
  if den(u)=1 and deg(u,x)>=1 then 1 else 0;
```

One can then use this in a boolean construct, such as

```
if polynomialp(q,z) and not polynomialp(q,y) then ...
```

In addition, any procedure that does not have a defined return value (for example, a block without a RETURN statement in it) has the boolean value *false*.

3.4 Equations

Equations are a particular type of expression with the syntax

$$\langle expression \rangle = \langle expression \rangle.$$

In addition to their role as boolean expressions, they can also be used as arguments to several operators (e.g., SOLVE), and can be returned as values.

Under normal circumstances, the right-hand-side of the equation is evaluated but not the left-hand-side. This also applies to any substitutions made by the SUB

operator. If both sides are to be evaluated, the switch `EVALLHSEQP` should be turned on.

To facilitate the handling of equations, two selectors, `LHS` and `RHS`, which return the left- and right-hand sides of an equation respectively, are provided. For example,

```
lhs(a+b=c) -> a+b
and
rhs(a+b=c) -> c.
```

3.5 Proper Statements as Expressions

Several kinds of proper statements deliver an algebraic or numerical result of some kind, which can in turn be used as an expression or part of an expression. For example, an assignment statement itself has a value, namely the value assigned. So

```
2 * (x := a+b)
```

is equal to $2 * (a+b)$, as well as having the “side-effect” of assigning the value $a+b$ to X . In context,

```
y := 2 * (x := a+b);
```

sets X to $a+b$ and Y to $2 * (a+b)$.

The sections on the various proper statement types indicate which of these statements are also useful as expressions.

Chapter 4

Lists

A list is an object consisting of a sequence of other objects (including lists themselves), separated by commas and surrounded by braces. Examples of lists are:

```
{a, b, c}
```

```
{1, a-b, c=d}
```

```
{{a}, {{b, c}, d}, e}.
```

The empty list is represented as

```
{ }.
```

4.1 Operations on Lists

Several operators in the system return their results as lists, and a user can create new lists using braces and commas. Alternatively, one can use the operator `LIST` to construct a list. An important class of operations on lists are `MAP` and `SELECT` operations. For details, please refer to the chapters on `MAP`, `SELECT` and the `FOR` command. See also the documentation on the [ASSIST](#) (chapter 16.5) package.

To facilitate the use of lists, a number of operators are also available for manipulating them. `PART(<list>, n)` for example will return the n^{th} element of a list. `LENGTH` will return the length of a list. Several operators are also defined uniquely for lists. For those familiar with them, these operators in fact mirror the operations defined for Lisp lists. These operators are as follows:

4.1.1 LIST

The operator LIST is an alternative to the usage of curly brackets. LIST accepts an arbitrary number of arguments and returns a list of its arguments. This operator is useful in cases where operators have to be passed as arguments. E.g.,

```
list(a, list(list(b, c), d), e);          ->  {{a}, {{b, c}, d}, e}
```

4.1.2 FIRST

This operator returns the first member of a list. An error occurs if the argument is not a list, or the list is empty.

4.1.3 SECOND

SECOND returns the second member of a list. An error occurs if the argument is not a list or has no second element.

4.1.4 THIRD

This operator returns the third member of a list. An error occurs if the argument is not a list or has no third element.

4.1.5 REST

REST returns its argument with the first element removed. An error occurs if the argument is not a list, or is empty.

4.1.6 . (Cons) Operator

This operator adds (“conses”) an expression to the front of a list. For example:

```
a . {b, c}          ->  {a, b, c}.
```

4.1.7 APPEND

This operator appends its first argument to its second to form a new list. *Examples:*

```
append({a, b}, {c, d})      ->  {a, b, c, d}
append({{a, b}}, {c, d})    ->  {{a, b}, c, d}.
```

4.1.8 REVERSE

The operator `REVERSE` returns its argument with the elements in the reverse order. It only applies to the top level list, not any lower level lists that may occur. Examples are:

```
reverse({a,b,c})      ->      {c,b,a}
reverse({{a,b,c},d}) ->      {d,{a,b,c}}.
```

4.1.9 List Arguments of Other Operators

If an operator other than those specifically defined for lists is given a single argument that is a list, then the result of this operation will be a list in which that operator is applied to each element of the list. For example, the result of evaluating `log{a,b,c}` is the expression `{LOG(A), LOG(B), LOG(C)}`.

There are two ways to inhibit this operator distribution. Firstly, the switch `LISTARGS`, if on, will globally inhibit such distribution. Secondly, one can inhibit this distribution for a specific operator by the declaration `LISTARGP`. For example, with the declaration `listargp log`, `log{a,b,c}` would evaluate to `LOG({A,B,C})`.

If an operator has more than one argument, no such distribution occurs.

4.1.10 Caveats and Examples

Some of the natural list operations such as *member* or *delete* are available only after loading the package [ASSIST](#) (chapter 16.5).

Please note that a non-list as second argument to `CONS` (a "dotted pair" in LISP terms) is not allowed and causes an "invalid as list" error.

```
a := 17 . 4;

***** 17 4 invalid as list
```

Also, the initialization of a scalar variable is not the empty list – one has to set list type variables explicitly, as in the following example:

```
load_package assist;

procedure lotto (n,m);
begin scalar list_1_n, luckies, hit;
  list_1_n := {};
```

```

luckies := {};
for k:=1:n do list_1_n := k . list_1_n;
for k:=1:m do
  << hit := part(list_1_n,random(n-k+1) + 1);
    list_1_n := delete(hit,list_1_n);
    luckies := hit . luckies >>;
return luckies;
end;                                     % In Germany, try lotto (49,6);

```

Another example: Find all coefficients of a multivariate polynomial with respect to a list of variables:

```

procedure allcoeffs(q,lis); % q : polynomial, lis: list of vars
  allcoeffs1 (list q,lis);

procedure allcoeffs1(q,lis);
  if lis={} then q else
    allcoeffs1(foreach qq in q join coeff(qq,first lis),
      rest lis);

```


Chapter 5

Statements

A statement is any combination of reserved words and expressions, and has the syntax

$$\langle \textit{statement} \rangle \longrightarrow \langle \textit{expression} \rangle \mid \langle \textit{proper statement} \rangle$$

A REDUCE program consists of a series of commands which are statements followed by a terminator:

$$\langle \textit{terminator} \rangle \longrightarrow ; \mid \$$$

The division of the program into lines is arbitrary. Several statements can be on one line, or one statement can be freely broken onto several lines. If the program is run interactively, statements ending with ; or \$ are not processed until an end-of-line character is encountered. This character can vary from system to system, but is normally the Return key on an ASCII terminal. Specific systems may also use additional keys as statement terminators.

If a statement is a proper statement, the appropriate action takes place.

Depending on the nature of the proper statement some result or response may or may not be printed out, and the response may or may not depend on the terminator used.

If a statement is an expression, it is evaluated. If the terminator is a semicolon, the result is printed. If the terminator is a dollar sign, the result is not printed. Because it is not usually possible to know in advance how large an expression will be, no explicit format statements are offered to the user. However, a variety of output declarations are available so that the output can be produced in different forms. These output declarations are explained in [Section 8.3.3](#).

The following sub-sections describe the types of proper statements in REDUCE.

5.1 Assignment Statements

These statements have the syntax

$$\langle \text{assignment statement} \rangle \longrightarrow \langle \text{expression} \rangle := \langle \text{expression} \rangle$$

The $\langle \text{expression} \rangle$ on the left side is normally the name of a variable, an operator symbol with its list of arguments filled in, or an array name with the proper number of integer subscript values within the array bounds. For example:

$$\begin{array}{ll} a1 := b + c & \\ h(1, m) := x - 2 * y & \text{(where } h \text{ is an operator)} \\ k(3, 5) := x - 2 * y & \text{(where } k \text{ is a 2-dim. array)} \end{array}$$

More general assignments such as $a + b := c$ are also allowed. The effect of these is explained in Section 11.2.5.

An assignment statement causes the expression on the right-hand-side to be evaluated. If the left-hand-side is a variable, the value of the right-hand-side is assigned to that unevaluated variable. If the left-hand-side is an operator or array expression, the arguments of that operator or array are evaluated, but no other simplification done. The evaluated right-hand-side is then assigned to the resulting expression. For example, if A is a single-dimensional array, $a(1+1) := b$ assigns the value B to the array element $a(2)$.

If a semicolon is used as the terminator when an assignment is issued as a command (i.e. not as a part of a group statement or procedure or other similar construct), the left-hand side symbol of the assignment statement is printed out, followed by a “:=”, followed by the value of the expression on the right.

It is also possible to write a multiple assignment statement:

$$\langle \text{expression} \rangle := \dots := \langle \text{expression} \rangle := \langle \text{expression} \rangle$$

In this form, each $\langle \text{expression} \rangle$ but the last is set to the value of the last $\langle \text{expression} \rangle$. If a semicolon is used as a terminator, each expression except the last is printed followed by a “:=” ending with the value of the last expression.

5.1.1 Set Statement

In some cases, it is desirable to perform an assignment in which *both* the left- and right-hand sides of an assignment are evaluated. In this case, the SET statement can be used with the syntax:

$$\text{SET } (\text{metaexpression}, \langle \text{expression} \rangle) ;$$

For example, the statements

```
j := 23;
set (mkid(a, j), x);
```

assigns the value X to A23.

5.2 Group Statements

The group statement is a construct used where REDUCE expects a single statement, but a series of actions needs to be performed. It is formed by enclosing one or more statements (of any kind) between the symbols << and >>, separated by semicolons or dollar signs – it doesn't matter which. The statements are executed one after another.

Examples will be given in the sections on IF and other types of statements in which the << ... >> construct is useful.

If the last statement in the enclosed group has a value, then that is also the value of the group statement. Care must be taken not to have a semicolon or dollar sign after the last grouped statement, if the value of the group is relevant: such an extra terminator causes the group to have the value NIL or zero.

5.3 Conditional Statements

The conditional statement has the following syntax:

$$\langle \text{conditional statement} \rangle \longrightarrow \text{IF } \langle \text{boolean expression} \rangle \text{ THEN } \langle \text{statement} \rangle \\ [\text{ELSE } \langle \text{statement} \rangle]$$

The boolean expression is evaluated. If this is *true*, the first $\langle \text{statement} \rangle$ is executed. If it is *false*, the second is.

Examples:

```
if x=5 then a:=b+c else d:=e+f

if x=5 and numberp y
  then <<ff:=q1; a:=b+c>>
  else <<ff:=q2; d:=e+f>>
```

Note the use of the group statement.

Conditional statements associate to the right; i.e.,

```
IF <a> THEN <b> ELSE IF <c> THEN <d> ELSE <e>
```

is equivalent to:

```
IF <a> THEN <b> ELSE (IF <c> THEN <d> ELSE <e>)
```

In addition, the construction

```
IF <a> THEN IF <b> THEN <c> ELSE <d>
```

parses as

```
IF <a> THEN (IF <b> THEN <c> ELSE <d>).
```

If the value of the conditional statement is of primary interest, it is often called a conditional expression instead. Its value is the value of whichever statement was executed. (If the executed statement has no value, the conditional expression has no value or the value 0, depending on how it is used.)

Examples:

```
a:=if x<5 then 123 else 456;
b:=u + v^(if numberp z then 10*z else 1) + w;
```

If the value is of no concern, the ELSE clause may be omitted if no action is required in the *false* case.

```
if x=5 then a:=b+c;
```

Note: As explained in Section 3.3, if a scalar or numerical expression is used in place of the boolean expression – for example, a variable is written there – the *true* alternative is followed unless the expression has the value 0.

5.4 FOR Statements

The FOR statement is used to define a variety of program loops. Its general syntax is as follows:

$$\text{FOR} \left\{ \begin{array}{l} \langle \text{var} \rangle := \langle \text{number} \rangle \left\{ \begin{array}{l} \text{STEP } \langle \text{number} \rangle \text{ UNTIL} \\ : \\ \text{EACH } \langle \text{var} \rangle \left\{ \begin{array}{l} \text{IN} \\ \text{ON} \end{array} \right\} \langle \text{list} \rangle \end{array} \right\} \langle \text{number} \rangle \end{array} \right\} \langle \text{action} \rangle \langle \text{exprn} \rangle$$

where

$\langle \text{action} \rangle \rightarrow \text{do} \mid \text{product} \mid \text{sum} \mid \text{collect} \mid \text{join}.$

The assignment form of the FOR statement defines an iteration over the indicated numerical range. If expressions that do not evaluate to numbers are used in the designated places, an error will result.

The FOR EACH form of the FOR statement is designed to iterate down a list. Again, an error will occur if a list is not used.

The action DO means that $\langle exprn \rangle$ is simply evaluated and no value kept; the statement returning 0 in this case (or no value at the top level). COLLECT means that the results of evaluating $\langle exprn \rangle$ each time are linked together to make a list, and JOIN means that the values of $\langle exprn \rangle$ are themselves lists that are joined to make one list (similar to CONC in Lisp). Finally, PRODUCT and SUM form the respective combined value out of the values of $\langle exprn \rangle$.

In all cases, $\langle exprn \rangle$ is evaluated algebraically within the scope of the current value of $\langle var \rangle$. If $\langle action \rangle$ is DO, then nothing else happens. In other cases, $\langle action \rangle$ is a binary operator that causes a result to be built up and returned by FOR. In those cases, the loop is initialized to a default value (0 for SUM, 1 for PRODUCT, and an empty list for the other actions). The test for the end condition is made before any action is taken. As in Pascal, if the variable is out of range in the assignment case, or the $\langle list \rangle$ is empty in the FOR EACH case, $\langle exprn \rangle$ is not evaluated at all.

Examples:

1. If A, B have been declared to be arrays, the following stores 5^2 through 10^2 in A(5) through A(10), and at the same time stores the cubes in the B array:

```
for i := 5 step 1 until 10 do <<a(i):=i^2; b(i):=i^3>>
```

2. As a convenience, the common construction

```
STEP 1 UNTIL
```

may be abbreviated to a colon. Thus, instead of the above we could write:

```
for i := 5:10 do <<a(i):=i^2; b(i):=i^3>>
```

3. The following sets C to the sum of the squares of 1,3,5,7,9; and D to the expression $x * (x+1) * (x+2) * (x+3) * (x+4)$:

```
c := for j:=1 step 2 until 9 sum j^2;
d := for k:=0 step 1 until 4 product (x+k);
```

4. The following forms a list of the squares of the elements of the list {a, b, c}:

```
for each x in {a,b,c} collect x^2;
```

5. The following forms a list of the listed squares of the elements of the list $\{a, b, c\}$ (i.e., $\{\{A^2\}, \{B^2\}, \{C^2\}\}$) :

```
for each x in {a,b,c} collect {x^2};
```

6. The following also forms a list of the squares of the elements of the list $\{a, b, c\}$, since the JOIN operation joins the individual lists into one list:

```
for each x in {a,b,c} join {x^2};
```

The control variable used in the FOR statement is actually a new variable, not related to the variable of the same name outside the FOR statement. In other words, executing a statement `for i := ...` doesn't change the system's assumption that $i^2 = -1$. Furthermore, in algebraic mode, the value of the control variable is substituted in $\langle exprn \rangle$ only if it occurs explicitly in that expression. It will not replace a variable of the same name in the value of that expression. For example:

```
b := a; for a := 1:2 do write b;
```

prints A twice, not 1 followed by 2.

5.5 WHILE ... DO

The FOR ... DO feature allows easy coding of a repeated operation in which the number of repetitions is known in advance. If the criterion for repetition is more complicated, WHILE ... DO can often be used. Its syntax is:

```
WHILE  $\langle boolean\ expression \rangle$  DO  $\langle statement \rangle$ 
```

The WHILE ... DO controls the single statement following DO. If several statements are to be repeated, as is almost always the case, they must be grouped using the `<< ... >>` or BEGIN ... END as in the example below.

The WHILE condition is tested each time *before* the action following the DO is attempted. If the condition is false to begin with, the action is not performed at all. Make sure that what is to be tested has an appropriate value initially.

Example:

Suppose we want to add up a series of terms, generated one by one, until we reach a term which is less than 1/1000 in value. For our simple example, let us suppose the first term equals 1 and each term is obtained from the one before by taking one third of it and adding one third its square. We would write:

```

ex:=0; term:=1;
while num(term - 1/1000) >= 0 do
    <<ex := ex+term; term:=(term + term^2)/3>>;
ex;

```

As long as TERM is greater than or equal to (\geq) 1/1000 it will be added to EX and the next TERM calculated. As soon as TERM becomes less than 1/1000 the WHILE test fails and the TERM will not be added.

5.6 REPEAT...UNTIL

REPEAT... UNTIL is very similar in purpose to WHILE... DO. Its syntax is:

```
REPEAT <statement> UNTIL <boolean expression>
```

(PASCAL users note: Only a single statement – usually a group statement – is allowed between the REPEAT and the UNTIL.)

There are two essential differences:

1. The test is performed *after* the controlled statement (or group of statements) is executed, so the controlled statement is always executed at least once.
2. The test is a test for when to stop rather than when to continue, so its “polarity” is the opposite of that in WHILE... DO.

As an example, we rewrite the example from the WHILE...DO section:

```

ex:=0; term:=1;
repeat <<ex := ex+term; term := (term + term^2)/3>>
    until num(term - 1/1000) < 0;
ex;

```

In this case, the answer will be the same as before, because in neither case is a term added to EX which is less than 1/1000.

5.7 Compound Statements

Often the desired process can best (or only) be described as a series of steps to be carried out one after the other. In many cases, this can be achieved by use of the group statement. However, each step often provides some intermediate result, until at the end we have the final result wanted. Alternatively, iterations on the steps are

needed that are not possible with constructs such as `WHILE` or `REPEAT` statements. In such cases the steps of the process must be enclosed between the words `BEGIN` and `END` forming what is technically called a *block* or *compound* statement. Such a compound statement can in fact be used wherever a group statement appears. The converse is not true: `BEGIN . . . END` can be used in ways that `<< . . . >>` cannot.

If intermediate results must be formed, local variables must be provided in which to store them. *Local* means that their values are deleted as soon as the block's operations are complete, and there is no conflict with variables outside the block that happen to have the same name. Local variables are created by a `SCALAR` declaration immediately after the `BEGIN`:

```
scalar a,b,c,z;
```

If more convenient, several `SCALAR` declarations can be given one after another:

```
scalar a,b,c;  
scalar z;
```

In place of `SCALAR` one can also use the declarations `INTEGER` or `REAL`. In the present version of `REDUCE` variables declared `INTEGER` are expected to have only integer values, and are initialized to 0. `REAL` variables on the other hand are currently treated as algebraic mode `SCALARS`.

CAUTION: `INTEGER`, `REAL` and `SCALAR` declarations can only be given immediately after a `BEGIN`. An error will result if they are used after other statements in a block (including `ARRAY` and `OPERATOR` declarations, which are global in scope), or outside the top-most block (e.g., at the top level). All variables declared `SCALAR` are automatically initialized to zero in algebraic mode (`NIL` in symbolic mode).

Any symbols not declared as local variables in a block refer to the variables of the same name in the current calling environment. In particular, if they are not so declared at a higher level (e.g., in a surrounding block or as parameters in a calling procedure), their values can be permanently changed.

Following the `SCALAR` declaration(s), if any, write the statements to be executed, one after the other, separated by delimiters (e.g., `;` or `$`) (it doesn't matter which). However, from a stylistic point of view, `;` is preferred.

The last statement in the body, just before `END`, need not have a terminator (since the `BEGIN . . . END` are in a sense brackets confining the block statements). The last statement must also be the command `RETURN` followed by the variable or expression whose value is to be the value returned by the procedure. If the `RETURN` is omitted (or nothing is written after the word `RETURN`) the procedure will have no value or the value zero, depending on how it is used (and `NIL` in symbolic mode). Remember to put a terminator after the `END`.

Example:

Given a previously assigned integer value for N , the following block will compute the Legendre polynomial of degree N in the variable X :

```
begin scalar seed,deriv,top,fact;
  seed:=1/(y^2 - 2*x*y +1)^(1/2);
  deriv:=df(seed,y,n);
  top:=sub(y=0,deriv);
  fact:=for i:=1:n product i;
  return top/fact
end;
```

5.7.1 Compound Statements with GO TO

It is possible to have more complicated structures inside the `BEGIN ... END` brackets than indicated in the previous example. That the individual lines of the program need not be assignment statements, but could be almost any other kind of statement or command, needs no explanation. For example, conditional statements, and `WHILE` and `REPEAT` constructions, have an obvious role in defining more intricate blocks.

If these structured constructs don't suffice, it is possible to use labels and `GO TO`s within a compound statement, and also to use `RETURN` in places within the block other than just before the `END`. The following subsections discuss these matters in detail. For many readers the following example, presenting one possible definition of a process to calculate the factorial of N for preassigned N will suffice:

Example:

```
begin scalar m;
  m:=1;
  l: if n=0 then return m;
  m:=m*n;
  n:=n-1;
  go to l
end;
```

5.7.2 Labels and GO TO Statements

Within a `BEGIN ... END` compound statement it is possible to label statements, and transfer to them out of sequence using `GO TO` statements. Only statements on the top level inside compound statements can be labeled, not ones inside subsidiary constructions like `<< ... >>`, `IF ... THEN ...`, `WHILE ... DO ...`, etc.

Labels and GO TO statements have the syntax:

$$\begin{array}{ll} \langle go\ to\ statement \rangle & \longrightarrow GO\ TO\ \langle label \rangle\ |\ GOTO\ \langle label \rangle \\ \langle label \rangle & \longrightarrow \langle identifier \rangle \\ \langle labeled\ statement \rangle & \longrightarrow \langle label \rangle : \langle statement \rangle \end{array}$$

Note that statement names cannot be used as labels.

While GO TO is an unconditional transfer, it is frequently used in conditional statements such as

```
if x>5 then go to abcd;
```

giving the effect of a conditional transfer.

Transfers using GO TOs can only occur within the block in which the GO TO is used. In other words, you cannot transfer from an inner block to an outer block using a GO TO. However, if a group statement occurs within a compound statement, it is possible to jump out of that group statement to a point within the compound statement using a GO TO.

5.7.3 RETURN Statements

The value corresponding to a BEGIN ... END compound statement, such as a procedure body, is normally 0 (NIL in symbolic mode). By executing a RETURN statement in the compound statement a different value can be returned. After a RETURN statement is executed, no further statements within the compound statement are executed.

Examples:

```
return x+y;
return m;
return;
```

Note that parentheses are not required around the `x+y`, although they are permitted. The last example is equivalent to `return 0` or `return nil`, depending on whether the block is used as part of an expression or not.

Since RETURN actually moves up only one block level, in a sense the casual user is not expected to understand, we tabulate some cautions concerning its use.

1. RETURN can be used on the top level inside the compound statement, i.e. as one of the statements bracketed together by the BEGIN ... END
2. RETURN can be used within a top level << ... >> construction within the

compound statement. In this case, the RETURN transfers control out of both the group statement and the compound statement.

3. RETURN can be used within an IF ... THEN ... ELSE ... on the top level within the compound statement.

NOTE: At present, there is no construct provided to permit early termination of a FOR, WHILE, or REPEAT statement. In particular, the use of RETURN in such cases results in a syntax error. For example,

```
begin scalar y;  
  y := for i:=0:99 do if a(i)=x then return b(i);  
  ...
```

will lead to an error.

Chapter 6

Commands and Declarations

A command is an order to the system to do something. Some commands cause visible results (such as calling for input or output); others, usually called declarations, set options, define properties of variables, or define procedures. Commands are formally defined as a statement followed by a terminator

$$\begin{aligned}\langle command \rangle &\longrightarrow \langle statement \rangle \langle terminator \rangle \\ \langle terminator \rangle &\longrightarrow ; \mid \$\end{aligned}$$

Some REDUCE commands and declarations are described in the following subsections.

6.1 Array Declarations

Array declarations in REDUCE are similar to FORTRAN dimension statements. For example:

```
array a(10), b(2, 3, 4);
```

Array indices each range from 0 to the value declared. An element of an array is referred to in standard FORTRAN notation, e.g. $A(2)$.

We can also use an expression for defining an array bound, provided the value of the expression is a positive integer. For example, if X has the value 10 and Y the value 7 then `array c(5*x+y)` is the same as `array c(57)`.

If an array is referenced by an index outside its range, an error occurs. If the array is to be one-dimensional, and the bound a number or a variable (not a more general expression) the parentheses may be omitted:

```
array a 10, c 57;
```

The operator `LENGTH` applied to an array name returns a list of its dimensions.

All array elements are initialized to 0 at declaration time. In other words, an array element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used instead.

Array declarations can appear anywhere in a program. Once a symbol is declared to name an array, it can not also be used as a variable, or to name an operator or a procedure. It can however be re-declared to be an array, and its size may be changed at that time. An array name can also continue to be used as a parameter in a procedure, or a local variable in a compound statement, although this use is not recommended, since it can lead to user confusion over the type of the variable.

Arrays once declared are global in scope, and so can then be referenced anywhere in the program. In other words, unlike arrays in most other languages, a declaration within a block (or a procedure) does not limit the scope of the array to that block, nor does the array go away on exiting the block (use `CLEAR` instead for this purpose).

6.2 Mode Handling Declarations

The `ON` and `OFF` declarations are available to the user for controlling various system options. Each option is represented by a *switch* name. `ON` and `OFF` take a list of switch names as argument and turn them on and off respectively, e.g.,

```
on time;
```

causes the system to print a message after each command giving the elapsed CPU time since the last command, or since `TIME` was last turned off, or the session began. Another useful switch with interactive use is `DEMO`, which causes the system to pause after each command in a file (with the exception of comments) until a Return is typed on the terminal. This enables a user to set up a demonstration file and step through it command by command.

As with most declarations, arguments to `ON` and `OFF` may be strung together separated by commas. For example,

```
off time,demo;
```

will turn off both the time messages and the demonstration switch.

We note here that while most `ON` and `OFF` commands are obeyed almost instantaneously, some trigger time-consuming actions such as reading in necessary modules from secondary storage.

A diagnostic message is printed if *ON* or *OFF* are used with a switch that is not known to the system. For example, if you misspell *DEMO* and type

```
on demq;
```

you will get the message

```
***** DEMQ not defined as switch.
```

6.3 **END**

The identifier *END* has two separate uses.

- 1) Its use in a *BEGIN ... END* bracket has been discussed in connection with compound statements.
- 2) Files to be read using *IN* should end with an extra *END;* command. The reason for this is explained in the section on the *IN* command. This use of *END* does not allow an immediately preceding *END* (such as the *END* of a procedure definition), so we advise using *;END;* there.

6.4 **BYE Command**

The command *BYE;* (or alternatively *QUIT;*) stops the execution of *REDUCE*, closes all open output files, and returns you to the calling program (usually the operating system). Your *REDUCE* session is normally destroyed.

6.5 **SHOWTIME Command**

SHOWTIME; prints the elapsed time since the last call of this command or, on its first call, since the current *REDUCE* session began. The time is normally given in milliseconds and gives the time as measured by a system clock. The operations covered by this measure are system dependent.

6.6 **DEFINE Command**

The command *DEFINE* allows a user to supply a new name for any identifier or replace it by any well-formed expression. Its argument is a list of expressions of the form

$$\langle identifier \rangle = \langle number \rangle \mid \langle identifier \rangle \mid \langle operator \rangle \mid \langle reserved\ word \rangle \mid \langle expression \rangle$$

Example:

```
define be==, x=y+z;
```

means that BE will be interpreted as an equal sign, and X as the expression y+z from then on. This renaming is done at parse time, and therefore takes precedence over any other replacement declared for the same identifier. It stays in effect until the end of the REDUCE run.

The identifiers ALGEBRAIC and SYMBOLIC have properties which prevent DEFINE from being used on them. To define ALG to be a synonym for ALGEBRAIC, use the more complicated construction

```
put('alg, 'newnam, 'algebraic);
```


Chapter 7

Built-in Prefix Operators

In the following subsections are descriptions of the most useful prefix operators built into REDUCE that are not defined in other sections (such as substitution operators). Some are fully defined internally as procedures; others are more nearly abstract operators, with only some of their properties known to the system.

In many cases, an operator is described by a prototypical header line as follows. Each formal parameter is given a name and followed by its allowed type. The names of classes referred to in the definition are printed in lower case, and parameter names in upper case. If a parameter type is not commonly used, it may be a specific set enclosed in brackets { ... }. Operators that accept formal parameter lists of arbitrary length have the parameter and type class enclosed in square brackets indicating that zero or more occurrences of that argument are permitted. Optional parameters and their type classes are enclosed in angle brackets.

7.1 Numerical Operators

REDUCE includes a number of functions that are analogs of those found in most numerical systems. With numerical arguments, such functions return the expected result. However, they may also be called with non-numerical arguments. In such cases, except where noted, the system attempts to simplify the expression as far as it can. In such cases, a residual expression involving the original operator usually remains. These operators are as follows:

7.1.1 ABS

ABS returns the absolute value of its single argument, if that argument has a numerical value. A non-numerical argument is returned as an absolute value, with an

overall numerical coefficient taken outside the absolute value operator. For example:

```
abs(-3/4)    -> 3/4
abs(2a)      -> 2*ABS(A)
abs(i)       -> 1
abs(-x)      -> ABS(X)
```

7.1.2 CEILING

This operator returns the ceiling (i.e., the least integer greater than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
ceiling(-5/4) -> -1
ceiling(-a)   -> CEILING(-A)
```

7.1.3 CONJ

This returns the complex conjugate of an expression, if that argument has a numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```
conj(1+i)    -> 1-I
conj(a+i*b)  -> REPART(A) - REPART(B)*I - IMPART(A)*I
              - IMPART(B)
```

7.1.4 FACTORIAL

If the single argument of `FACTORIAL` evaluates to a non-negative integer, its factorial is returned. Otherwise an expression involving `FACTORIAL` is returned. For example:

```
factorial(5) -> 120
factorial(a) -> FACTORIAL(A)
```

7.1.5 FIX

This operator returns the fixed value (i.e., the integer part of the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
fix(-5/4)    ->  -1
fix(a)       ->  FIX(A)
```

7.1.6 FLOOR

This operator returns the floor (i.e., the greatest integer less than the given argument) if its single argument has a numerical value. A non-numerical argument is returned as an expression in the original operator. For example:

```
floor(-5/4)   ->  -2
floor(a)      ->  FLOOR(A)
```

7.1.7 IMPART

This operator returns the imaginary part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```
impart(1+i)   ->  1
impart(a+i*b) ->  REPART(B) + IMPART(A)
```

7.1.8 MAX/MIN

`MAX` and `MIN` can take an arbitrary number of expressions as their arguments. If all arguments evaluate to numerical values, the maximum or minimum of the argument list is returned. If any argument is non-numeric, an appropriately reduced expression is returned. For example:

```
max(2,-3,4,5) ->  5
min(2,-2)     ->  -2.
max(a,2,3)    ->  MAX(A,3)
min(x)        ->  X
```

`MAX` or `MIN` of an empty list returns 0.

7.1.9 NEXTPRIME

`NEXTPRIME` returns the next prime greater than its integer argument, using a probabilistic algorithm. A type error occurs if the value of the argument is not an integer. For example:

```

nextprime(5)      -> 7
nextprime(-2)     -> 2
nextprime(-7)     -> -5
nextprime 1000000 -> 1000003

```

whereas `nextprime(a)` gives a type error.

7.1.10 RANDOM

`random(n)` returns a random number r in the range $0 \leq r < n$. A type error occurs if the value of the argument is not a positive integer in algebraic mode, or positive number in symbolic mode. For example:

```

random(5)      -> 3
random(1000)   -> 191

```

whereas `random(a)` gives a type error.

7.1.11 RANDOM_NEW_SEED

`random_new_seed(n)` reseeds the random number generator to a sequence determined by the integer argument n . It can be used to ensure that a repeatable pseudo-random sequence will be delivered regardless of any previous use of `RANDOM`, or can be called early in a run with an argument derived from something variable (such as the time of day) to arrange that different runs of a `REDUCE` program will use different random sequences. When a fresh copy of `REDUCE` is first created it is as if `random_new_seed(1)` has been obeyed.

A type error occurs if the value of the argument is not a positive integer.

7.1.12 REPART

This returns the real part of an expression, if that argument has an numerical value. A non-numerical argument is returned as an expression in the operators `REPART` and `IMPART`. For example:

```

repart(1+i)      -> 1
repart(a+i*b)    -> REPART(A) - IMPART(B)

```

7.1.13 ROUND

This operator returns the rounded value (i.e, the nearest integer) of its single argument if that argument has a numerical value. A non-numeric argument is returned as an expression in the original operator. For example:

```
round(-5/4)    ->  -1
round(a)       ->  ROUND(A)
```

7.1.14 SIGN

SIGN tries to evaluate the sign of its argument. If this is possible SIGN returns one of 1, 0 or -1. Otherwise, the result is the original form or a simplified variant. For example:

```
sign(-5)       ->  -1
sign(-a^2*b)   ->  -SIGN(B)
```

Note that even powers of formal expressions are assumed to be positive only as long as the switch COMPLEX is off.

7.2 Mathematical Functions

REDUCE knows that the following represent mathematical functions that can take arbitrary scalar expressions as their argument(s):

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 BETA CI COS COSH COT COTH CSC CSCH
DILOG EI EXP GAMMA HYPOT IBETA IGAMMA LN LOG LOGB LOG10
SEC SECH SI SIN SINH Sqrt TAN TANH
AIRY_AI AIRY_AIPRIME AIRY_BI AIRY_BIPRIME
BESSELI BESSELJ BESSELK BESSELY
HANKEL1 HANKEL2 KUMMER KUMMERU LOMMEL1 LOMMEL2
STRUVEH STRUVEL WHITTAKERM WHITTAKERU
POLYGAMMA PSI ZETA
SOLIDHARMONICY SPHERICALHARMONICY
```

where LOG is the natural logarithm (and equivalent to LN), and LOGB has two arguments of which the second is the logarithmic base.

The derivatives of all these functions are also known to the system.

REDUCE knows various elementary identities and properties of these functions. For example:

$$\begin{array}{ll}
 \cos(-x) = \cos(x) & \sin(-x) = -\sin(x) \\
 \cos(n\pi) = (-1)^n & \sin(n\pi) = 0 \\
 \log(e) = 1 & e^{(i\pi/2)} = i \\
 \log(1) = 0 & e^{(i\pi)} = -1 \\
 \log(e^x) = x & e^{(3i\pi/2)} = -i
 \end{array}$$

Beside these identities, there are a lot of simplifications for elementary functions defined in the REDUCE system as rulelists. In order to view these, the SHOWRULES operator can be used, e.g.

```

SHOWRULES tan;

{tan(~n*arbint(~i)*pi + ~(~ x)) => tan(x) when fixp(n),

tan(~x)

=> trigquot(sin(x),cos(x)) when knowledge_about(sin,x,tan)

,

~x + ~(~ k)*pi
tan(-----)
~d

=> -cot(---) when x freeof pi and abs(---)=---,
      x      k      1
      d      d      2

~(~ w) + ~(~ k)*pi      w + remainder(k,d)*pi
tan(-----) => tan(-----)
~(~ d)                  d

when w freeof pi and ratnump(---) and fixp(k)
                        k
                        d

and abs(---)>=1,
      k
      d

tan(atan(~x)) => x,

```

$$\text{df}(\tan(\tilde{x}), \tilde{x}) \Rightarrow 1 + \tan^2(x) \}$$

For further simplification, especially of expressions involving trigonometric functions, see the TRIGSIMP package (chapter 16.62) documentation.

Functions not listed above may be defined in the special functions package SPECFN.

The user can add further rules for the reduction of expressions involving these operators by using the LET command.

In many cases it is desirable to expand product arguments of logarithms, or collect a sum of logarithms into a single logarithm. Since these are inverse operations, it is not possible to provide rules for doing both at the same time and preserve the REDUCE concept of idempotent evaluation. As an alternative, REDUCE provides two switches EXPANDLOGS and COMBINELOGS to carry out these operations. Both are off by default, and are subject to the value of the switch PRECISE. This switch is on by default and prevents modifications that may be false in a complex domain. Thus to expand $\text{LOG}(3*Y)$ into a sum of logs, one can say

```
ON EXPANDLOGS; LOG(3*Y);
```

whereas to expand $\text{LOG}(X*Y)$ into a sum of logs, one needs to say

```
OFF PRECISE; ON EXPANDLOGS; LOG(X*Y);
```

To combine this sum into a single log:

```
OFF PRECISE; ON COMBINELOGS; LOG(X) + LOG(Y);
```

These switches affect the logarithmic functions LOG10 (base 10) and LOGB (arbitrary base) as well.

At the present time, it is possible to have both switches on at once, which could lead to infinite recursion. However, an expression is switched from one form to the other in this case. Users should not rely on this behavior, since it may change in the next release.

The current version of REDUCE does a poor job of simplifying surds. In particular, expressions involving the product of variables raised to non-integer powers do not usually have their powers combined internally, even though they are printed as if those powers were combined. For example, the expression

```
x^(1/3)*x^(1/6);
```

will print as

$$\text{SQRT}(X)$$

but will have an internal form containing the two exponentiated terms. If you now subtract `sqrt(x)` from this expression, you will *not* get zero. Instead, the confusing form

$$\text{SQRT}(X) - \text{SQRT}(X)$$

will result. To combine such exponentiated terms, the switch `COMBINEEXPT` should be turned on.

The square root function can be input using the name `SQRT`, or the power operation $^{(1/2)}$. On output, unsimplified square roots are normally represented by the operator `SQRT` rather than a fractional power. With the default system switch settings, the argument of a square root is first simplified, and any divisors of the expression that are perfect squares taken outside the square root argument. The remaining expression is left under the square root. Thus the expression

$$\text{sqrt}(-8a^2b)$$

becomes

$$2*a*\text{sqrt}(-2*b).$$

Note that such simplifications can cause trouble if `A` is eventually given a value that is a negative number. If it is important that the positive property of the square root and higher even roots always be preserved, the switch `PRECISE` should be set on (the default value). This causes any non-numerical factors taken out of surds to be represented by their absolute value form. With `PRECISE` on then, the above example would become

$$2*\text{abs}(a)*\text{sqrt}(-2*b).$$

However, this is incorrect in the complex domain, where the $\sqrt{x^2}$ is not identical to $|x|$. To avoid the above simplification, the switch `PRECISE_COMPLEX` should be set on (default is off). For example:

```
on precise_complex; sqrt(-8a^2*b);
```

yields the output


```
2*sqrt( - 2*a *b)
```

The statement that REDUCE knows very little about these functions applies only in the mathematically exact off rounded mode. If ROUNDED is on, any of the functions

```
ACOS ACOSH ACOT ACOTH ACSC ACSCH ASEC ASECH ASIN ASINH
ATAN ATANH ATAN2 COS COSH COT COTH CSC CSCH EXP HYPOT
IBETA IGAMMA LN LOG LOGB LOG10 SEC SECH SIN SINH SQRT TAN TANH
```

when given a numerical argument has its value calculated to the current degree of floating point precision. In addition, real (non-integer valued) powers of numbers will also be evaluated.

If the COMPLEX switch is turned on in addition to ROUNDED, these functions will also calculate a real or complex result, again to the current degree of floating point precision, if given complex arguments. For example, with on rounded, complex;

```
2.3^(5.6i)    ->  -0.0480793490914 - 0.998843519372*I
cos(2+3i)     ->  -4.18962569097 - 9.10922789376*I
```

7.3 Bernoulli Numbers and Euler Numbers

The unary operator `Bernoulli` provides notation and computation for Bernoulli numbers. `Bernoulli(n)` evaluates to the n th Bernoulli number; all of the odd Bernoulli numbers, except `Bernoulli(1)`, are zero.

The algorithms are based upon those by Herbert Wilf, presented by Sandra Filibrown [?]. If the ROUNDED switch is off, the algorithms are exactly those; if it is on, some further rounding may be done to prevent computation of redundant digits. Hence, these functions are particularly fast when used to approximate the Bernoulli numbers in rounded mode.

Euler numbers are computed by the unary operator `Euler`, which return the n th Euler number. The computation is derived directly from Pascal's triangle of binomial coefficients.

7.4 Fibonacci Numbers and Fibonacci Polynomials

The unary operator `Fibonacci` provides notation and computation for Fibonacci numbers. `Fibonacci(n)` evaluates to the n th Fibonacci number. If n is a positive or negative integer, it will be evaluated following the definition:

$$F_0 = 0; F_1 = 1; F_n = F_{n-1} + F_{n-2}$$

Fibonacci Polynomials are computed by the binary operator `FibonacciP`. `FibonacciP(n,x)` returns the n th Fibonacci polynomial in the variable x . If n is a positive or negative integer, it will be evaluated following the definition:

$$F_0(x) = 0; F_1(x) = 1; F_n(x) = xF_{n-1}(x) + F_{n-2}(x)$$

7.5 Motzkin numbers

A Motzkin number M_n (named after Theodore Motzkin) is the number of different ways of drawing non-intersecting chords on a circle between n points. For a non-negative integer n , the operator `Motzkin(n)` returns the n th Motzkin number, according to the recursion formula

$$M_0 = 1; \quad M_1 = 1; \quad M_{n+1} = \frac{2n+3}{n+3}M_n + \frac{3n}{n+3}M_{n-1}.$$

7.6 CONTINUED_FRACTION Operator

The operator `CONTINUED_FRACTION` approximates the real number (rational number, rounded number) into a continued fraction. `CONTINUED_FRACTION` has one or two arguments, the number to be converted and an optional precision:

```
continued_fraction(<num>)
```

or

```
continued_fraction(<num>, <size>)
```

The result is a list of two elements: the first one is the rational value of the approximation, the second one is the list of terms of the continued fraction which represents the same value according to the definition $t_0 + 1/(t_1 + 1/(t_2 + \dots))$. Precision: the second optional parameter `<size>` is an upper bound for the absolute value of the result denominator. If omitted, the approximation is performed up to the current system precision.

Examples:

```
continued_fraction pi;
```

->

```
1146408
```

```
{-----, {3, 7, 15, 1, 292, 1, 1, 1, 2, 1}}
 364913
```

```
continued_fraction(pi, 100);
```

```
->
```

```
 22
{----, {3, 7}}
 7
```

7.7 DF Operator

The operator `DF` is used to represent partial differentiation with respect to one or more variables. It is used with the syntax:

```
DF (EXPRN:algebraic[, VAR:kernel<, NUM:integer>]):algebraic.
```

The first argument is the expression to be differentiated. The remaining arguments specify the differentiation variables and the number of times they are applied.

The number `NUM` may be omitted if it is 1. For example,

```
df (y, x)           = ∂y/∂x
df (y, x, 2)        = ∂²y/∂x²
df (y, x1, 2, x2, x3, 2) = ∂⁵y/∂x₁² ∂x₂ ∂x₃².
```

The evaluation of `df (y, x)` proceeds as follows: first, the values of `Y` and `X` are found. Let us assume that `X` has no assigned value, so its value is `X`. Each term or other part of the value of `Y` that contains the variable `X` is differentiated by the standard rules. If `Z` is another variable, not `X` itself, then its derivative with respect to `X` is taken to be 0, unless `Z` has previously been declared to `DEPEND` on `X`, in which case the derivative is reported as the symbol `df (z, x)`.

7.7.1 Switches influencing differentiation

Consider `df (u, x, y, z)`. If non of `x, y, z` are equal to `u` then the order of differentiation is commuted into a canonical form, unless the switch `NOCOMMUTEDF` is turned on (default is off). if at least one of `x, y, z` is equal to `u` then the order of differentiation is *not* commuted and the derivative is *not* simplified to zero, unless the switch `COMMUTEDF` is turned on. It is off by default.

If `COMMUTEDF` is off and the switch `SIMPNONCOMDF` is on then simplify as follows:

$$\begin{aligned} \text{DF}(U, X, U) &\rightarrow \text{DF}(U, X, 2) / \text{DF}(U, X) \\ \text{DF}(U, X, N, U) &\rightarrow \text{DF}(U, X, N+1) / \text{DF}(U, X) \end{aligned}$$

provided U depends only on the one variable X . This simplification removes the non-commutative aspect of the derivative.

If the switch `EXPANDDF` is turned on then `REDUCE` uses the chain rule to expand symbolic derivatives of indirectly dependent variables provided the result is unambiguous, i.e. provided there is no direct dependence. It is off by default. Thus, for example, given

$$\begin{aligned} &\text{DEPEND } F, U, V; \text{ DEPEND } \{U, V\}, X; \\ &\text{ON EXPANDDF;} \\ &\text{DF}(F, X) \rightarrow \text{DF}(F, U) * \text{DF}(U, X) + \text{DF}(F, V) * \text{DF}(V, X) \end{aligned}$$

whereas after

$$\text{DEPEND } F, X;$$

$\text{DF}(F, X)$ does not expand at all (since the result would be ambiguous and the algorithm would loop).

Turning on the switch `ALLOWDFINT` allows "differentiation under the integral sign", i.e.

$$\text{DF}(\text{INT}(Y, X), V) \rightarrow \text{INT}(\text{DF}(Y, V), X)$$

if this results in a simplification. If the switch `DFINT` is also turned on then this happens regardless of whether the result simplifies. Both switches are off by default.

7.7.2 Adding Differentiation Rules

The `LET` statement can be used to introduce rules for differentiation of user-defined operators. Its general form is

$$\begin{aligned} &\text{FOR ALL } \langle var1 \rangle, \dots, \langle varn \rangle \\ &\text{LET DF}(\langle operator \rangle \langle varlist \rangle, \langle vari \rangle) = \langle expression \rangle \end{aligned}$$

where

$$\langle varlist \rangle \rightarrow (\langle var1 \rangle, \dots, \langle varn \rangle),$$

and $\langle var1 \rangle, \dots, \langle varn \rangle$ are the dummy variable arguments of $\langle operator \rangle$.

An analogous form applies to infix operators.

Examples:

```
for all x let df(tan x, x) = 1 + tan(x)^2;
```

(This is how the tan differentiation rule appears in the REDUCE source.)

```
for all x, y let df(f(x, y), x) = 2*f(x, y),
                df(f(x, y), y) = x*f(x, y);
```

Notice that all dummy arguments of the relevant operator must be declared arbitrary by the `FOR ALL` command, and that rules may be supplied for operators with any number of arguments. If no differentiation rule appears for an argument in an operator, the differentiation routines will return as result an expression in terms of `DF`. For example, if the rule for the differentiation with respect to the second argument of `F` is not supplied, the evaluation of `df(f(x, z), z)` would leave this expression unchanged. (No `DEPEND` declaration is needed here, since `f(x, z)` obviously “depends on” `z`.)

Once such a rule has been defined for a given operator, any future differentiation rules for that operator must be defined with the same number of arguments for that operator, otherwise we get the error message

```
Incompatible DF rule argument length for <operator>
```

7.8 INT Operator

`INT` is an operator in `REDUCE` for indefinite integration using a combination of the Risch-Norman algorithm and pattern matching. It is used with the syntax:

```
INT(EXPRN:algebraic, VAR:kernel):algebraic.
```

This will return correctly the indefinite integral for expressions comprising polynomials, log functions, exponential functions and `tan` and `atan`. The arbitrary constant is not represented. If the integral cannot be done in closed terms, it returns a formal integral for the answer in one of two ways:

1. It returns the input, `INT(...)` unchanged.
2. It returns an expression involving `INTs` of some other functions (sometimes more complicated than the original one, unfortunately).

Rational functions can be integrated when the denominator is factorizable by the program. In addition it will attempt to integrate expressions involving error functions, dilogarithms and other trigonometric expressions. In these cases it might not always succeed in finding the solution, even if one exists.

Examples:

```
int(log(x), x) -> X*(LOG(X) - 1),
int(e^x, x)    -> E**X.
```

The program checks that the second argument is a variable and gives an error if it is not.

Note: If the `int` operator is called with 4 arguments, REDUCE will implicitly call the definite integration package (DEFINT) and this package will interpret the third and fourth arguments as the lower and upper limit of integration, respectively. For details, consult the documentation on the DEFINT package.

7.8.1 Options

The switch `TRINT` when on will trace the operation of the algorithm. It produces a great deal of output in a somewhat illegible form, and is not of much interest to the general user. It is normally off.

The switch `TRINTSUBST` when on will trace the heuristic attempts to solve the integral by substitution. It is normally off.

If the switch `FAILHARD` is on the algorithm will terminate with an error if the integral cannot be done in closed terms, rather than return a formal integration form. `FAILHARD` is normally off.

The switch `NOLNR` suppresses the use of the linear properties of integration in cases when the integral cannot be found in closed terms. It is normally off.

The switch `NOINTSUBST` disables the heuristic attempts to solve the integral by substitution. It is normally off.

7.8.2 Advanced Use

If a function appears in the integrand that is not one of the functions `EXP`, `ERF`, `TAN`, `ATAN`, `LOG`, `DIALOG` then the algorithm will make an attempt to integrate the argument if it can, differentiate it and reach a known function. However the answer cannot be guaranteed in this case. If a function is known to be algebraically independent of this set it can be flagged transcendental by

```
flag('trilog)', 'transcendental');
```

in which case this function will be added to the permitted field descriptors for a genuine decision procedure. If this is done the user is responsible for the mathematical correctness of his actions.

The standard version does not deal with algebraic extensions. Thus integration of expressions involving square roots and other like things can lead to trouble. A contributed package that supports integration of functions involving square roots is available, however (ALGINT, chapter 16.1). In addition there is a definite integration package, DEFINT(chapter 16.16).

7.8.3 References

A. C. Norman & P. M. A. Moore, “Implementing the New Risch Algorithm”, Proc. 4th International Symposium on Advanced Comp. Methods in Theor. Phys., CNRS, Marseilles, 1977.

S. J. Harrington, “A New Symbolic Integration System in Reduce”, Comp. Journ. 22 (1979) 2.

A. C. Norman & J. H. Davenport, “Symbolic Integration — The Dust Settles?”, Proc. EUROSAM 79, Lecture Notes in Computer Science 72, Springer-Verlag, Berlin Heidelberg New York (1979) 398-407.

7.9 LENGTH Operator

LENGTH is a generic operator for finding the length of various objects in the system. The meaning depends on the type of the object. In particular, the length of an algebraic expression is the number of additive top-level terms its expanded representation.

Examples:

```
length(a+b)    ->  2
length(2)      ->  1.
```

Other objects that support a length operator include arrays, lists and matrices. The explicit meaning in these cases is included in the description of these objects.

7.10 MAP Operator

The MAP operator applies a uniform evaluation pattern to all members of a composite structure: a matrix, a list, or the arguments of an operator expression. The

evaluation pattern can be a unary procedure, an operator, or an algebraic expression with one free variable.

It is used with the syntax:

```
MAP (U: function, V: object)
```

Here `object` is a list, a matrix or an operator expression. `Function` can be one of the following:

1. the name of an operator for a single argument: the operator is evaluated once with each element of `object` as its single argument;
2. an algebraic expression with exactly one free variable, that is a variable preceded by the tilde symbol. The expression is evaluated for each element of `object`, where the element is substituted for the free variable;
3. a replacement rule of the form `var => rep` where `var` is a variable (a kernel without a subscript) and `rep` is an expression that contains `var`. `Rep` is evaluated for each element of `object` where the element is substituted for `var`. `Var` may be optionally preceded by a tilde.

The rule form for `function` is needed when more than one free variable occurs.

Examples:

```
map(abs, {1, -2, a, -a}) -> {1, 2, ABS(A), ABS(A)}
map(int(~w, x), mat((x^2, x^5), (x^4, x^5))) ->
```

```
[ 3      6 ]
[ x      x ]
[---- ----]
[ 3      6 ]
[      ]
[ 5      6 ]
[ x      x ]
[---- ----]
[ 5      6 ]
```

```
map(~w*6, x^2/3 = y^3/2 -1) -> 2*X^2=3*(Y^3-2)
```

You can use `MAP` in nested expressions. However, you cannot apply `MAP` to a non-composed object, e.g. an identifier or a number.

7.11 MKID Operator

In many applications, it is useful to create a set of identifiers for naming objects in a consistent manner. In most cases, it is sufficient to create such names from two components. The operator MKID is provided for this purpose. Its syntax is:

`MKID (U:id,V:id|non-negative integer):id`

for example

```
mkid(a,3)      -> A3
mkid(apple,s)  -> APPLES
```

while `mkid(a+b,2)` gives an error.

The SET operator can be used to give a value to the identifiers created by MKID, for example

```
set(mkid(a,3),3);
```

will give A3 the value 2.

7.12 The Pochhammer Notation

The Pochhammer notation $(a)_k$ is supported by the binary operator POCHHAMMER. With `ROUNDED off`, this expression is evaluated numerically if a and k are both integral, and otherwise may be simplified where appropriate. The simplification rules are based upon algorithms supplied by Wolfram Koepf [?].

7.13 PF Operator

`PF (<exp>,<var>)` transforms the expression $\langle exp \rangle$ into a list of partial fractions with respect to the main variable, $\langle var \rangle$. PF does a complete partial fraction decomposition, and as the algorithms used are fairly unsophisticated (factorization and the extended Euclidean algorithm), the code may be unacceptably slow in complicated cases.

Example: Given $2 / ((x+1)^2 * (x+2))$ in the workspace, `pf(ws,x);` gives the result

$$\left\{ \frac{2}{-----}, -\frac{2}{-----}, \frac{2}{-----} \right\} .$$

$$\frac{X^2 + 2}{X^2 + 2X + 1} \quad \frac{X + 1}{X^2 + 2X + 1}$$

If you want the denominators in factored form, use `off exp;`. Thus, with `2 / ((x+1)^2 * (x+2))` in the workspace, the commands `off exp; pf(ws, x);` give the result

$$\left\{ \frac{2}{X^2 + 2}, \frac{-2}{X + 1}, \frac{2}{(X + 1)^2} \right\}.$$

To recombine the terms, `FOR EACH ... SUM` can be used. So with the above list in the workspace, `for each j in ws sum j;` returns the result

$$\frac{2}{(X + 2) * (X + 1)^2}$$

Alternatively, one can use the operations on lists to extract any desired term.

7.14 SELECT Operator

The `SELECT` operator extracts from a list, or from the arguments of an n -ary operator, elements corresponding to a boolean predicate. It is used with the syntax:

```
SELECT(U:function,V:list)
```

Function can be one of the following forms:

1. the name of an operator for a single argument: the operator is evaluated once with each element of `object` as its single argument;
2. an algebraic expression with exactly one free variable, that is a variable preceded by the tilde symbol. The expression is evaluated for each element of `<object>`, where the element is substituted for the free variable;
3. a replacement rule of the form `<var> => <rep>` where `<var>` is a variable (a kernel without subscript) and `<rep>` is an expression that contains `<var>`. `<rep>` is evaluated for each element of `object` where the element is substituted for `<var>`. `<var>` may be optionally preceded by a tilde.

The rule form for `function` is needed when more than one free variable occurs.

The result of evaluating `function` is interpreted as a boolean value corresponding to the conventions of REDUCE. These values are composed with the leading operator of the input expression.

Examples:

```
select( ~w>0 , {1,-1,2,-3,3}) -> {1,2,3}
select(evenp deg(~w,y),part((x+y)^5,0):=list)
      -> {X^5 ,10*X^3*Y^2 ,5*X*Y^4}
select(evenp deg(~w,x),2x^2+3x^3+4x^4) -> 4X^4 + 2X^2
```

7.15 SOLVE Operator

SOLVE is an operator for solving one or more simultaneous algebraic equations. It is used with the syntax:

```
SOLVE (EXPRN:algebraic[,VAR:kernel|,VARLIST:list of kernels])
      :list.
```

EXPRN is of the form $\langle expression \rangle$ or $\{ \langle expression1 \rangle, \langle expression2 \rangle, \dots \}$. Each expression is an algebraic equation, or is the difference of the two sides of the equation. The second argument is either a kernel or a list of kernels representing the unknowns in the system. This argument may be omitted if the number of distinct, non-constant, top-level kernels equals the number of unknowns, in which case these kernels are presumed to be the unknowns.

For one equation, SOLVE recursively uses factorization and decomposition, together with the known inverses of LOG, SIN, COS, ^, ACOS, ASIN, and linear, quadratic, cubic, quartic, or binomial factors. Solutions of equations built with exponentials or logarithms are often expressed in terms of Lambert's W function. This function is (partially) implemented in the special functions package.

Linear equations are solved by the multi-step elimination method due to Bareiss, unless the switch CRAMER is on, in which case Cramer's method is used. The Bareiss method is usually more efficient unless the system is large and dense.

Non-linear equations are solved using the Groebner basis package (chapter 16.26). Users should note that this can be quite a time consuming process.

Examples:

```
solve(log(sin(x+3))^5 = 8, x);
solve(a*log(sin(x+3))^5 - b, sin(x+3));
solve({a*x+y=3, y=-2}, {x, y});
```

SOLVE returns a list of solutions. If there is one unknown, each solution is an equation for the unknown. If a complete solution was found, the unknown will appear by itself on the left-hand side of the equation. On the other hand, if the solve package could not find a solution, the "solution" will be an equation for the unknown in terms of the operator ROOT_OF. If there are several unknowns, each solution will be a list of equations for the unknowns. For example,

```
solve(x^2=1, x);                                -> {X=-1, X=1}

solve(x^7-x^6+x^2=1, x)
                                     6
-> {X=ROOT_OF(X_  + X_ + 1, X_, TAG_1), X=1}
```

```
solve({x+3y=7,y-x=1},{x,y}) -> {{X=1,Y=2}}.
```

The TAG argument is used to uniquely identify those particular solutions. Solution multiplicities are stored in the global variable `ROOT_MULTIPPLICITIES` rather than the solution list. The value of this variable is a list of the multiplicities of the solutions for the last call of `SOLVE`. For example,

```
solve(x^2=2x-1,x); root_multiplicities;
```

gives the results

```
{X=1}
```

```
{2}
```

If you want the multiplicities explicitly displayed, the switch `MULTIPLICITIES` can be turned on. For example

```
on multiplicities; solve(x^2=2x-1,x);
```

yields the result

```
{X=1,X=1}
```

7.15.1 Handling of Undetermined Solutions

When `SOLVE` cannot find a solution to an equation, it normally returns an equation for the relevant indeterminates in terms of the operator `ROOT_OF`. For example, the expression

```
solve(cos(x) + log(x),x);
```

returns the result

```
{X=ROOT_OF(COS(X_) + LOG(X_),X_,TAG_1)} .
```

An expression with a top-level `ROOT_OF` operator is implicitly a list with an unknown number of elements (since we don't always know how many solutions an equation has). If a substitution is made into such an expression, closed form solutions can emerge. If this occurs, the `ROOT_OF` construct is replaced by an operator `ONE_OF`. At this point it is of course possible to transform the result of the original `SOLVE` operator expression into a standard `SOLVE` solution. To effect this, the operator `EXPAND_CASES` can be used.

The following example shows the use of these facilities:

```
solve(-a*x^3+a*x^2+x^4-x^3-4*x^2+4,x);
      2      3
{X=ROOT_OF(A*X_ - X_ + 4*X_ + 4,X_,TAG_2),X=1}

sub(a=-1,ws);

{X=ONE_OF({2,-1,-2},TAG_2),X=1}

expand_cases ws;

{X=2,X=-1,X=-2,X=1}
```

7.15.2 Solutions of Equations Involving Cubics and Quartics

Since roots of cubics and quartics can often be very messy, a switch `FULLROOTS` is available, that, when off (the default), will prevent the production of a result in closed form. The `ROOT_OF` construct will be used in this case instead.

In constructing the solutions of cubics and quartics, trigonometrical forms are used where appropriate. This option is under the control of a switch `TRIGFORM`, which is normally on.

The following example illustrates the use of these facilities:

```
let xx = solve(x^3+x+1,x);

xx;
      3
{X=ROOT_OF(X_ + X_ + 1,X_)}

on fullroots;

xx;
      - Sqrt(31)*I
      ATAN(-----)
              3*Sqrt(3)
{X=(I*(Sqrt(3))*SIN(-----))
                      3}
```

$$- \frac{\sqrt{31} \cdot I}{3 \sqrt{3}} \operatorname{ATAN}\left(\frac{-\sqrt{31} \cdot I}{3 \sqrt{3}}\right) - \cos\left(\frac{-\sqrt{31} \cdot I}{3 \sqrt{3}}\right) / \sqrt{3},$$

$$X = \left(-I \cdot (\sqrt{3}) \cdot \sin\left(\frac{-\sqrt{31} \cdot I}{3 \sqrt{3}}\right) \right.$$

$$\left. + \cos\left(\frac{-\sqrt{31} \cdot I}{3 \sqrt{3}}\right) \right) / \sqrt{3}$$

3),

$$X = \frac{2 \cdot \cos\left(\frac{-\sqrt{31} \cdot I}{3 \sqrt{3}}\right) \cdot I}{\sqrt{3}}$$

off trigform;

xx;

$$\{X = \left(-(\sqrt{31}) - 3\sqrt{3} \right)^{2/3} \cdot \sqrt{3} \cdot I - (\sqrt{31}) - 3\sqrt{3} \right)^{2/3} - 2^{2/3} \cdot \sqrt{3} \cdot I + 2^{2/3} / (2 \cdot (\sqrt{31}) - 3\sqrt{3})^{1/3} \cdot 6^{1/3} \cdot 3^{1/6} \right),$$

$$2/3$$

$$\begin{aligned}
X = & ((\text{SQRT}(31) - 3*\text{SQRT}(3)) * \text{SQRT}(3) * I \\
& - (\text{SQRT}(31) - 3*\text{SQRT}(3))^{2/3} + 2^{2/3} * \text{SQRT}(3) * I \\
& + 2^{2/3}) / (2 * (\text{SQRT}(31) - 3*\text{SQRT}(3))^{1/3} * 6^{1/3} \\
& * 3^{1/6}), \\
X = & \frac{(\text{SQRT}(31) - 3*\text{SQRT}(3))^{2/3} - 2^{2/3}}{(\text{SQRT}(31) - 3*\text{SQRT}(3))^{1/3} * 6^{1/3} * 3^{1/6}}
\end{aligned}$$

7.15.3 Other Options

If SOLVESINGULAR is on (the default setting), degenerate systems such as $x+y=0$, $2x+2y=0$ will be solved by introducing appropriate arbitrary constants. The consistent singular equation $0=0$ or equations involving functions with multiple inverses may introduce unique new indeterminant kernels $\text{ARBCOMPLEX}(j)$, or $\text{ARBINT}(j)$, ($j=1,2,\dots$), representing arbitrary complex or integer numbers respectively. To automatically select the principal branches, do `off allbranch`; . `ALLBRANCH` To avoid the introduction of new indeterminant kernels do `OFF ARBVARS` – then no equations are generated for the free variables and their original names are used to express the solution forms. To suppress solutions of consistent singular equations do `OFF SOLVESINGULAR`.

To incorporate additional inverse functions do, for example:

```
put('sinh','inverse','asinh);
put('asinh','inverse','sinh);
```

together with any desired simplification rules such as

```
for all x let sinh(asinh(x))=x, asinh(sinh(x))=x;
```

For completeness, functions with non-unique inverses should be treated as \wedge , `SIN`, and `COS` are in the `SOLVE` module source.

Arguments of `ASIN` and `ACOS` are not checked to ensure that the absolute value of the real part does not exceed 1; and arguments of `LOG` are not checked to ensure

that the absolute value of the imaginary part does not exceed π ; but checks (perhaps involving user response for non-numerical arguments) could be introduced using LET statements for these operators.

7.15.4 Parameters and Variable Dependency

The proper design of a variable sequence supplied as a second argument to SOLVE is important for the structure of the solution of an equation system. Any unknown in the system not in this list is considered totally free. E.g. the call

```
solve ({x=2*z, z=2*y}, {z}) ;
```

produces an empty list as a result because there is no function $z = z(x, y)$ which fulfills both equations for arbitrary x and y values. In such a case the share variable REQUIREMENTS displays a set of restrictions for the parameters of the system:

```
requirements;
```

```
{x - 4*y}
```

The non-existence of a formal solution is caused by a contradiction which disappears only if the parameters of the initial system are set such that all members of the requirements list take the value zero. For a linear system the set is complete: a solution of the requirements list makes the initial system solvable. E.g. in the above case a substitution $x = 4y$ makes the equation set consistent. For a non-linear system only one inconsistency is detected. If such a system has more than one inconsistency, you must reduce them one after the other.¹ The set shows you also the dependency among the parameters: here one of x and y is free and a formal solution of the system can be computed by adding it to the variable list of solve. The requirement set is not unique – there may be other such sets.

A system with parameters may have a formal solution, e.g.

```
solve ({x=a*z+1, 0=b*z-y}, {z, x}) ;
```

```

      y      a*y + b
{ { z=---, x=----- } }
      b      b

```

¹ The difference between linear and non-linear inconsistent systems is based on the algorithms which produce this information as a side effect when attempting to find a formal solution; example: $\text{solve}(\{x = a, x = b, y = c, y = d\}, \{x, y\})$ gives a set $\{a - b, c - d\}$ while $\text{solve}(\{x^2 = a, x^2 = b, y^2 = c, y^2 = d\}, \{x, y\})$ leads to $\{a - b\}$.

which is not valid for all possible values of the parameters. The variable `ASSUMPTIONS` contains then a list of restrictions: the solutions are valid only as long as none of these expressions vanishes. Any zero of one of them represents a special case that is not covered by the formal solution. In the above case the value is

```
assumptions;

{b}
```

which excludes formally the case $b = 0$; obviously this special parameter value makes the system singular. The set of assumptions is complete for both, linear and non-linear systems.

`SOLVE` rearranges the variable sequence to reduce the (expected) computing time. This behavior is controlled by the switch `VAROPT`, which is on by default. If it is turned off, the supplied variable sequence is used or the system kernel ordering is taken if the variable list is omitted. The effect is demonstrated by an example:

```
s:= {y^3+3x=0,x^2+y^2=1};

solve(s,{y,x});

      6      2
{{y=root_of(y_  + 9*y_  - 9,y_),
      3
      - y
x=-----}}
      3

off varopt; solve(s,{y,x});

      6      4      2
{{x=root_of(x_  - 3*x_  + 12*x_  - 1,x_),
      4      2
      x*( - x  + 2*x  - 10)
y=-----}}
      3
```

In the first case, `solve` forms the solution as a set of pairs $(y_i, x(y_i))$ because the degree of x is higher – such a rearrangement makes the internal computation of the

Gröbner basis generally faster. For the second case the explicitly given variable sequence is used such that the solution has now the form $(x_i, y(x_i))$. Controlling the variable sequence is especially important if the system has one or more free variables. As an alternative to turning off `varopt`, a partial dependency among the variables can be declared using the `depend` statement: `solve` then rearranges the variable sequence but keeps any variable ahead of those on which it depends.

```

on varopt;
s:={a^3+b,b^2+c}$
solve(s,{a,b,c});

{{a=arbcomplex(1),b=- a^3 ,c= - a^6 }}

depend a,c; depend b,c; solve(s,{a,b,c});

{{c=arbcomplex(2),

a=root_of(a_6 + c,a_6),

b= - a^3 }}

```

Here `solve` is forced to put c after a and after b , but there is no obstacle to interchanging a and b .

7.16 Even and Odd Operators

An operator can be declared to be *even* or *odd* in its first argument by the declarations `EVEN` and `ODD` respectively. Expressions involving an operator declared in this manner are transformed if the first argument contains a minus sign. Any other arguments are not affected. In addition, if say F is declared odd, then $f(0)$ is replaced by zero unless F is also declared *non zero* by the declaration `NONZERO`. For example, the declarations

```
even f1; odd f2;
```

mean that

```

f1(-a)    ->    F1(A)
f2(-a)    ->    -F2(A)
f1(-a,-b) ->    F1(A,-B)

```

$$f2(0) \rightarrow 0.$$

To inhibit the last transformation, say `nonzero f2;`.

7.17 Linear Operators

An operator can be declared to be linear in its first argument over powers of its second argument. If an operator F is so declared, F of any sum is broken up into sums of F s, and any factors that are not powers of the variable are taken outside. This means that F must have (at least) two arguments. In addition, the second argument must be an identifier (or more generally a kernel), not an expression.

Example:

If F were declared linear, then

$$f(a \cdot x^5 + b \cdot x + c, x) \rightarrow F(X^5, X) * A + F(X, X) * B + F(1, X) * C$$

More precisely, not only will the variable and its powers remain within the scope of the F operator, but so will any variable and its powers that had been declared to `DEPEND` on the prescribed variable; and so would any expression that contains that variable or a dependent variable on any level, e.g. `cos(sin(x))`.

To declare operators F and G to be linear operators, use:

```
linear f,g;
```

The analysis is done of the first argument with respect to the second; any other arguments are ignored. It uses the following rules of evaluation:

$$\begin{aligned} f(0) &\rightarrow 0 \\ f(-y, x) &\rightarrow -F(Y, X) \\ f(y+z, x) &\rightarrow F(Y, X) + F(Z, X) \\ f(y \cdot z, x) &\rightarrow Z * F(Y, X) && \text{if } Z \text{ does not depend on } X \\ f(y/z, x) &\rightarrow F(Y, X) / Z && \text{if } Z \text{ does not depend on } X \end{aligned}$$

To summarize, Y “depends” on the indeterminate X in the above if either of the following hold:

1. Y is an expression that contains X at any level as a variable, e.g.: `cos(sin(x))`
2. Any variable in the expression Y has been declared dependent on X by use of the declaration `DEPEND`.

The use of such linear operators can be seen in the paper Fox, J.A. and A. C. Hearn, "Analytic Computation of Some Integrals in Fourth Order Quantum Electrodynamics" Journ. Comp. Phys. 14 (1974) 301-317, which contains a complete listing of a program for definite integration of some expressions that arise in fourth order quantum electrodynamics.

7.18 Non-Commuting Operators

An operator can be declared to be non-commutative under multiplication by the declaration `NONCOM`.

Example:

After the declaration

```
noncom u, v;
```

the expressions $u(x) * u(y) - u(y) * u(x)$ and $u(x) * v(y) - v(y) * u(x)$ will remain unchanged on simplification, and in particular will not simplify to zero.

Note that it is the operator (U and V in the above example) and not the variable that has the non-commutative property.

The `LET` statement may be used to introduce rules of evaluation for such operators. In particular, the boolean operator `ORDP` is useful for introducing an ordering on such expressions.

Example:

The rule

```
for all x,y such that x neq y and ordp(x,y)
  let u(x)*u(y) = u(y)*u(x)+comm(x,y);
```

would introduce the commutator of $u(x)$ and $u(y)$ for all x and y . Note that since `ordp(x, x)` is *true*, the equality check is necessary in the degenerate case to avoid a circular loop in the rule.

7.19 Symmetric and Antisymmetric Operators

An operator can be declared to be symmetric with respect to its arguments by the declaration `SYMMETRIC`. For example

```
symmetric u, v;
```

means that any expression involving the top level operators U or V will have its arguments reordered to conform to the internal order used by REDUCE. The user can change this order for kernels by the command KORDER.

For example, $u(x, v(1, 2))$ would become $u(v(2, 1), x)$, since numbers are ordered in decreasing order, and expressions are ordered in decreasing order of complexity.

Similarly the declaration ANTISYMMETRIC declares an operator antisymmetric. For example,

```
antisymmetric l,m;
```

means that any expression involving the top level operators L or M will have its arguments reordered to conform to the internal order of the system, and the sign of the expression changed if there are an odd number of argument interchanges necessary to bring about the new order.

For example, $l(x, m(1, 2))$ would become $-l(-m(2, 1), x)$ since one interchange occurs with each operator. An expression like $l(x, x)$ would also be replaced by 0.

7.20 Declaring New Prefix Operators

The user may add new prefix operators to the system by using the declaration OPERATOR. For example:

```
operator h,g1,arctan;
```

adds the prefix operators H , $G1$ and $ARCTAN$ to the system.

This allows symbols like $h(w)$, $h(x, y, z)$, $g1(p+q)$, $arctan(u/v)$ to be used in expressions, but no meaning or properties of the operator are implied. The same operator symbol can be used equally well as a 0-, 1-, 2-, 3-, etc.-place operator.

To give a meaning to an operator symbol, or express some of its properties, LET statements can be used, or the operator can be given a definition as a procedure.

If the user forgets to declare an identifier as an operator, the system will prompt the user to do so in interactive mode, or do it automatically in non-interactive mode. A diagnostic message will also be printed if an identifier is declared OPERATOR more than once.

Operators once declared are global in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure)

does not limit the scope of the operator to that block, nor does the operator go away on exiting the block (use `CLEAR` instead for this purpose).

7.21 Declaring New Infix Operators

Users can add new infix operators by using the declarations `INFIX` and `PRECEDENCE`. For example,

```
infix mm;
precedence mm, -;
```

The declaration `infix mm;` would allow one to use the symbol `MM` as an infix operator:

`a mm b` instead of `mm(a, b)`.

The declaration `precedence mm, -;` says that `MM` should be inserted into the infix operator precedence list just *after* the `-` operator. This gives it higher precedence than `-` and lower precedence than `*`. Thus

`a - b mm c - d` means `a - (b mm c) - d`,

while

`a * b mm c * d` means `(a * b) mm (c * d)`.

Both infix and prefix operators have no transformation properties unless `LET` statements or procedure declarations are used to assign a meaning.

We should note here that infix operators so defined are always binary:

`a mm b mm c` means `(a mm b) mm c`.

7.22 Creating/Removing Variable Dependency

There are several facilities in `REDUCE`, such as the differentiation operator and the linear operator facility, that can utilize knowledge of the dependency between various variables, or kernels. Such dependency may be expressed by the command `DEPEND`. This takes an arbitrary number of arguments and sets up a dependency of the first argument on the remaining arguments. For example,

```
depend x, y, z;
```

says that X is dependent on both Y and Z .

```
depend z, cos(x), y;
```

says that Z is dependent on $\text{COS}(X)$ and Y .

Dependencies introduced by `DEPEND` can be removed by `NODEPEND`. The arguments of this are the same as for `DEPEND`. For example, given the above dependencies,

```
nodepend z, cos(x);
```

says that Z is no longer dependent on $\text{COS}(X)$, although it remains dependent on Y .

Chapter 8

Display and Structuring of Expressions

In this section, we consider a variety of commands and operators that permit the user to obtain various parts of algebraic expressions and also display their structure in a variety of forms. Also presented are some additional concepts in the REDUCE design that help the user gain a better understanding of the structure of the system.

8.1 Kernels

REDUCE is designed so that each operator in the system has an evaluation (or simplification) function associated with it that transforms the expression into an internal canonical form. This form, which bears little resemblance to the original expression, is described in detail in Hearn, A. C., "REDUCE 2: A System and Language for Algebraic Manipulation," Proc. of the Second Symposium on Symbolic and Algebraic Manipulation, ACM, New York (1971) 128-133.

The evaluation function may transform its arguments in one of two alternative ways. First, it may convert the expression into other operators in the system, leaving no functions of the original operator for further manipulation. This is in a sense true of the evaluation functions associated with the operators $+$, $*$ and $/$, for example, because the canonical form does not include these operators explicitly. It is also true of an operator such as the determinant operator `DET` because the relevant evaluation function calculates the appropriate determinant, and the operator `DET` no longer appears. On the other hand, the evaluation process may leave some residual functions of the relevant operator. For example, with the operator `COS`, a residual expression like `COS (X)` may remain after evaluation unless a rule for the reduction of cosines into exponentials, for example, were introduced. These residual functions of an operator are termed *kernels* and are stored uniquely like

variables. Subsequently, the kernel is carried through the calculation as a variable unless transformations are introduced for the operator at a later stage.

In those cases where the evaluation process leaves an operator expression with non-trivial arguments, the form of the argument can vary depending on the state of the system at the point of evaluation. Such arguments are normally produced in expanded form with no terms factored or grouped in any way. For example, the expression `COS (2*x+2*y)` will normally be returned in the same form. If the argument `2*x+2*y` were evaluated at the top level, however, it would be printed as `2* (X+Y)`. If it is desirable to have the arguments themselves in a similar form, the switch `INTSTR` (for “internal structure”), if on, will cause this to happen.

In cases where the arguments of the kernel operators may be reordered, the system puts them in a canonical order, based on an internal intrinsic ordering of the variables. However, some commands allow arguments in the form of kernels, and the user has no way of telling what internal order the system will assign to these arguments. To resolve this difficulty, we introduce the notion of a *kernel form* as an expression that transforms to a kernel on evaluation.

Examples of kernel forms are:

```
a
cos (x*y)
log (sin (x) )
```

whereas

```
a*b
(a+b) ^ 4
```

are not.

We see that kernel forms can usually be used as generalized variables, and most algebraic properties associated with variables may also be associated with kernels.

8.2 The Expression Workspace

Several mechanisms are available for saving and retrieving previously evaluated expressions. The simplest of these refers to the last algebraic expression simplified. When an assignment of an algebraic expression is made, or an expression is evaluated at the top level, (i.e., not inside a compound statement or procedure) the results of the evaluation are automatically saved in a variable `WS` that we shall refer to as the workspace. (More precisely, the expression is assigned to the variable `WS` that is then available for further manipulation.)

Example:

If we evaluate the expression $(x+y)^2$ at the top level and next wish to differentiate it with respect to Y , we can simply say

```
df(ws,y);
```

to get the desired answer.

If the user wishes to assign the workspace to a variable or expression for later use, the `SAVEAS` statement can be used. It has the syntax

```
SAVEAS <expression>
```

For example, after the differentiation in the last example, the workspace holds the expression $2*x+2*y$. If we wish to assign this to the variable Z we can now say

```
saveas z;
```

If the user wishes to save the expression in a form that allows him to use some of its variables as arbitrary parameters, the `FOR ALL` command can be used.

Example:

```
for all x saveas h(x);
```

with the above expression would mean that $h(z)$ evaluates to $2*Y+2*Z$.

A further method for referencing more than the last expression is described in chapter 13 on interactive use of REDUCE.

8.3 Output of Expressions

A considerable degree of flexibility is available in REDUCE in the printing of expressions generated during calculations. No explicit format statements are supplied, as these are in most cases of little use in algebraic calculations, where the size of output or its composition is not generally known in advance. Instead, REDUCE provides a series of mode options to the user that should enable him to produce his output in a comprehensible and possibly pleasing form.

The most extreme option offered is to suppress the output entirely from any top level evaluation. This is accomplished by turning off the switch `OUTPUT` which is normally on. It is useful for limiting output when loading large files or producing “clean” output from the prettyprint programs.

In most circumstances, however, we wish to view the output, so we need to know

how to format it appropriately. As we mentioned earlier, an algebraic expression is normally printed in an expanded form, filling the whole output line with terms. Certain output declarations, however, can be used to affect this format. To begin with, we look at an operator for changing the length of the output line.

8.3.1 LINELENGTH Operator

This operator is used with the syntax

```
LINELENGTH (NUM:integer) :integer
```

and sets the output line length to the integer NUM. It returns the previous output line length (so that it can be stored for later resetting of the output line if needed).

8.3.2 Output Declarations

We now describe a number of switches and declarations that are available for controlling output formats. It should be noted, however, that the transformation of large expressions to produce these varied output formats can take a lot of computing time and space. If a user wishes to speed up the printing of the output in such cases, he can turn off the switch PRI. If this is done, then output is produced in one fixed format, which basically reflects the internal form of the expression, and none of the options below apply. PRI is normally on.

With PRI on, the output declarations and switches available are as follows:

ORDER Declaration

The declaration ORDER may be used to order variables on output. The syntax is:

```
order v1, ... vn;
```

where the v_i are kernels. Thus,

```
order x, y, z;
```

orders X ahead of Y, Y ahead of Z and all three ahead of other variables not given an order. `order nil;` resets the output order to the system default. The order of variables may be changed by further calls of ORDER, but then the reordered variables would have an order lower than those in earlier ORDER calls. Thus,

```
order x, y, z;
order y, x;
```

would order Z ahead of Y and X . The default ordering is usually alphabetic.

FACTOR Declaration

This declaration takes a list of identifiers or kernels as argument. `FACTOR` is not a factoring command (use `FACTORIZE` or the `FACTOR` switch for this purpose); rather it is a separation command. All terms involving fixed powers of the declared expressions are printed as a product of the fixed powers and a sum of the rest of the terms.

All expressions involving a given prefix operator may also be factored by putting the operator name in the list of factored identifiers. For example:

```
factor x,cos,sin(x);
```

causes all powers of X and $\sin(X)$ and all functions of \cos to be factored.

Note that `FACTOR` does not affect the order of its arguments. You should also use `ORDER` if this is important.

The declaration `remfac v1, ..., vn;` removes the factoring flag from the expressions v_1 through v_n .

8.3.3 Output Control Switches

In addition to these declarations, the form of the output can be modified by switching various output control switches using the declarations `ON` and `OFF`. We shall illustrate the use of these switches by an example, namely the printing of the expression

$$x^2 * (y^2 + 2 * y) + x * (y^2 + z) / (2 * a) \quad .$$

The relevant switches are as follows:

ALLFAC Switch

This switch will cause the system to search the whole expression, or any sub-expression enclosed in parentheses, for simple multiplicative factors and print them outside the parentheses. Thus our expression with `ALLFAC` off will print as

$$(2 * X^2 * Y^2 * A^2 + 4 * X^2 * Y * A^2 + X * Y^2 + X * Z) / (2 * A)$$

and with `ALLFAC` on as

$$X * (2 * X * Y^2 * A + 4 * X * Y * A^2 + Y^2 + Z) / (2 * A) \quad .$$

ALLFAC is normally on, and is on in the following examples, except where otherwise stated.

DIV Switch

This switch makes the system search the denominator of an expression for simple factors that it divides into the numerator, so that rational fractions and negative powers appear in the output. With DIV on, our expression would print as

$$X * (X * Y^2 + 2 * X * Y + 1/2 * Y^2 * A^{(-1)} + 1/2 * A^{(-1)} * Z) \quad .$$

DIV is normally off.

LIST Switch

This switch causes the system to print each term in any sum on a separate line. With LIST on, our expression prints as

$$\begin{aligned} &X * (2 * X * Y^2 * A \\ &+ 4 * X * Y * A^2 \\ &+ Y^2 \\ &+ Z) / (2 * A) \quad . \end{aligned}$$

LIST is normally off.

NOSPLIT Switch

Under normal circumstances, the printing routines try to break an expression across lines at a natural point. This is a fairly expensive process. If you are not overly concerned about where the end-of-line breaks come, you can speed up the printing of expressions by turning off the switch NOSPLIT. This switch is normally on.

RAT Switch

This switch is only useful with expressions in which variables are factored with FACTOR. With this mode, the overall denominator of the expression is printed with each factored sub-expression. We assume a prior declaration `factor x`; in the following output. We first print the expression with RAT set to off:

$$(2 * X^2 * Y * A * (Y^2 + 2) + X * (Y^2 + Z)) / (2 * A) .$$

With RAT on the output becomes:

$$X^2 * Y * (Y^2 + 2) + X * (Y^2 + Z) / (2 * A) .$$

RAT is normally off.

Next, if we leave X factored, and turn on both DIV and RAT, the result becomes

$$X^2 * Y * (Y^2 + 2) + 1/2 * X * A^{(-1)} * (Y^2 + Z) .$$

Finally, with X factored, RAT on and ALLFAC off we retrieve the original structure

$$X^2 * (Y^2 + 2 * Y) + X * (Y^2 + Z) / (2 * A) .$$

RATPRI Switch

If the numerator and denominator of an expression can each be printed in one line, the output routines will print them in a two dimensional notation, with numerator and denominator on separate lines and a line of dashes in between. For example, $(a+b)/2$ will print as

$$\begin{array}{c} A + B \\ \hline 2 \end{array}$$

Turning this switch off causes such expressions to be output in a linear form.

REVPRI Switch

The normal ordering of terms in output is from highest to lowest power. In some situations (e.g., when a power series is output), the opposite ordering is more con-

venient. The switch `REVPRI` if on causes such a reverse ordering of terms. For example, the expression $y * (x+1)^2 + (y+3)^2$ will normally print as

$$x^2 * y + 2 * x * y + y^2 + 7 * y + 9$$

whereas with `REVPRI` on, it will print as

$$9 + 7 * y + y^2 + 2 * x * y + x^2 * y.$$

8.3.4 WRITE Command

In simple cases no explicit output command is necessary in `REDUCE`, since the value of any expression is automatically printed if a semicolon is used as a delimiter. There are, however, several situations in which such a command is useful.

In a `FOR`, `WHILE`, or `REPEAT` statement it may be desired to output something each time the statement within the loop construct is repeated.

It may be desired for a procedure to output intermediate results or other information while it is running. It may be desired to have results labeled in special ways, especially if the output is directed to a file or device other than the terminal.

The `WRITE` command consists of the word `WRITE` followed by one or more items separated by commas, and followed by a terminator. There are three kinds of items that can be used:

1. Expressions (including variables and constants). The expression is evaluated, and the result is printed out.
2. Assignments. The expression on the right side of the `:=` operator is evaluated, and is assigned to the variable on the left; then the symbol on the left is printed, followed by a `“:=”`, followed by the value of the expression on the right – almost exactly the way an assignment followed by a semicolon prints out normally. (The difference is that if the `WRITE` is in a `FOR` statement and the left-hand side of the assignment is an array position or something similar containing the variable of the `FOR` iteration, then the value of that variable is inserted in the printout.)
3. Arbitrary strings of characters, preceded and followed by double-quote marks (e.g., `"string"`).

The items specified by a single `WRITE` statement print side by side on one line. (The line is broken automatically if it is too long.) Strings print exactly as quoted. The `WRITE` command itself however does not return a value.

The print line is closed at the end of a `WRITE` command evaluation. Therefore the command `WRITE " "`; (specifying nothing to be printed except the empty string) causes a line to be skipped.

Examples:

1. If `A` is `X+5`, `B` is itself, `C` is 123, `M` is an array, and `Q=3`, then

```
write m(q):=a, " ",b/c, " THANK YOU";
```

will set `M(3)` to `x+5` and print

```
M(Q) := X + 5 B/123 THANK YOU
```

The blanks between the 5 and B, and the 3 and T, come from the blanks in the quoted strings.

2. To print a table of the squares of the integers from 1 to 20:

```
for i:=1:20 do write i, " ",i^2;
```

3. To print a table of the squares of the integers from 1 to 20, and at the same time store them in positions 1 to 20 of an array `A`:

```
for i:=1:20 do <<a(i):=i^2; write i, " ",a(i)>>;
```

This will give us two columns of numbers. If we had used

```
for i:=1:20 do write i, " ",a(i):=i^2;
```

we would also get `A(i) :=` repeated on each line.

4. The following more complete example calculates the famous `f` and `g` series, first reported in Sconzo, P., LeSchack, A. R., and Tobey, R., "Symbolic Computation of `f` and `g` Series by Computer", *Astronomical Journal* 70 (May 1965).

```
x1:= -sig*(mu+2*eps)$
x2:= eps - 2*sig^2$
x3:= -3*mu*sig$
f:= 1$
g:= 0$
for i:= 1 step 1 until 10 do begin
  f1:= -mu*g+x1*df(f,eps)+x2*df(f,sig)+x3*df(f,mu);
  write "f(",i,") := ",f1;
  g1:= f+x1*df(g,eps)+x2*df(g,sig)+x3*df(g,mu);
```

```

write "g(",i,") := ",g1;
f:=f1$
g:=g1$
end;

```

A portion of the output, to illustrate the printout from the WRITE command, is as follows:

```

... <prior output> ...

                                2
F(4) := MU*(3*EPS - 15*SIG  + MU)

G(4) := 6*SIG*MU

                                2
F(5) := 15*SIG*MU*( - 3*EPS + 7*SIG  - MU)

                                2
G(5) := MU*(9*EPS - 45*SIG  + MU)

... <more output> ...

```

8.3.5 Suppression of Zeros

It is sometimes annoying to have zero assignments (i.e. assignments of the form `<expression> := 0`) printed, especially in printing large arrays with many zero elements. The output from such assignments can be suppressed by turning on the switch `NERO`.

8.3.6 FORTRAN Style Output Of Expressions

It is naturally possible to evaluate expressions numerically in REDUCE by giving all variables and sub-expressions numerical values. However, as we pointed out elsewhere the user must declare real arithmetical operation by turning on the switch `ROUNDED`. However, it should be remembered that arithmetic in REDUCE is not particularly fast, since results are interpreted rather than evaluated in a compiled form. The user with a large amount of numerical computation after all necessary algebraic manipulations have been performed is therefore well advised to perform these calculations in a FORTRAN or similar system. For this purpose, REDUCE

offers facilities for users to produce FORTRAN compatible files for numerical processing.

First, when the switch `FOR` is on, the system will print expressions in a FORTRAN notation. Expressions begin in column seven. If an expression extends over one line, a continuation mark (.) followed by a blank appears on subsequent cards. After a certain number of lines have been produced (according to the value of the variable `CARD_NO`), a new expression is started. If the expression printed arises from an assignment to a variable, the variable is printed as the name of the expression. Otherwise the expression is given the default name `ANS`. An error occurs if identifiers or numbers are outside the bounds permitted by FORTRAN.

A second option is to use the `WRITE` command to produce other programs.

Example:

The following `REDUCE` statements

```
on fort;
out "forfil";
write "C      this is a fortran program";
write " 1      format (e13.5) ";
write "      u=1.23";
write "      v=2.17";
write "      w=5.2";
x:=(u+v+w)^11;
write "C      it was foolish to expand this expression";
write "      print 1,x";
write "      end";
shut "forfil";
off fort;
```

will generate a file `forfil` that contains:

```
c this is a fortran program
1      format (e13.5)
      u=1.23
      v=2.17
      w=5.2
      ans1=1320.*u**3*v**w**7+165.*u**3*w**8+55.*u**2*v**9+495.*u
. **2*v**8*w+1980.*u**2*v**7*w**2+4620.*u**2*v**6*w**3+
. 6930.*u**2*v**5*w**4+6930.*u**2*v**4*w**5+4620.*u**2*v**3*
. w**6+1980.*u**2*v**2*w**7+495.*u**2*v*w**8+55.*u**2*w**9+
. 11.*u*v**10+110.*u*v**9*w+495.*u*v**8*w**2+1320.*u*v**7*w
. **3+2310.*u*v**6*w**4+2772.*u*v**5*w**5+2310.*u*v**4*w**6
. +1320.*u*v**3*w**7+495.*u*v**2*w**8+110.*u*v*w**9+11.*u*w
. **10+v**11+11.*v**10*w+55.*v**9*w**2+165.*v**8*w**3+330.*
```

```

. v**7*w**4+462.*v**6*w**5+462.*v**5*w**6+330.*v**4*w**7+
. 165.*v**3*w**8+55.*v**2*w**9+11.*v*w**10+w**11
x=u**11+11.*u**10*v+11.*u**10*w+55.*u**9*v**2+110.*u**9*v*
. w+55.*u**9*w**2+165.*u**8*v**3+495.*u**8*v**2*w+495.*u**8
. *v*w**2+165.*u**8*w**3+330.*u**7*v**4+1320.*u**7*v**3*w+
. 1980.*u**7*v**2*w**2+1320.*u**7*v*w**3+330.*u**7*w**4+462.
. *u**6*v**5+2310.*u**6*v**4*w+4620.*u**6*v**3*w**2+4620.*u
. **6*v**2*w**3+2310.*u**6*v*w**4+462.*u**6*w**5+462.*u**5*
. v**6+2772.*u**5*v**5*w+6930.*u**5*v**4*w**2+9240.*u**5*v
. **3*w**3+6930.*u**5*v**2*w**4+2772.*u**5*v*w**5+462.*u**5
. *w**6+330.*u**4*v**7+2310.*u**4*v**6*w+6930.*u**4*v**5*w
. **2+11550.*u**4*v**4*w**3+11550.*u**4*v**3*w**4+6930.*u**
. 4*v**2*w**5+2310.*u**4*v*w**6+330.*u**4*w**7+165.*u**3*v
. **8+1320.*u**3*v**7*w+4620.*u**3*v**6*w**2+9240.*u**3*v**
. 5*w**3+11550.*u**3*v**4*w**4+9240.*u**3*v**3*w**5+4620.*u
. **3*v**2*w**6+ans1
c      it was foolish to expand this expression
      print 1,x
      end

```

If the arguments of a `WRITE` statement include an expression that requires continuation records, the output will need editing, since the output routine prints the arguments of `WRITE` sequentially, and the continuation mechanism therefore generates its auxiliary variables after the preceding expression has been printed.

Finally, since there is no direct analog of *list* in FORTRAN, a comment line of the form

```
c ***** invalid fortran construct (list) not printed
```

will be printed if you try to print a list with `FORT` on.

FORTRAN Output Options

There are a number of methods available to change the default format of the FORTRAN output.

The breakup of the expression into subparts is such that the number of continuation lines produced is less than a given number. This number can be modified by the assignment

```
card_no := <number>;
```

where *<number>* is the *total* number of cards allowed in a statement. The default value of `CARD_NO` is 20.

The width of the output expression is also adjustable by the assignment

```
fort_width := <integer>;
```

FORT_WIDTH which sets the total width of a given line to *<integer>*. The initial FORTRAN output width is 70.

REDUCE automatically inserts a decimal point after each isolated integer coefficient in a FORTRAN expression (so that, for example, 4 becomes 4 .). To prevent this, set the PERIOD mode switch to OFF.

FORTRAN output is normally produced in lower case. If upper case is desired, the switch FORTUPPER should be turned on.

Finally, the default name ANS assigned to an unnamed expression and its subparts can be changed by the operator VARNAME. This takes a single identifier as argument, which then replaces ANS as the expression name. The value of VARNAME is its argument.

Further facilities for the production of FORTRAN and other language output are provided by the SCOPE and GENTRAN packages described in chapters [16.24](#) and [16.52](#).

8.3.7 Saving Expressions for Later Use as Input

It is often useful to save an expression on an external file for use later as input in further calculations. The commands for opening and closing output files are explained elsewhere. However, we see in the examples on output of expressions that the standard “natural” method of printing expressions is not compatible with the input syntax. So to print the expression in an input compatible form we must inhibit this natural style by turning off the switch NAT. If this is done, a dollar sign will also be printed at the end of the expression.

Example:

The following sequence of commands

```
off nat; out "out"; x := (y+z)^2; write "end";
shut "out"; on nat;
```

will generate a file out that contains

```
X := Y**2 + 2*Y*Z + Z**2$
END$
```

8.3.8 Displaying Expression Structure

In those cases where the final result has a complicated form, it is often convenient to display the skeletal structure of the answer. The operator `STRUCTR`, that takes a single expression as argument, will do this for you. Its syntax is:

```
STRUCTR(EXPRN:algebraic[,ID1:identifier[,ID2:identifier]]);
```

The structure is printed effectively as a tree, in which the subparts are laid out with auxiliary names. If the optional `ID1` is absent, the auxiliary names are prefixed by the root `ANS`. This root may be changed by the operator `VARNAME`. If the optional `ID1` is present, and is an array name, the subparts are named as elements of that array, otherwise `ID1` is used as the root prefix. (The second optional argument `ID2` is explained later.)

The `EXPRN` can be either a scalar or a matrix expression. Use of any other will result in an error.

Example:

Let us suppose that the workspace contains $((A+B)^2+C)^3+D$. Then the input `STRUCTR WS;` will (with `EXP` off) result in the output:

ANS3

where

$$\text{ANS3} := \text{ANS2}^3 + D$$

$$\text{ANS2} := \text{ANS1}^2 + C$$

$$\text{ANS1} := A + B$$

The workspace remains unchanged after this operation, since `STRUCTR` in the default situation returns no value (if `STRUCTR` is used as a sub-expression, its value is taken to be 0). In addition, the sub-expressions are normally only displayed and not retained. If you wish to access the sub-expressions with their displayed names, the switch `SAVESTRUCTR` should be turned on. In this case, `STRUCTR` returns a list whose first element is a representation for the expression, and subsequent elements are the sub-expression relations. Thus, with `SAVESTRUCTR` on, `STRUCTR WS` in the above example would return

$$\{\text{ANS3}, \text{ANS3}=\text{ANS2}^3 + D, \text{ANS2}=\text{ANS1}^2 + C, \text{ANS1}=A + B\}$$

The `PART` operator can be used to retrieve the required parts of the expression. For example, to get the value of `ANS2` in the above, one could say:

```
part (ws, 3, 2) ;
```

If `FORT` is on, then the results are printed in the reverse order; the algorithm in fact guaranteeing that no sub-expression will be referenced before it is defined. The second optional argument `ID2` may also be used in this case to name the actual expression (or expressions in the case of a matrix argument).

Example:

Let us suppose that `M`, a 2 by 1 matrix, contains the elements $(a+b)^2 + c^3 + d$ and $(a + b) * (c + d)$ respectively, and that `V` has been declared to be an array. With `EXP` off and `FORT` on, the statement `structr(2*m, v, k) ;` will result in the output

```
V(1)=A+B
V(2)=V(1)**2+C
V(3)=V(2)**3+D
V(4)=C+D
K(1,1)=2.*V(3)
K(2,1)=2.*V(1)*V(4)
```

8.4 Changing the Internal Order of Variables

The internal ordering of variables (more specifically kernels) can have a significant effect on the space and time associated with a calculation. In its default state, `REDUCE` uses a specific order for this which may vary between sessions. However, it is possible for the user to change this internal order by means of the declaration `KORDER`. The syntax for this is:

```
korder v1, ..., vn;
```

where the `Vi` are kernels. With this declaration, the `Vi` are ordered internally ahead of any other kernels in the system. `V1` has the highest order, `V2` the next highest, and so on. A further call of `KORDER` replaces a previous one. `KORDER NIL;` resets the internal order to the system default.

Unlike the `ORDER` declaration, that has a purely cosmetic effect on the way results are printed, the use of `KORDER` can have a significant effect on computation time. In critical cases then, the user can experiment with the ordering of the variables used to determine the optimum set for a given problem.

8.5 Obtaining Parts of Algebraic Expressions

There are many occasions where it is desirable to obtain a specific part of an expression, or even change such a part to another expression. A number of operators are available in REDUCE for this purpose, and will be described in this section. In addition, operators for obtaining specific parts of polynomials and rational functions (such as a denominator) are described in another section.

8.5.1 COEFF Operator

Syntax:

`COEFF (EXPRN:polynomial, VAR:kernel)`

COEFF is an operator that partitions EXPRN into its various coefficients with respect to VAR and returns them as a list, with the coefficient independent of VAR first.

Under normal circumstances, an error results if EXPRN is not a polynomial in VAR, although the coefficients themselves can be rational as long as they do not depend on VAR. However, if the switch RATARG is on, denominators are not checked for dependence on VAR, and are taken to be part of the coefficients.

Example:

`coeff ((y^2+z)^3/z, y);`

returns the result

$\{z^2, 0, 3z, 0, 3, 0, 1/z\}.$

whereas

`coeff ((y^2+z)^3/y, y);`

gives an error if RATARG is off, and the result

$\{z^3/Y, 0, 3z^2/Y, 0, 3z/Y, 0, 1/Y\}$

if RATARG is on.

The length of the result of COEFF is the highest power of VAR encountered plus 1. In the above examples it is 7. In addition, the variable HIGHPOW is set to

the highest non-zero power found in `EXPRN` during the evaluation, and `LOW_POW` to the lowest non-zero power, or zero if there is a constant term. If `EXPRN` is a constant, then `HIGH_POW` and `LOW_POW` are both set to zero.

8.5.2 COEFFN Operator

The `COEFFN` operator is designed to give the user a particular coefficient of a variable in a polynomial, as opposed to `COEFF` that returns all coefficients. `COEFFN` is used with the syntax

```
COEFFN (EXPRN:polynomial, VAR:kernel, N:integer)
```

It returns the n^{th} coefficient of `VAR` in the polynomial `EXPRN`.

8.5.3 PART Operator

Syntax:

```
PART (EXPRN:algebraic[, INTEXP:integer])
```

This operator works on the form of the expression as printed *or as it would have been printed at that point in the calculation* bearing in mind all the relevant switch settings at that point. The reader therefore needs some familiarity with the way that expressions are represented in prefix form in `REDUCE` to use these operators effectively. Furthermore, it is assumed that `PRI` is `ON` at that point in the calculation. The reason for this is that with `PRI` off, an expression is printed by walking the tree representing the expression internally. To save space, it is never actually transformed into the equivalent prefix expression as occurs when `PRI` is on. However, the operations on polynomials described elsewhere can be equally well used in this case to obtain the relevant parts.

The evaluation proceeds recursively down the integer expression list. In other words,

$$\begin{aligned} & \text{PART}(\langle \text{expression} \rangle, \langle \text{integer1} \rangle, \langle \text{integer2} \rangle) \\ & \rightarrow \text{PART}(\text{PART}(\langle \text{expression} \rangle, \langle \text{integer1} \rangle), \langle \text{integer2} \rangle) \end{aligned}$$

and so on, and

$$\text{PART}(\langle \text{expression} \rangle) \rightarrow \langle \text{expression} \rangle.$$

`INTEXP` can be any expression that evaluates to an integer. If the integer is positive, then that term of the expression is found. If the integer is 0, the operator

is returned. Finally, if the integer is negative, the counting is from the tail of the expression rather than the head.

For example, if the expression $a+b$ is printed as $A+B$ (i.e., the ordering of the variables is alphabetical), then

```

part(a+b, 2)  ->  B
part(a+b, -1) ->  B
and
part(a+b, 0)  ->  PLUS

```

An operator `ARGLength` is available to determine the number of arguments of the top level operator in an expression. If the expression does not contain a top level operator, then -1 is returned. For example,

```

arglength(a+b+c) -> 3
arglength(f())   -> 0
arglength(a)     -> -1

```

8.5.4 Substituting for Parts of Expressions

`PART` may also be used to substitute for a given part of an expression. In this case, the `PART` construct appears on the left-hand side of an assignment statement, and the expression to replace the given part on the right-hand side.

For example, with the normal settings of the `REDUCE` switches:

```

xx := a+b;
part(xx, 2) := c;    ->  A+C
part(c+d, 0) := -;   ->  C-D

```

Note that `xx` in the above is not changed by this substitution. In addition, unlike expressions such as array and matrix elements that have an *instant evaluation* property, the values of `part(xx, 2)` and `part(c+d, 0)` are also not changed.

Chapter 9

Polynomials and Rationals

Many operations in computer algebra are concerned with polynomials and rational functions. In this section, we review some of the switches and operators available for this purpose. These are in addition to those that work on general expressions (such as `DF` and `INT`) described elsewhere. In the case of operators, the arguments are first simplified before the operations are applied. In addition, they operate only on arguments of prescribed types, and produce a type mismatch error if given arguments which cannot be interpreted in the required mode with the current switch settings. For example, if an argument is required to be a kernel and $a/2$ is used (with no other rules for A), an error

```
A/2 invalid as kernel
```

will result.

With the exception of those that select various parts of a polynomial or rational function, these operations have potentially significant effects on the space and time associated with a given calculation. The user should therefore experiment with their use in a given calculation in order to determine the optimum set for a given problem.

One such operation provided by the system is an operator `LENGTH` which returns the number of top level terms in the numerator of its argument. For example,

```
length ((a+b+c)^3/(c+d));
```

has the value 10. To get the number of terms in the denominator, one would first select the denominator by the operator `DEN` and then call `LENGTH`, as in

```
length den ((a+b+c)^3/(c+d));
```

Other operations currently supported, the relevant switches and operators, and the required argument and value modes of the latter, follow.

9.1 Controlling the Expansion of Expressions

The switch `EXP` controls the expansion of expressions. If it is off, no expansion of powers or products of expressions occurs. Users should note however that in this case results come out in a normal but not necessarily canonical form. This means that zero expressions simplify to zero, but that two equivalent expressions need not necessarily simplify to the same form.

Example: With `EXP` on, the two expressions

$$(a+b) * (a+2*b)$$

and

$$a^2+3*a*b+2*b^2$$

will both simplify to the latter form. With `EXP` off, they would remain unchanged, unless the complete factoring (`ALLFAC`) option were in force. `EXP` is normally on.

Several operators that expect a polynomial as an argument behave differently when `EXP` is off, since there is often only one term at the top level. For example, with `EXP` off

$$\text{length}((a+b+c)^3/(c+d));$$

returns the value 1.

9.2 Factorization of Polynomials

`REDUCE` is capable of factorizing univariate and multivariate polynomials that have integer coefficients, finding all factors that also have integer coefficients. The package for doing this was written by Dr. Arthur C. Norman and Ms. P. Mary Ann Moore at The University of Cambridge. It is described in P. M. A. Moore and A. C. Norman, "Implementing a Polynomial Factorization and GCD Package", Proc. SYMSAC '81, ACM (New York) (1981), 109-116.

The easiest way to use this facility is to turn on the switch `FACTOR`, which causes all expressions to be output in a factored form. For example, with `FACTOR` on, the expression A^2-B^2 is returned as $(A+B) * (A-B)$.

It is also possible to factorize a given expression explicitly. The operator

FACTORIZE that invokes this facility is used with the syntax

```
FACTORIZE (EXPRN:polynomial[,INTEXP:prime integer]):list,
```

the optional argument of which will be described later. Thus to find and display all factors of the cyclotomic polynomial $x^{105} - 1$, one could write:

```
factorize(x^105-1);
```

The result is a list of factor,exponent pairs. In the above example, there is no overall numerical factor in the result, so the results will consist only of polynomials in x . The number of such polynomials can be found by using the operator LENGTH. If there is a numerical factor, as in factorizing $12x^2 - 12$, that factor will appear as the first member of the result. It will however not be factored further. Prime factors of such numbers can be found, using a probabilistic algorithm, by turning on the switch IFACTOR. For example,

```
on ifactor; factorize(12x^2-12);
```

would result in the output

```
{{2,2},{3,1},{X + 1,1},{X - 1,1}}.
```

If the first argument of FACTORIZE is an integer, it will be decomposed into its prime components, whether or not IFACTOR is on.

Note that the IFACTOR switch only affects the result of FACTORIZE. It has no effect if the FACTOR switch is also on.

The order in which the factors occur in the result (with the exception of a possible overall numerical coefficient which comes first) can be system dependent and should not be relied on. Similarly it should be noted that any pair of individual factors can be negated without altering their product, and that REDUCE may sometimes do that.

The factorizer works by first reducing multivariate problems to univariate ones and then solving the univariate ones modulo small primes. It normally selects both evaluation points and primes using a random number generator that should lead to different detailed behavior each time any particular problem is tackled. If, for some reason, it is known that a certain (probably univariate) factorization can be performed effectively with a known prime, P say, this value of P can be handed to FACTORIZE as a second argument. An error will occur if a non-prime is provided to FACTORIZE in this manner. It is also an error to specify a prime that divides the discriminant of the polynomial being factored, but users should note that this condition is not checked by the program, so this capability should be used with care.

Factorization can be performed over a number of polynomial coefficient domains in addition to integers. The particular description of the relevant domain should be consulted to see if factorization is supported. For example, the following statements will factorize $x^4 + 1$ modulo 7:

```
setmod 7;
on modular;
factorize(x^4+1);
```

The factorization module is provided with a trace facility that may be useful as a way of monitoring progress on large problems, and of satisfying curiosity about the internal workings of the package. The most simple use of this is enabled by issuing the REDUCE command `on trfac;`. Following this, all calls to the factorizer will generate informative messages reporting on such things as the reduction of multivariate to univariate cases, the choice of a prime and the reconstruction of full factors from their images. Further levels of detail in the trace are intended mainly for system tuners and for the investigation of suspected bugs. For example, `TRALLFAC` gives tracing information at all levels of detail. The switch that can be set by `on timings;` makes it possible for one who is familiar with the algorithms used to determine what part of the factorization code is consuming the most resources. `on overview;` reduces the amount of detail presented in other forms of trace. Other forms of trace output are enabled by directives of the form

```
symbolic set!-trace!-factor(<number>,<filename>);
```

where useful numbers are 1, 2, 3 and 100, 101, This facility is intended to make it possible to discover in fairly great detail what just some small part of the code has been doing — the numbers refer mainly to depths of recursion when the factorizer calls itself, and to the split between its work forming and factorizing images and reconstructing full factors from these. If `NIL` is used in place of a filename the trace output requested is directed to the standard output stream. After use of this trace facility the generated trace files should be closed by calling

```
symbolic close!-trace!-files();
```

NOTE: Using the factorizer with `MCD` off will result in an error.

9.3 Cancellation of Common Factors

Facilities are available in REDUCE for cancelling common factors in the numerators and denominators of expressions, at the option of the user. The system will perform this greatest common divisor computation if the switch `GCD` is on. (`GCD` is normally off.)

A check is automatically made, however, for common variable and numerical products in the numerators and denominators of expressions, and the appropriate cancellations made.

When GCD is on, and EXP is off, a check is made for square free factors in an expression. This includes separating out and independently checking the content of a given polynomial where appropriate. (For an explanation of these terms, see Anthony C. Hearn, “Non-Modular Computation of Polynomial GCDs Using Trial Division”, Proc. EUROSAM 79, published as Lecture Notes on Comp. Science, Springer-Verlag, Berlin, No 72 (1979) 227-239.)

Example: With EXP off and GCD on, the polynomial $a*c+a*d+b*c+b*d$ would be returned as $(A+B) * (C+D)$.

Under normal circumstances, GCDs are computed using an algorithm described in the above paper. It is also possible in REDUCE to compute GCDs using an alternative algorithm, called the EZGCD Algorithm, which uses modular arithmetic. The switch EZGCD, if on in addition to GCD, makes this happen.

In non-trivial cases, the EZGCD algorithm is almost always better than the basic algorithm, often by orders of magnitude. We therefore *strongly* advise users to use the EZGCD switch where they have the resources available for supporting the package.

For a description of the EZGCD algorithm, see J. Moses and D.Y.Y. Yun, “The EZ GCD Algorithm”, Proc. ACM 1973, ACM, New York (1973) 159-166.

NOTE: This package shares code with the factorizer, so a certain amount of trace information can be produced using the factorizer trace switches.

9.3.1 Determining the GCD of Two Polynomials

This operator, used with the syntax

```
GCD (EXPRN1:polynomial, EXPRN2:polynomial):polynomial,
```

returns the greatest common divisor of the two polynomials EXPRN1 and EXPRN2.

Examples:

```
gcd (x^2+2*x+1, x^2+3*x+2) -> X+1
gcd (2*x^2-2*y^2, 4*x+4*y) -> 2*X+2*Y
gcd (x^2+y^2, x-y)         -> 1.
```

9.4 Working with Least Common Multiples

Greatest common divisor calculations can often become expensive if extensive work with large rational expressions is required. However, in many cases, the only significant cancellations arise from the fact that there are often common factors in the various denominators which are combined when two rationals are added. Since these denominators tend to be smaller and more regular in structure than the numerators, considerable savings in both time and space can occur if a full GCD check is made when the denominators are combined and only a partial check when numerators are constructed. In other words, the true least common multiple of the denominators is computed at each step. The switch LCM is available for this purpose, and is normally on.

In addition, the operator LCM, used with the syntax

```
LCM(EXPRN1:polynomial,EXPRN2:polynomial):polynomial,
```

returns the least common multiple of the two polynomials EXPRN1 and EXPRN2.

Examples:

```
lcm(x^2+2*x+1,x^2+3*x+2) -> X**3 + 4*X**2 + 5*X + 2
lcm(2*x^2-2*y^2,4*x+4*y) -> 4*(X**2 - Y**2)
```

9.5 Controlling Use of Common Denominators

When two rational functions are added, REDUCE normally produces an expression over a common denominator. However, if the user does not want denominators combined, he or she can turn off the switch MCD which controls this process. The latter switch is particularly useful if no greatest common divisor calculations are desired, or excessive differentiation of rational functions is required.

CAUTION: With MCD off, results are not guaranteed to come out in either normal or canonical form. In other words, an expression equivalent to zero may in fact not be simplified to zero. This option is therefore most useful for avoiding expression swell during intermediate parts of a calculation.

MCD is normally on.

9.6 REMAINDER Operator

This operator is used with the syntax


```
REMAINDER(EXPRN1:polynomial,EXPRN2:polynomial):polynomial.
```

It returns the remainder when EXPRN1 is divided by EXPRN2. This is the true remainder based on the internal ordering of the variables, and not the pseudo-remainder. The pseudo-remainder and in general pseudo-division of polynomials can be calculated after loading the `polydiv` package. Please refer to the documentation of this package for details.

Examples:

```
remainder((x+y)*(x+2*y),x+3*y) -> 2*Y**2
remainder(2*x+y,2) -> Y.
```

CAUTION: In the default case, remainders are calculated over the integers. If you need the remainder with respect to another domain, it must be declared explicitly.

Example:

```
remainder(x^2-2,x+sqrt(2)); -> X^2 - 2
load_package arnum;
defpoly sqrt2**2-2;
remainder(x^2-2,x+sqrt2); -> 0
```

9.7 RESULTANT Operator

This is used with the syntax

```
RESULTANT(EXPRN1:polynomial,EXPRN2:polynomial,VAR:kernel):
polynomial.
```

It computes the resultant of the two given polynomials with respect to the given variable, the coefficients of the polynomials can be taken from any domain. The result can be identified as the determinant of a Sylvester matrix, but can often also be thought of informally as the result obtained when the given variable is eliminated between the two input polynomials. If the two input polynomials have a non-trivial GCD their resultant vanishes.

The switch `BEZOUT` controls the computation of the resultants. It is off by default. In this case a subresultant algorithm is used. If the switch `Bezout` is turned on, the resultant is computed via the Bezout Matrix. However, in the latter case, only polynomial coefficients are permitted.

The sign conventions used by the resultant function follow those in R. Loos, “Computing in Algebraic Extensions” in “Computer Algebra — Symbolic and Algebraic Computation”, Second Ed., Edited by B. Buchberger, G.E. Collins and R. Loos, Springer-Verlag, 1983. Namely, with A and B not dependent on X:

$$\text{resultant}(p(x), q(x), x) = (-1)^{\deg(p) \cdot \deg(q)} \cdot \text{resultant}(q, p, x)$$

$$\text{resultant}(a, p(x), x) = a^{\deg(p)}$$

$$\text{resultant}(a, b, x) = 1$$

Examples:

$$\text{resultant}(x/r*u+y, u*y, u) \rightarrow -y^2$$

calculation in an algebraic extension:

```
load arnum;
defpoly sqrt2**2 - 2;

resultant(x + sqrt2, sqrt2 * x + 1, x) -> -1
```

or in a modular domain:

```
setmod 17;
on modular;

resultant(2x+1, 3x+4, x) -> 5
```

9.8 DECOMPOSE Operator

The DECOMPOSE operator takes a multivariate polynomial as argument, and returns an expression and a list of equations from which the original polynomial can be found by composition. Its syntax is:

```
DECOMPOSE (EXPRN:polynomial) :list.
```

For example:

```

decompose (x^8-88*x^7+2924*x^6-43912*x^5+263431*x^4-
          218900*x^3+65690*x^2-7700*x+234)
          2          2          2
-> {U  + 35*U + 234, U=V  + 10*V, V=X  - 22*X}
          2
decompose (u^2+v^2+2u*v+1) -> {W  + 1, W=U  + V}

```

Users should note however that, unlike factorization, this decomposition is not unique.

9.9 INTERPOL operator

Syntax:

```
INTERPOL (<values>, <variable>, metapoints) ;
```

where $\langle values \rangle$ and $\langle points \rangle$ are lists of equal length and $\langle variable \rangle$ is an algebraic expression (preferably a kernel).

INTERPOL generates an interpolation polynomial f in the given variable of degree $\text{length}(\langle values \rangle)-1$. The unique polynomial f is defined by the property that for corresponding elements v of $\langle values \rangle$ and p of $\langle points \rangle$ the relation $f(p) = v$ holds.

The Aitken-Neville interpolation algorithm is used which guarantees a stable result even with rounded numbers and an ill-conditioned problem.

9.10 Obtaining Parts of Polynomials and Rationals

These operators select various parts of a polynomial or rational function structure. Except for the cost of rearrangement of the structure, these operations take very little time to perform.

For those operators in this section that take a kernel VAR as their second argument, an error results if the first expression is not a polynomial in VAR, although the coefficients themselves can be rational as long as they do not depend on VAR. However, if the switch RATARG is on, denominators are not checked for dependence on VAR, and are taken to be part of the coefficients.

9.10.1 DEG Operator

This operator is used with the syntax

```
DEG (EXPRN:polynomial, VAR:kernel) : integer.
```

It returns the leading degree of the polynomial `EXPRN` in the variable `VAR`. If `VAR` does not occur as a variable in `EXPRN`, 0 is returned.

Examples:

```
deg((a+b)*(c+2*d)^2,a) -> 1
deg((a+b)*(c+2*d)^2,d) -> 2
deg((a+b)*(c+2*d)^2,e) -> 0.
```

Note also that if `RATARG` is on,

```
deg((a+b)^3/a,a) -> 3
```

since in this case, the denominator `A` is considered part of the coefficients of the numerator in `A`. With `RATARG` off, however, an error would result in this case.

9.10.2 DEN Operator

This is used with the syntax:

```
DEN(EXPRN:rational):polynomial.
```

It returns the denominator of the rational expression `EXPRN`. If `EXPRN` is a polynomial, 1 is returned.

Examples:

```
den(x/y^2) -> Y**2
den(100/6) -> 3
           [since 100/6 is first simplified to 50/3]
den(a/4+b/6) -> 12
den(a+b) -> 1
```

9.10.3 LCOF Operator

`LCOF` is used with the syntax

```
LCOF(EXPRN:polynomial,VAR:kernel):polynomial.
```

It returns the leading coefficient of the polynomial `EXPRN` in the variable `VAR`. If `VAR` does not occur as a variable in `EXPRN`, `EXPRN` is returned.

Examples:

```
lcof((a+b)*(c+2*d)^2,a) -> C**2+4*C*D+4*D**2
lcof((a+b)*(c+2*d)^2,d) -> 4*(A+B)
lcof((a+b)*(c+2*d),e)    -> A*C+2*A*D+B*C+2*B*D
```

9.10.4 LPOWER Operator

Syntax:

```
LPOWER(EXPRN:polynomial,VAR:kernel):polynomial.
```

LPOWER returns the leading power of EXPRN with respect to VAR. If EXPRN does not depend on VAR, 1 is returned.

Examples:

```
lpower((a+b)*(c+2*d)^2,a) -> A
lpower((a+b)*(c+2*d)^2,d) -> D**2
lpower((a+b)*(c+2*d),e)    -> 1
```

9.10.5 LTERM Operator

Syntax:

```
LTERM(EXPRN:polynomial,VAR:kernel):polynomial.
```

LTERM returns the leading term of EXPRN with respect to VAR. If EXPRN does not depend on VAR, EXPRN is returned.

Examples:

```
lterm((a+b)*(c+2*d)^2,a) -> A*(C**2+4*C*D+4*D**2)
lterm((a+b)*(c+2*d)^2,d) -> 4*D**2*(A+B)
lterm((a+b)*(c+2*d),e)    -> A*C+2*A*D+B*C+2*B*D
```

Compatibility Note: In some earlier versions of REDUCE, LTERM returned 0 if the EXPRN did not depend on VAR. In the present version, EXPRN is always equal to LTERM(EXPRN,VAR) + REDUCT(EXPRN,VAR).

9.10.6 MAINVAR Operator

Syntax:

`MAINVAR (EXPRN:polynomial):expression.`

Returns the main variable (based on the internal polynomial representation) of EXPRN. If EXPRN is a domain element, 0 is returned.

Examples:

Assuming A has higher kernel order than B, C, or D:

```
mainvar((a+b)*(c+2*d)^2) -> A
mainvar(2)                -> 0
```

9.10.7 NUM Operator

Syntax:

`NUM (EXPRN:rational):polynomial.`

Returns the numerator of the rational expression EXPRN. If EXPRN is a polynomial, that polynomial is returned.

Examples:

```
num(x/y^2)   -> X
num(100/6)   -> 50
num(a/4+b/6) -> 3*A+2*B
num(a+b)     -> A+B
```

9.10.8 REDUCT Operator

Syntax:

`REDUCT (EXPRN:polynomial,VAR:kernel):polynomial.`

Returns the reductum of EXPRN with respect to VAR (i.e., the part of EXPRN left after the leading term is removed). If EXPRN does not depend on the variable VAR, 0 is returned.

Examples:

```
reduct((a+b)*(c+2*d),a) -> B*(C + 2*D)
reduct((a+b)*(c+2*d),d) -> C*(A + B)
reduct((a+b)*(c+2*d),e) -> 0
```

Compatibility Note: In some earlier versions of REDUCE, REDUCT returned EXPRN if it did not depend on VAR. In the present version, EXPRN is always equal to LTERM(EXPRN, VAR) + REDUCT(EXPRN, VAR).

9.10.9 TOTALDEG Operator

Syntax:

```
totaldeg(a*x^2+b*x+c, x)    => 2
totaldeg(a*x^2+b*x+c, {a,b,c}) => 1
totaldeg(a*x^2+b*x+c, {x, a}) => 3
totaldeg(a*x^2+b*x+c, {x,b}) => 2
totaldeg(a*x^2+b*x+c, {p,q,r}) => 0
```

totaldeg(u, kernlist) finds the total degree of the polynomial u in the variables in kernlist. If kernlist is not a list it is treated as a simple single variable. The denominator of u is ignored, and "degree" here does not pay attention to fractional powers. Mentions of a kernel within the argument to any operator or function (eg sin, cos, log, sqrt) are ignored. Really u is expected to be just a polynomial.

9.11 Polynomial Coefficient Arithmetic

REDUCE allows for a variety of numerical domains for the numerical coefficients of polynomials used in calculations. The default mode is integer arithmetic, although the possibility of using real coefficients has been discussed elsewhere. Rational coefficients have also been available by using integer coefficients in both the numerator and denominator of an expression, using the ON DIV option to print the coefficients as rationals. However, REDUCE includes several other coefficient options in its basic version which we shall describe in this section. All such coefficient modes are supported in a table-driven manner so that it is straightforward to extend the range of possibilities. A description of how to do this is given in R.J. Bradford, A.C. Hearn, J.A. Padget and E. Schröder, "Enlarging the REDUCE Domain of Computation," Proc. of SYMSAC '86, ACM, New York (1986), 100–106.

9.11.1 Rational Coefficients in Polynomials

Instead of treating rational numbers as the numerator and denominator of a rational expression, it is also possible to use them as polynomial coefficients directly. This is accomplished by turning on the switch RATIONAL.

Example: With RATIONAL off, the input expression $a/2$ would be converted

into a rational expression, whose numerator was A and denominator 2. With `RATIONAL` on, the same input would become a rational expression with numerator $1/2 * A$ and denominator 1. Thus the latter can be used in operations that require polynomial input whereas the former could not.

9.11.2 Real Coefficients in Polynomials

The switch `ROUNDED` permits the use of arbitrary sized real coefficients in polynomial expressions. The actual precision of these coefficients can be set by the operator `PRECISION`. For example, `precision 50;` sets the precision to fifty decimal digits. The default precision is system dependent and can be found by `precision 0; .` In this mode, denominators are automatically made monic, and an appropriate adjustment is made to the numerator.

Example: With `ROUNDED` on, the input expression $a/2$ would be converted into a rational expression whose numerator is $0.5 * A$ and denominator 1.

Internally, `REDUCE` uses floating point numbers up to the precision supported by the underlying machine hardware, and so-called *bigfloats* for higher precision or whenever necessary to represent numbers whose value cannot be represented in floating point. The internal precision is two decimal digits greater than the external precision to guard against roundoff inaccuracies. Bigfloats represent the fraction and exponent parts of a floating-point number by means of (arbitrary precision) integers, which is a more precise representation in many cases than the machine floating point arithmetic, but not as efficient. If a case arises where use of the machine arithmetic leads to problems, a user can force `REDUCE` to use the bigfloat representation at all precisions by turning on the switch `ROUNDBF`. In rare cases, this switch is turned on by the system, and the user informed by the message

ROUNDBF turned on to increase accuracy

Rounded numbers are normally printed to the specified precision. However, if the user wishes to print such numbers with less precision, the printing precision can be set by the command `PRINT_PRECISION`. For example, `print_precision 5;` will cause such numbers to be printed with five digits maximum.

Under normal circumstances when `ROUNDED` is on, `REDUCE` converts the number 1.0 to the integer 1. If this is not desired, the switch `NOCONVERT` can be turned on.

Numbers that are stored internally as bigfloats are normally printed with a space between every five digits to improve readability. If this feature is not required, it can be suppressed by turning off the switch `BFSpace`.

Further information on the bigfloat arithmetic may be found in T. Sasaki, "Manual for Arbitrary Precision Real Arithmetic System in `REDUCE`", Department of

Computer Science, University of Utah, Technical Note No. TR-8 (1979).

When a real number is input, it is normally truncated to the precision in effect at the time the number is read. If it is desired to keep the full precision of all numbers input, the switch ADJPREC (for *adjust precision*) can be turned on. While on, ADJPREC will automatically increase the precision, when necessary, to match that of any integer or real input, and a message printed to inform the user of the precision increase.

When ROUNDED is on, rational numbers are normally converted to rounded representation. However, if a user wishes to keep such numbers in a rational form until used in an operation that returns a real number, the switch ROUNDALL can be turned off. This switch is normally on.

Results from rounded calculations are returned in rounded form with two exceptions: if the result is recognized as 0 or 1 to the current precision, the integer result is returned.

9.11.3 Modular Number Coefficients in Polynomials

REDUCE includes facilities for manipulating polynomials whose coefficients are computed modulo a given base. To use this option, two commands must be used; SETMOD *<integer>*, to set the prime modulus, and ON MODULAR to cause the actual modular calculations to occur. For example, with `setmod 3;` and `on modular;`, the polynomial $(a+2*b)^3$ would become A^3+2*B^3 .

The argument of SETMOD is evaluated algebraically, except that non-modular (integer) arithmetic is used. Thus the sequence

```
setmod 3; on modular; setmod 7;
```

will correctly set the modulus to 7.

Modular numbers are by default represented by integers in the interval $[0, p-1]$ where p is the current modulus. Sometimes it is more convenient to use an equivalent symmetric representation in the interval $[-p/2+1, p/2]$, or more precisely $[-\text{floor}((p-1)/2), \text{ceiling}((p-1)/2)]$, especially if the modular numbers map objects that include negative quantities. The switch BALANCED_MOD allows you to select the symmetric representation for output.

Users should note that the modular calculations are on the polynomial coefficients only. It is not currently possible to reduce the exponents since no check for a prime modulus is made (which would allow x^{p-1} to be reduced to $1 \bmod p$). Note also that any division by a number not co-prime with the modulus will result in the error "Invalid modular division".

9.11.4 Complex Number Coefficients in Polynomials

Although REDUCE routinely treats the square of the variable i as equivalent to -1 , this is not sufficient to reduce expressions involving i to lowest terms, or to factor such expressions over the complex numbers. For example, in the default case,

```
factorize(a^2+1);
```

gives the result

```
{{A**2+1,1}}
```

and

```
(a^2+b^2)/(a+i*b)
```

is not reduced further. However, if the switch COMPLEX is turned on, full complex arithmetic is then carried out. In other words, the above factorization will give the result

```
{{A + I,1},{A - I,1}}
```

and the quotient will be reduced to $A - I \star B$.

The switch COMPLEX may be combined with ROUNDED to give complex real numbers; the appropriate arithmetic is performed in this case.

Complex conjugation is used to remove complex numbers from denominators of expressions. To do this if COMPLEX is off, you must turn the switch RATIONALIZE on.

9.12 ROOT_VAL Operator

The ROOT_VAL operator takes a single univariate polynomial as argument, and returns a list of root values at system precision (or greater if required to separate roots). It is used with the syntax

```
ROOT_VAL(EXPRN:univariate polynomial):list.
```

For example, the sequence

```
on rounded; root_val(x^3-x-1);
```

gives the result

$$\{0.562279512062*I - 0.662358978622, - 0.562279512062*I$$
$$- 0.662358978622, 1.32471795724\}$$

Chapter 10

Assigning and Testing Algebraic Properties

Sometimes algebraic expressions can be further simplified if there is additional information about the value ranges of its components. The following section describes how to inform REDUCE of such assumptions.

10.1 REALVALUED Declaration and Check

The declaration `REALVALUED` may be used to restrict variables to the real numbers. The syntax is:

```
realvalued v1, ...vn;
```

For such variables the operator `IMPART` gives the result zero. Thus, with

```
realvalued x,y;
```

the expression `impart(x+sin(y))` is evaluated as zero. You may also declare an operator as real valued with the meaning, that this operator maps real arguments always to real values. Example:

```
operator h; realvalued h,x;  
impart h(x);
```

```
0
```

```
impart h(w);
```

```
impart (h(w))
```

Such declarations are not needed for the standard elementary functions.

To remove the property from a variable or an operator use the declaration `NOREALVALUED` with the syntax:

```
norealvalued v1,...vn;
```

The boolean operator `REALVALUEDP` allows you to check if a variable, an operator, or an operator expression is known as real valued. Thus,

```
realvalued x;
write if realvaluedp(sin x) then "yes" else "no";
write if realvaluedp(sin z) then "yes" else "no";
```

would print first yes and then no. For general expressions test the `impart` for checking the value range:

```
realvalued x,y; w:=(x+i*y); w1:=conj w;
impart (w*w1);

0

impart (w*w);

2*x*y
```

10.2 Declaring Expressions Positive or Negative

Detailed knowledge about the sign of expressions allows `REDUCE` to simplify expressions involving exponentials or `ABS`. You can express assumptions about the *positivity* or *negativity* of expressions by rules for the operator `SIGN`. Examples:

```
abs(a*b*c);

abs(a*b*c);

let sign(a)=>1,sign(b)=>1; abs(a*b*c);

abs(c)*a*b

on precise; sqrt(x^2-2x+1);
```

```

abs(x - 1)

ws where sign(x-1)=>1;

x - 1

```

Here factors with known sign are factored out of an ABS expression.

```

on precise; on factor;

(q*x-2q) ^w;

      w
( (x - 2) *q)

ws where sign(x-2)=>1;

      w      w
q  * (x - 2)

```

In this case the factor $(x - 2)^w$ may be extracted from the base of the exponential because it is known to be positive.

Note that REDUCE knows a lot about sign propagation. For example, with x and y also $x + y$, $x + y + \pi$ and $(x + e)/y^2$ are known as positive. Nevertheless, it is often necessary to declare additionally the sign of a combined expression. E.g. at present a positivity declaration of $x - 2$ does not automatically lead to sign evaluation for $x - 1$ or for x .

Chapter 11

Substitution Commands

An important class of commands in REDUCE define substitutions for variables and expressions to be made during the evaluation of expressions. Such substitutions use the prefix operator SUB, various forms of the command LET, and rule sets.

11.1 SUB Operator

Syntax:

```
SUB(<substitution_list>, EXPRN1:algebraic):algebraic
```

where *<substitution_list>* is a list of one or more equations of the form

```
VAR:kernel=EXPRN:algebraic
```

or a kernel that evaluates to such a list.

The SUB operator gives the algebraic result of replacing every occurrence of the variable VAR in the expression EXPRN1 by the expression EXPRN. Specifically, EXPRN1 is first evaluated using all available rules. Next the substitutions are made, and finally the substituted expression is reevaluated. When more than one variable occurs in the substitution list, the substitution is performed by recursively walking down the tree representing EXPRN1, and replacing every VAR found by the appropriate EXPRN. The EXPRN are not themselves searched for any occurrences of the various VARs. The trivial case SUB(EXPRN1) returns the algebraic value of EXPRN1.

Examples:

2

2

$$\text{sub}(\{x=a+y, y=y+1\}, x^2+y^2) \rightarrow A^2 + 2* A * Y + 2* Y^2 + 2* Y + 1$$

and with $s := \{x=a+y, y=y+1\}$,

$$\text{sub}(s, x^2+y^2) \rightarrow A^2 + 2* A * Y + 2* Y^2 + 2* Y + 1$$

Note that the global assignments $x:=a+y$, etc., do not take place.

EXPRN1 can be any valid algebraic expression whose type is such that a substitution process is defined for it (e.g., scalar expressions, lists and matrices). An error will occur if an expression of an invalid type for substitution occurs either in EXPRN or EXPRN1.

The braces around the substitution list may also be omitted, as in:

$$\text{sub}(x=a+y, y=y+1, x^2+y^2) \rightarrow A^2 + 2* A * Y + 2* Y^2 + 2* Y + 1$$

11.2 LET Rules

Unlike substitutions introduced via SUB, LET rules are global in scope and stay in effect until replaced or CLEARED.

The simplest use of the LET statement is in the form

LET $\langle substitution\ list \rangle$

where $\langle substitution\ list \rangle$ is a list of rules separated by commas, each of the form:

$\langle variable \rangle = \langle expression \rangle$

or

$\langle prefix\ operator \rangle (\langle argument \rangle, \dots, \langle argument \rangle) = \langle expression \rangle$

or

$\langle argument \rangle \langle infix\ operator \rangle, \dots, \langle argument \rangle = \langle expression \rangle$

For example,

```
let {x => y^2,
     h(u,v) => u - v,
```

```

cos(pi/3) => 1/2,
a*b => c,
l+m => n,
w^3 => 2*z - 3,
z^10 => 0}

```

The list brackets can be left out if preferred. The above rules could also have been entered as seven separate LET statements.

After such LET rules have been input, X will always be evaluated as the square of Y , and so on. This is so even if at the time the LET rule was input, the variable Y had a value other than Y . (In contrast, the assignment $x := y^2$ will set X equal to the square of the current value of Y , which could be quite different.)

The rule `let a*b=c` means that whenever A and B are both factors in an expression their product will be replaced by C . For example, $a^5 * b^7 * w$ would be replaced by $c^5 * b^2 * w$.

The rule for `l+m` will not only replace all occurrences of $l+m$ by N , but will also normally replace L by $n-m$, but not M by $n-l$. A more complete description of this case is given in Section 11.2.5.

The rule pertaining to w^3 will apply to any power of W greater than or equal to the third.

Note especially the last example, `let z^10=0`. This declaration means, in effect: ignore the tenth or any higher power of Z . Such declarations, when appropriate, often speed up a computation to a considerable degree. (See Section 11.4 for more details.)

Any new operators occurring in such LET rules will be automatically declared OPERATOR by the system, if the rules are being read from a file. If they are being entered interactively, the system will ask `DECLARE ... OPERATOR?`. Answer `Y` or `N` and hit Return.

In each of these examples, substitutions are only made for the explicit expressions given; i.e., none of the variables may be considered arbitrary in any sense. For example, the command

```
let h(u,v) = u - v;
```

will cause $h(u, v)$ to evaluate to $U - V$, but will not affect $h(u, z)$ or H with any arguments other than precisely the symbols U, V .

These simple LET rules are on the same logical level as assignments made with the `:=` operator. An assignment $x := p+q$ cancels a rule `let x = y^2` made earlier, and vice versa.

CAUTION: A recursive rule such as

```
let x = x + 1;
```

is erroneous, since any subsequent evaluation of X would lead to a non-terminating chain of substitutions:

$$x \rightarrow x + 1 \rightarrow (x + 1) + 1 \rightarrow ((x + 1) + 1) + 1 \rightarrow \dots$$

Similarly, coupled substitutions such as

```
let l = m + n, n = l + r;
```

would lead to the same error. As a result, if you try to evaluate an X , L or N defined as above, you will get an error such as

```
X improperly defined in terms of itself
```

Array and matrix elements can appear on the left-hand side of a `LET` statement. However, because of their *instant evaluation* property, it is the value of the element that is substituted for, rather than the element itself. E.g.,

```
array a(5);
a(2) := b;
let a(2) = c;
```

results in B being substituted by C ; the assignment for $a(2)$ does not change.

Finally, if an error occurs in any equation in a `LET` statement (including generalized statements involving `FOR ALL` and `SUCH THAT`), the remaining rules are not evaluated.

11.2.1 FOR ALL ... LET

If a substitution for all possible values of a given argument of an operator is required, the declaration `FOR ALL` may be used. The syntax of such a command is

```
FOR ALL <variable>, ..., <variable>
  <LET statement> <terminator>
```

e.g.,

```
for all x,y let h(x,y) = x-y;
for all x let k(x,y) = x^y;
```

The first of these declarations would cause $h(a, b)$ to be evaluated as $A-B$, $h(u+v, u+w)$ to be $V-W$, etc. If the operator symbol H is used with more or fewer argument places, not two, the `LET` would have no effect, and no error would result.

The second declaration would cause $k(a, y)$ to be evaluated as a^y , but would have no effect on $k(a, z)$ since the rule didn't say `FOR ALL Y...`

Where we used X and Y in the examples, any variables could have been used. This use of a variable doesn't affect the value it may have outside the `LET` statement. However, you should remember what variables you actually used. If you want to delete the rule subsequently, you must use the same variables in the `CLEAR` command.

It is possible to use more complicated expressions as a template for a `LET` statement, as explained in the section on substitutions for general expressions. In nearly all cases, the rule will be accepted, and a consistent application made by the system. However, if there is a sole constant or a sole free variable on the left-hand side of a rule (e.g., `let 2=3` or `for all x let x=2`), then the system is unable to handle the rule, and the error message

```
Substitution for ... not allowed
```

will be issued. Any variable listed in the `FOR ALL` part will have its symbol preceded by an equal sign: X in the above example will appear as $=X$. An error will also occur if a variable in the `FOR ALL` part is not properly matched on both sides of the `LET` equation.

11.2.2 FOR ALL ... SUCH THAT ... LET

If a substitution is desired for more than a single value of a variable in an operator or other expression, but not all values, a conditional form of the `FOR ALL ... LET` declaration can be used.

Example:

```
for all x such that numberp x and x<0 let h(x)=0;
```

will cause $h(-5)$ to be evaluated as 0, but H of a positive integer, or of an argument that is not an integer at all, would not be affected. Any boolean expression can follow the `SUCH THAT` keywords.

11.2.3 Removing Assignments and Substitution Rules

The user may remove all assignments and substitution rules from any expression by the command `CLEAR`, in the form

$$\text{CLEAR } \langle expression \rangle, \dots, \langle expression \rangle = \langle terminator \rangle$$

e.g.

```
clear x, h(x,y);
```

Because of their *instant evaluation* property, array and matrix elements cannot be cleared with `CLEAR`. For example, if `A` is an array, you must say

```
a(3) := 0;
```

rather than

```
clear a(3);
```

to “clear” element `a(3)`.

On the other hand, a whole array (or matrix) `A` can be cleared by the command `clear a`; This means much more than resetting to 0 all the elements of `A`. The fact that `A` is an array, and what its dimensions are, are forgotten, so `A` can be redefined as another type of object, for example an operator.

The more general types of `LET` declarations can also be deleted by using `CLEAR`. Simply repeat the `LET` rule to be deleted, using `CLEAR` in place of `LET`, and omitting the equal sign and right-hand part. The same dummy variables must be used in the `FOR ALL` part, and the boolean expression in the `SUCH THAT` part must be written the same way. (The placing of blanks doesn’t have to be identical.)

Example: The `LET` rule

```
for all x such that numberp x and x<0 let h(x)=0;
```

can be erased by the command

```
for all x such that numberp x and x<0 clear h(x);
```

11.2.4 Overlapping LET Rules

`CLEAR` is not the only way to delete a `LET` rule. A new `LET` rule identical to the first, but with a different expression after the equal sign, replaces the first.

Replacements are also made in other cases where the existing rule would be in conflict with the new rule. For example, a rule for x^4 would replace a rule for x^5 . The user should however be cautioned against having several LET rules in effect that relate to the same expression. No guarantee can be given as to which rules will be applied by REDUCE or in what order. It is best to CLEAR an old rule before entering a new related LET rule.

11.2.5 Substitutions for General Expressions

The examples of substitutions discussed in other sections have involved very simple rules. However, the substitution mechanism used in REDUCE is very general, and can handle arbitrarily complicated rules without difficulty.

The general substitution mechanism used in REDUCE is discussed in Hearn, A. C., "REDUCE, A User-Oriented Interactive System for Algebraic Simplification," Interactive Systems for Experimental Applied Mathematics, (edited by M. Klerer and J. Reinfelds), Academic Press, New York (1968), 79-90, and Hearn, A. C., "The Problem of Substitution," Proc. 1968 Summer Institute on Symbolic Mathematical Computation, IBM Programming Laboratory Report FSC 69-0312 (1969). For the reasons given in these references, REDUCE does not attempt to implement a general pattern matching algorithm. However, the present system uses far more sophisticated techniques than those discussed in the above papers. It is now possible for the rules appearing in arguments of LET to have the form

$$\langle substitution\ expression \rangle = \langle expression \rangle$$

where any rule to which a sensible meaning can be assigned is permitted. However, this meaning can vary according to the form of $\langle substitution\ expression \rangle$. The semantic rules associated with the application of the substitution are completely consistent, but somewhat complicated by the pragmatic need to perform such substitutions as efficiently as possible. The following rules explain how the majority of the cases are handled.

To begin with, the $\langle substitution\ expression \rangle$ is first partly simplified by collecting like terms and putting identifiers (and kernels) in the system order. However, no substitutions are performed on any part of the expression with the exception of expressions with the *instant evaluation* property, such as array and matrix elements, whose actual values are used. It should also be noted that the system order used is not changeable by the user, even with the KORDER command. Specific cases are then handled as follows:

1. If the resulting simplified rule has a left-hand side that is an identifier, an expression with a top-level algebraic operator or a power, then the rule is added without further change to the appropriate table.

2. If the operator $*$ appears at the top level of the simplified left-hand side, then any constant arguments in that expression are moved to the right-hand side of the rule. The remaining left-hand side is then added to the appropriate table. For example,

$$\text{let } 2*x*y=3$$

becomes

$$\text{let } x*y=3/2$$

so that $x*y$ is added to the product substitution table, and when this rule is applied, the expression $x*y$ becomes $3/2$, but x or y by themselves are not replaced.

3. If the operators $+$, $-$ or $/$ appear at the top level of the simplified left-hand side, all but the first term is moved to the right-hand side of the rule. Thus the rules

$$\text{let } l+m=n, \quad x/2=y, \quad a-b=c$$

become

$$\text{let } l=n-m, \quad x=2*y, \quad a=c+b.$$

One problem that can occur in this case is that if a quantified expression is moved to the right-hand side, a given free variable might no longer appear on the left-hand side, resulting in an error because of the unmatched free variable. E.g.,

$$\text{for all } x,y \text{ let } f(x)+f(y)=x*y$$

would become

$$\text{for all } x,y \text{ let } f(x)=x*y-f(y)$$

which no longer has y on both sides.

The fact that array and matrix elements are evaluated in the left-hand side of rules can lead to confusion at times. Consider for example the statements

$$\text{array } a(5); \text{ let } x+a(2)=3; \text{ let } a(3)=4;$$

The left-hand side of the first rule will become x , and the second 0 . Thus the first rule will be instantiated as a substitution for x , and the second will result in an error.

The order in which a list of rules is applied is not easily understandable without a detailed knowledge of the system simplification protocol. It is also possible for this order to change from release to release, as improved substitution techniques are implemented. Users should therefore assume that the order of application of rules is arbitrary, and program accordingly.

After a substitution has been made, the expression being evaluated is reexamined in case a new allowed substitution has been generated. This process is continued until no more substitutions can be made.

As mentioned elsewhere, when a substitution expression appears in a product, the substitution is made if that expression divides the product. For example, the rule

```
let a^2*c = 3*z;
```

would cause a^2*c*x to be replaced by $3*z*x$ and a^2*c^2 by $3*z*c$. If the substitution is desired only when the substitution expression appears in a product with the explicit powers supplied in the rule, the command `MATCH` should be used instead.

For example,

```
match a^2*c = 3*z;
```

would cause a^2*c*x to be replaced by $3*z*x$, but a^2*c^2 would not be replaced. `MATCH` can also be used with the `FOR ALL` constructions described above.

To remove substitution rules of the type discussed in this section, the `CLEAR` command can be used, combined, if necessary, with the same `FOR ALL` clause with which the rule was defined, for example:

```
for all x clear log(e^x), e^log(x), cos(w*t+theta(x));
```

Note, however, that the arbitrary variable names in this case *must* be the same as those used in defining the substitution.

11.3 Rule Lists

Rule lists offer an alternative approach to defining substitutions that is different from either `SUB` or `LET`. In fact, they provide the best features of both, since they have all the capabilities of `LET`, but the rules can also be applied locally as is possible with `SUB`. In time, they will be used more and more in `REDUCE`. However, since they are relatively new, much of the `REDUCE` code you see uses the older

constructs.

A rule list is a list of *rules* that have the syntax

```
<expression> => <expression> (WHEN <boolean expression>)
```

For example,

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(~n*pi)      => (-1)^n when remainder(n,2)=0}
```

The tilde preceding a variable marks that variable as *free* for that rule, much as a variable in a FOR ALL clause in a LET statement. The first occurrence of that variable in each relevant rule must be so marked on input, otherwise inconsistent results can occur. For example, the rule list

```
{cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
 cos(x)^2        => (1+cos(2x))/2}
```

designed to replace products of cosines, would not be correct, since the second rule would only apply to the explicit argument X. Later occurrences in the same rule may also be marked, but this is optional (internally, all such rules are stored with each relevant variable explicitly marked). The optional WHEN clause allows constraints to be placed on the application of the rule, much as the SUCH THAT clause in a LET statement.

A rule list may be named, for example

```
trig1 := {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2,
          cos(~x)*sin(~y) => (sin(x+y)-sin(x-y))/2,
          sin(~x)*sin(~y) => (cos(x-y)-cos(x+y))/2,
          cos(~x)^2      => (1+cos(2*x))/2,
          sin(~x)^2      => (1-cos(2*x))/2};
```

Such named rule lists may be inspected as needed. E.g., the command `trig1;` would cause the above list to be printed.

Rule lists may be used in two ways. They can be globally instantiated by means of the command LET. For example,

```
let trig1;
```

would cause the above list of rules to be globally active from then on until cancelled by the command `CLEARRULES`, as in

```
clearrules trig1;
```

CLEARRULES has the syntax

```
CLEARRULES <rule list>|<name of rule list>(, ...) .
```

The second way to use rule lists is to invoke them locally by means of a WHERE clause. For example

```
cos(a)*cos(b+c)
  where {cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2};
```

or

```
cos(a)*sin(b) where trigrules;
```

The syntax of an expression with a WHERE clause is:

```
<expression>
  WHERE <rule>|<rule list>(,<rule>|<rule list> ...)
```

so the first example above could also be written

```
cos(a)*cos(b+c)
  where cos(~x)*cos(~y) => (cos(x+y)+cos(x-y))/2;
```

The effect of this construct is that the rule list(s) in the WHERE clause only apply to the expression on the left of WHERE. They have no effect outside the expression. In particular, they do not affect previously defined WHERE clauses or LET statements. For example, the sequence

```
let a=2;
a where a=>4;
a;
```

would result in the output

```
4
```

```
2
```

Although WHERE has a precedence less than any other infix operator, it still binds higher than keywords such as ELSE, THEN, DO, REPEAT and so on. Thus the expression

```
if a=2 then 3 else a+2 where a=3
```

will parse as

```
if a=2 then 3 else (a+2 where a=3)
```

WHERE may be used to introduce auxiliary variables in symbolic mode expressions, as described in Section 17.4. However, the symbolic mode use has different semantics, so expressions do not carry from one mode to the other.

Compatibility Note: In order to provide compatibility with older versions of rule lists released through the Network Library, it is currently possible to use an equal sign interchangeably with the replacement sign \Rightarrow in rules and LET statements. However, since this will change in future versions, the replacement sign is preferable in rules and the equal sign in non-rule-based LET statements.

Advanced Use of Rule Lists

Some advanced features of the rule list mechanism make it possible to write more complicated rules than those discussed so far, and in many cases to write more compact rule lists. These features are:

- Free operators
- Double slash operator
- Double tilde variables.

A *free operator* in the left hand side of a pattern will match any operator with the same number of arguments. The free operator is written in the same style as a variable. For example, the implementation of the product rule of differentiation can be written as:

```
operator diff, !~f, !~g;

prule := {diff(~f(~x) * ~g(~x), x) =>
          diff(f(x), x) * g(x) + diff(g(x), x) * f(x)};

let prule;

diff(sin(z)*cos(z), z);

cos(z)*diff(sin(z), z) + diff(cos(z), z)*sin(z)
```

The *double slash operator* may be used as an alternative to a single slash (quotient) in order to match quotients properly. E.g., in the example of the Gamma function above, one can use:

```

gammarrule :=
  {gamma(~z) // (~c*gamma(~zz)) => gamma(z) / (c*gamma(zz-1)*zz)
   when fixp(zz -z) and (zz -z) >0,
   gamma(~z) // gamma(~zz) => gamma(z) / (gamma(zz-1)*zz)
   when fixp(zz -z) and (zz -z) >0};

```

```
let gammarrule;
```

```
gamma(z) / gamma(z+3);
```

$$\frac{1}{z^3 + 6z^2 + 11z + 6}$$

The above example suffers from the fact that two rules had to be written in order to perform the required operation. This can be simplified by the use of *double tilde variables*. E.g. the rule list

```

GGrule := {
  gamma(~z) // (~~c*gamma(~zz)) => gamma(z) / (c*gamma(zz-1)*zz)
  when fixp(zz -z) and (zz -z) >0};

```

will implement the same operation in a much more compact way. In general, double tilde variables are bound to the neutral element with respect to the operation in which they are used.

Pattern given	Argument used	Binding
$\sim z + \sim\sim y$	x	$z=x; y=0$
$\sim z + \sim\sim y$	$x+3$	$z=x; y=3$ or $z=3; y=x$
$\sim z * \sim\sim y$	x	$z=x; y=1$
$\sim z * \sim\sim y$	$x*3$	$z=x; y=3$ or $z=3; y=x$
$\sim z / \sim\sim y$	x	$z=x; y=1$
$\sim z / \sim\sim y$	$x/3$	$z=x; y=3$

Remarks: A double tilde variable as the numerator of a pattern is not allowed. Also, using double tilde variables may lead to recursion errors when the zero case is not handled properly.

```
let f(~~a * ~x, x) => a * f(x, x) when freeof (a, x);
```

```

f(z,z);

***** f(z,z) improperly defined in terms of itself

% BUT:

let ff(~a * ~x,x)
    => a * ff(x,x) when freeof (a,x) and a neq 1;

ff(z,z);
                                ff(z,z)

ff(3*z,z);
                                3*ff(z,z)

```

Displaying Rules Associated with an Operator

The operator `SHOWRULES` takes a single identifier as argument, and returns in rule-list form the operator rules associated with that argument. For example:

```

showrules log;

{LOG(E) => 1,

 LOG(1) => 0,

      ~X
 LOG(E  ) => ~X,

      1
 DF (LOG (~X) , ~X) => ----}
      ~X

```

Such rules can then be manipulated further as with any list. For example `rhs first ws;` has the value 1. Note that an operator may have other properties that cannot be displayed in such a form, such as the fact it is an odd function, or has a definition defined as a procedure.

Order of Application of Rules

If rules have overlapping domains, their order of application is important. In general, it is very difficult to specify this order precisely, so that it is best to assume

that the order is arbitrary. However, if only one operator is involved, the order of application of the rules for this operator can be determined from the following:

1. Rules containing at least one free variable apply before all rules without free variables.
2. Rules activated in the most recent `LET` command are applied first.
3. `LET` with several entries generate the same order of application as a corresponding sequence of commands with one rule or rule set each.
4. Within a rule set, the rules containing at least one free variable are applied in their given order. In other words, the first member of the list is applied first.
5. Consistent with the first item, any rule in a rule list that contains no free variables is applied after all rules containing free variables.

Example: The following rule set enables the computation of exact values of the Gamma function:

```
operator gamma,gamma_error;
gamma_rules :=
{gamma(~x)=>sqrt(pi)/2 when x=1/2,
 gamma(~n)=>factorial(n-1) when fixp n and n>0,
 gamma(~n)=>gamma_error(n) when fixp n,
 gamma(~x)=>(x-1)*gamma(x-1) when fixp(2*x) and x>1,
 gamma(~x)=>gamma(x+1)/x when fixp(2*x)};
```

Here, rule by rule, cases of known or definitely uncomputable values are sorted out; e.g. the rule leading to the error expression will be applied for negative integers only, since the positive integers are caught by the preceding rule, and the last rule will apply for negative odd multiples of $1/2$ only. Alternatively the first rule could have been written as

```
gamma(1/2) => sqrt(pi)/2,
```

but then the case $x = 1/2$ should be excluded in the `WHEN` part of the last rule explicitly because a rule without free variables cannot take precedence over the other rules.

11.4 Asymptotic Commands

In expansions of polynomials involving variables that are known to be small, it is often desirable to throw away all powers of these variables beyond a certain point

to avoid unnecessary computation. The command `LET` may be used to do this. For example, if only powers of `X` up to `x^7` are needed, the command

```
let x^8 = 0;
```

will cause the system to delete all powers of `X` higher than 7.

CAUTION: This particular simplification works differently from most substitution mechanisms in `REDUCE` in that it is applied during polynomial manipulation rather than to the whole evaluated expression. Thus, with the above rule in effect, `x^10/x^5` would give the result zero, since the numerator would simplify to zero. Similarly `x^20/x^10` would give a `Zero divisor` error message, since both numerator and denominator would first simplify to zero.

The method just described is not adequate when expressions involve several variables having different degrees of smallness. In this case, it is necessary to supply an asymptotic weight to each variable and count up the total weight of each product in an expanded expression before deciding whether to keep the term or not. There are two associated commands in the system to permit this type of asymptotic constraint. The command `WEIGHT` takes a list of equations of the form

$$\langle \text{kernel form} \rangle = \langle \text{number} \rangle$$

where $\langle \text{number} \rangle$ must be a positive integer (not just evaluate to a positive integer). This command assigns the weight $\langle \text{number} \rangle$ to the relevant kernel form. A check is then made in all algebraic evaluations to see if the total weight of the term is greater than the weight level assigned to the calculation. If it is, the term is deleted. To compute the total weight of a product, the individual weights of each kernel form are multiplied by their corresponding powers and then added.

The weight level of the system is initially set to 1. The user may change this setting by the command

```
wtlevel <number>;
```

which sets $\langle \text{number} \rangle$ as the new weight level of the system. `meta` must evaluate to a positive integer. `WTLEVEL` will also allow `NIL` as an argument, in which case the current weight level is returned.

Chapter 12

File Handling Commands

In many applications, it is desirable to load previously prepared REDUCE files into the system, or to write output on other files. REDUCE offers four commands for this purpose, namely, IN, OUT, SHUT, LOAD, and LOAD_PACKAGE. The first three operators are described here; LOAD and LOAD_PACKAGE are discussed in [Section 19.2](#).

12.1 IN Command

This command takes a list of file names as argument and directs the system to input each file (that should contain REDUCE statements and commands) into the system. File names can either be an identifier or a string. The explicit format of these will be system dependent and, in many cases, site dependent. The explicit instructions for the implementation being used should therefore be consulted for further details. For example:

```
in f1, "ggg.rr.s";
```

will first load file `F1`, then `ggg.rr.s`. When a semicolon is used as the terminator of the IN statement, the statements in the file are echoed on the terminal or written on the current output file. If `$` is used as the terminator, the input is not shown. Echoing of all or part of the input file can be prevented, even if a semicolon was used, by placing an `off echo;` command in the input file.

Files to be read using IN should end with `;END;`. Note the two semicolons! First of all, this is protection against obscure difficulties the user will have if there are, by mistake, more `BEGINS` than `ENDs` on the file. Secondly, it triggers some file control book-keeping which may improve system efficiency. If `END` is omitted, an error message `"End-of-file read"` will occur.

While a file is being loaded, the special identifier `_LINE_` is replaced by the number of the current line in the file currently being read.

12.2 OUT Command

This command takes a single file name as argument, and directs output to that file from then on, until another `OUT` changes the output file, or `SHUT` closes it. Output can go to only one file at a time, although many can be open. If the file has previously been used for output during the current job, and not `SHUT`, the new output is appended to the end of the file. Any existing file is erased before its first use for output in a job, or if it had been `SHUT` before the new `OUT`.

To output on the terminal without closing the output file, the reserved file name `T` (for terminal) may be used. For example, `out ofile;` will direct output to the file `OFILE` and `out t;` will direct output to the user's terminal.

The output sent to the file will be in the same form that it would have on the terminal. In particular x^2 would appear on two lines, an `X` on the lower line and a `2` on the line above. If the purpose of the output file is to save results to be read in later, this is not an appropriate form. We first must turn off the `NAT` switch that specifies that output should be in standard mathematical notation.

Example: To create a file `ABCD` from which it will be possible to read – using `IN` – the value of the expression `XYZ`:

```
off echo$      % needed if your input is from a file.
off nat$       % output in IN-readable form. Each expression
                % printed will end with a $ .
out abcd$      % output to new file
linelength 72$ % for systems with fixed input line length.
xyz:=xyz;      % will output "XYZ := " followed by the value
                % of XYZ
write ";end"$  % standard for ending files for IN
shut abcd$     % save ABCD, return to terminal output
on nat$        % restore usual output form
```

12.3 SHUT Command

This command takes a list of names of files that have been previously opened via an `OUT` statement and closes them. Most systems require this action by the user before he ends the `REDUCE` job (if not sooner), otherwise the output may be lost. If a file is shut and a further `OUT` command issued for the same file, the file is erased before the new output is written.

If it is the current output file that is shut, output will switch to the terminal. Attempts to shut files that have not been opened by `OUT`, or an input file, will lead to errors.

Chapter 13

Commands for Interactive Use

REDUCE is designed as an interactive system, but naturally it can also operate in a batch processing or background mode by taking its input command by command from the relevant input stream. There is a basic difference, however, between interactive and batch use of the system. In the former case, whenever the system discovers an ambiguity at some point in a calculation, such as a forgotten type assignment for instance, it asks the user for the correct interpretation. In batch operation, it is not practical to terminate the calculation at such points and require resubmission of the job, so the system makes the most obvious guess of the user's intentions and continues the calculation.

There is also a difference in the handling of errors. In the former case, the computation can continue since the user has the opportunity to correct the mistake. In batch mode, the error may lead to consequent erroneous (and possibly time consuming) computations. So in the default case, no further evaluation occurs, although the remainder of the input is checked for syntax errors. A message "Continuing with parsing only" informs the user that this is happening. On the other hand, the switch `ERRCONT`, if on, will cause the system to continue evaluating expressions after such errors occur.

When a syntactical error occurs, the place where the system detected the error is marked with three dollar signs (`$$$`). In interactive mode, the user can then use `ED` to correct the error, or retype the command. When a non-syntactical error occurs in interactive mode, the command being evaluated at the time the last error occurred is saved, and may later be reevaluated by the command `RETRY`.

13.1 Referencing Previous Results

It is often useful to be able to reference results of previous computations during a REDUCE session. For this purpose, REDUCE maintains a history of all interactive

inputs and the results of all interactive computations during a given session. These results are referenced by the command number that REDUCE prints automatically in interactive mode. To use an input expression in a new computation, one writes `input (n)`, where n is the command number. To use an output expression, one writes `ws (n)`. `WS` references the previous command. E.g., if command number 1 was `INT (X-1, X)`; and the result of command number 7 was $X-1$, then

$$2 * \text{input}(1) - \text{ws}(7)^2;$$

would give the result -1 , whereas

$$2 * \text{ws}(1) - \text{ws}(7)^2;$$

would yield the same result, but *without* a recomputation of the integral.

The operator `DISPLAY` is available to display previous inputs. If its argument is a positive integer, n say, then the previous n inputs are displayed. If its argument is `ALL` (or in fact any non-numerical expression), then all previous inputs are displayed.

13.2 Interactive Editing

It is possible when working interactively to edit any REDUCE input that comes from the user's terminal, and also some user-defined procedure definitions. At the top level, one can access any previous command string by the command `ed (n)`, where n is the desired command number as prompted by the system in interactive mode. `ED`; (i.e. no argument) accesses the previous command.

After `ED` has been called, you can now edit the displayed string using a string editor with the following commands:

<code>B</code>	move pointer to beginning
<code>C<character></code>	replace next character by <code><character></code>
<code>D</code>	delete next character
<code>E</code>	end editing and reread text
<code>F<character></code>	move pointer to next occurrence of <code><character></code>
<code>I<string><escape></code>	insert <code><string></code> in front of pointer
<code>K<character></code>	delete all characters until <code><character></code>
<code>P</code>	print string from current pointer
<code>Q</code>	give up with error exit
<code>S<string><escape></code>	search for first occurrence of <code><string></code> , positioning pointer just before it
<code>space or X</code>	move pointer right one character.

The above table can be displayed online by typing a question mark followed by a carriage return to the editor. The editor prompts with an angle bracket. Commands can be combined on a single line, and all command sequences must be followed by a carriage return to become effective.

Thus, to change the command `x := a+1;` to `x := a+2;` and cause it to be executed, the following edit command sequence could be used:

```
flc2e<return>.
```

The interactive editor may also be used to edit a user-defined procedure that has not been compiled. To do this, one says:

```
editdef <id>;
```

where `<id>` is the name of the procedure. The procedure definition will then be displayed in editing mode, and may then be edited and redefined on exiting from the editor.

Some versions of REDUCE now include input editing that uses the capabilities of modern window systems. Please consult your system dependent documentation to see if this is possible. Such editing techniques are usually much easier to use than ED or EDITDEF.

13.3 Interactive File Control

If input is coming from an external file, the system treats it as a batch processed calculation. If the user desires interactive response in this case, he can include the command `ON INT;` in the file. Likewise, he can issue the command `off int;` in the main program if he does not desire continual questioning from the system. Regardless of the setting of `INT`, input commands from a file are not kept in the system, and so cannot be edited using ED. However, many implementations of REDUCE provide a link to an external system editor that can be used for such editing. The specific instructions for the particular implementation should be consulted for information on this.

Two commands are available in REDUCE for interactive use of files. `PAUSE;` may be inserted at any point in an input file. When this command is encountered on input, the system prints the message `CONT?` on the user's terminal and halts. If the user responds Y (for yes), the calculation continues from that point in the file. If the user responds N (for no), control is returned to the terminal, and the user can input further statements and commands. Later on he can use the command `cont;` to transfer control back to the point in the file following the last `PAUSE` encountered. A top-level `pause;` from the user's terminal has no effect.

Chapter 14

Matrix Calculations

A very powerful feature of REDUCE is the ease with which matrix calculations can be performed. To extend our syntax to this class of calculations we need to add another prefix operator, `MAT`, and a further variable and expression type as follows:

14.1 MAT Operator

This prefix operator is used to represent $n \times m$ matrices. `MAT` has n arguments interpreted as rows of the matrix, each of which is a list of m expressions representing elements in that row. For example, the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix}$$

would be written as `mat ((a, b, c) , (d, e, f))`.

Note that the single column matrix

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

becomes `mat ((x) , (y))`. The inside parentheses are required to distinguish it from the single row matrix

$$(x \ y)$$

that would be written as `mat ((x, y))`.

14.2 Matrix Variables

An identifier may be declared a matrix variable by the declaration `MATRIX`. The size of the matrix may be declared explicitly in the matrix declaration, or by default

in assigning such a variable to a matrix expression. For example,

```
matrix x(2,1), y(3,4), z;
```

declares X to be a 2 x 1 (column) matrix, Y to be a 3 x 4 matrix and Z a matrix whose size is to be declared later.

Matrix declarations can appear anywhere in a program. Once a symbol is declared to name a matrix, it can not also be used to name an array, operator or a procedure, or used as an ordinary variable. It can however be redeclared to be a matrix, and its size may be changed at that time. Note however that matrices once declared are *global* in scope, and so can then be referenced anywhere in the program. In other words, a declaration within a block (or a procedure) does not limit the scope of the matrix to that block, nor does the matrix go away on exiting the block (use `CLEAR` instead for this purpose). An element of a matrix is referred to in the expected manner; thus $x(1,1)$ gives the first element of the matrix X defined above. References to elements of a matrix whose size has not yet been declared leads to an error. All elements of a matrix whose size is declared are initialized to 0. As a result, a matrix element has an *instant evaluation* property and cannot stand for itself. If this is required, then an operator should be used to name the matrix elements as in:

```
matrix m; operator x; m := mat((x(1,1), x(1,2)));
```

14.3 Matrix Expressions

These follow the normal rules of matrix algebra as defined by the following syntax:

$$\begin{aligned} \langle \text{matrix expression} \rangle \longrightarrow & \text{MAT} \langle \text{matrix description} \rangle \mid \langle \text{matrix variable} \rangle \mid \\ & \langle \text{scalar expression} \rangle * \langle \text{matrix expression} \rangle \mid \\ & \langle \text{matrix expression} \rangle * \langle \text{matrix expression} \rangle \mid \\ & \langle \text{matrix expression} \rangle + \langle \text{matrix expression} \rangle \mid \\ & \langle \text{matrix expression} \rangle ^ \langle \text{integer} \rangle \mid \\ & \langle \text{matrix expression} \rangle / \langle \text{matrix expression} \rangle \end{aligned}$$

Sums and products of matrix expressions must be of compatible size; otherwise an error will result during their evaluation. Similarly, only square matrices may be raised to a power. A negative power is computed as the inverse of the matrix raised to the corresponding positive power. a/b is interpreted as $a * b^{-1}$.

Examples:

Assuming X and Y have been declared as matrices, the following are matrix expressions

```

y
y^2*x-3*y^(-2)*x
y + mat((1,a),(b,c))/2

```

The computation of the quotient of two matrices normally uses a two-step elimination method due to Bareiss. An alternative method using Cramer's method is also available. This is usually less efficient than the Bareiss method unless the matrices are large and dense, although we have no solid statistics on this as yet. To use Cramer's method instead, the switch `CRAMER` should be turned on.

14.4 Operators with Matrix Arguments

The operator `LENGTH` applied to a matrix returns a list of the number of rows and columns in the matrix. Other operators useful in matrix calculations are defined in the following subsections. Attention is also drawn to the `LINALG` (section 16.34) and `NORMFORM` (section 16.38) packages.

14.4.1 DET Operator

Syntax:

```
DET(EXPRN:matrix_expression):algebraic.
```

The operator `DET` is used to represent the determinant of a square matrix expression. E.g.,

```
det(y^2)
```

is a scalar expression whose value is the determinant of the square of the matrix Y , and

```
det mat((a,b,c),(d,e,f),(g,h,j));
```

is a scalar expression whose value is the determinant of the matrix

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & j \end{pmatrix}$$

Determinant expressions have the *instant evaluation* property. In other words, the statement

```
let det mat((a,b),(c,d)) = 2;
```

sets the *value* of the determinant to 2, and does not set up a rule for the determinant itself.

14.4.2 MATEIGEN Operator

Syntax:

```
MATEIGEN (EXPRN:matrix_expression, ID):list.
```

MATEIGEN calculates the eigenvalue equation and the corresponding eigenvectors of a matrix, using the variable `ID` to denote the eigenvalue. A square free decomposition of the characteristic polynomial is carried out. The result is a list of lists of 3 elements, where the first element is a square free factor of the characteristic polynomial, the second its multiplicity and the third the corresponding eigenvector (as an n by 1 matrix). If the square free decomposition was successful, the product of the first elements in the lists is the minimal polynomial. In the case of degeneracy, several eigenvectors can exist for the same eigenvalue, which manifests itself in the appearance of more than one arbitrary variable in the eigenvector. To extract the various parts of the result use the operations defined on lists.

Example: The command

```
mateigen(mat((2,-1,1),(0,1,1),(-1,1,1)),eta);
```

gives the output

```
{ {ETA - 1, 2,
  [ARBCOMPLEX(1)]
  [
  [ARBCOMPLEX(1)]
  [
  [ 0
  ]
  ],
},
{ETA - 2, 1,
  [ 0
  ]
  [
  [ARBCOMPLEX(2)]
  [
  [ARBCOMPLEX(2)]
  ]
  ]
}
```

```
}}
```

14.4.3 TP Operator

Syntax:

```
TP (EXPRN:matrix_expression):matrix.
```

This operator takes a single matrix argument and returns its transpose.

14.4.4 Trace Operator

Syntax:

```
TRACE (EXPRN:matrix_expression):algebraic.
```

The operator TRACE is used to represent the trace of a square matrix.

14.4.5 Matrix Cofactors

Syntax:

```
COFACTOR (EXPRN:matrix_expression, ROW:integer, COLUMN:integer):  
algebraic
```

The operator COFACTOR returns the cofactor of the element in row ROW and column COLUMN of the matrix MATRIX. Errors occur if ROW or COLUMN do not simplify to integer expressions or if MATRIX is not square.

14.4.6 NULLSPACE Operator

Syntax:

```
NULLSPACE (EXPRN:matrix_expression):list
```

NULLSPACE calculates for a matrix A a list of linear independent vectors (a basis) whose linear combinations satisfy the equation $Ax = 0$. The basis is provided in a form such that as many upper components as possible are isolated.

Note that with `b := nullspace a` the expression `length b` is the *nullity* of A, and that `second length a - length b` calculates the *rank* of A. The

rank of a matrix expression can also be found more directly by the RANK operator described below.

Example: The command

```
nullspace mat((1,2,3,4),(5,6,7,8));
```

gives the output

```
{
  [ 1 ]
  [   ]
  [ 0 ]
  [   ]
  [ -3]
  [   ]
  [ 2 ]
,
  [ 0 ]
  [   ]
  [ 1 ]
  [   ]
  [ -2]
  [   ]
  [ 1 ]
}
```

In addition to the REDUCE matrix form, NULLSPACE accepts as input a matrix given as a list of lists, that is interpreted as a row matrix. If that form of input is chosen, the vectors in the result will be represented by lists as well. This additional input syntax facilitates the use of NULLSPACE in applications different from classical linear algebra.

14.4.7 RANK Operator

Syntax:

```
RANK(EXPRN:matrix_expression):integer
```

RANK calculates the rank of its argument, that, like NULLSPACE can either be a standard matrix expression, or a list of lists, that can be interpreted either as a row matrix or a set of equations.

Example:

```
rank mat((a,b,c),(d,e,f));
```

returns the value 2.

14.5 Matrix Assignments

Matrix expressions may appear in the right-hand side of assignment statements. If the left-hand side of the assignment, which must be a variable, has not already been declared a matrix, it is declared by default to the size of the right-hand side. The variable is then set to the value of the right-hand side.

Such an assignment may be used very conveniently to find the solution of a set of linear equations. For example, to find the solution of the following set of equations

$$\begin{aligned}a_{11}x(1) + a_{12}x(2) &= y_1 \\ a_{21}x(1) + a_{22}x(2) &= y_2\end{aligned}$$

we simply write

```
x := 1/mat((a11,a12),(a21,a22))*mat((y1),(y2));
```

14.6 Evaluating Matrix Elements

Once an element of a matrix has been assigned, it may be referred to in standard array element notation. Thus $y(2,1)$ refers to the element in the second row and first column of the matrix Y .

Chapter 15

Procedures

It is often useful to name a statement for repeated use in calculations with varying parameters, or to define a complete evaluation procedure for an operator. REDUCE offers a procedural declaration for this purpose. Its general syntax is:

$$[\langle procedural\ type \rangle] \text{ PROCEDURE } \langle name \rangle [\langle varlist \rangle]; \langle statement \rangle;$$

where

$$\langle varlist \rangle \longrightarrow (\langle variable \rangle, \dots, \langle variable \rangle)$$

This will be explained more fully in the following sections.

In the algebraic mode of REDUCE the $\langle procedural\ type \rangle$ can be omitted, since the default is ALGEBRAIC. Procedures of type INTEGER or REAL may also be used. In the former case, the system checks that the value of the procedure is an integer. At present, such checking is not done for a real procedure, although this will change in the future when a more complete type checking mechanism is installed. Users should therefore only use these types when appropriate. An empty variable list may also be omitted.

All user-defined procedures are automatically declared to be operators.

In order to allow users relatively easy access to the whole REDUCE source program, system procedures are not protected against user redefinition. If a procedure is redefined, a message

$$*** \langle procedure\ name \rangle \text{ REDEFINED}$$

is printed. If this occurs, and the user is not redefining his own procedure, he is well advised to rename it, and possibly start over (because he has *already* redefined some internal procedure whose correct functioning may be required for his job!)

All required procedures should be defined at the top level, since they have global scope throughout a program. In particular, an attempt to define a procedure within a procedure will cause an error to occur.

15.1 Procedure Heading

Each procedure has a heading consisting of the word `PROCEDURE` (optionally preceded by the word `ALGEBRAIC`), followed by the name of the procedure to be defined, and followed by its formal parameters – the symbols that will be used in the body of the definition to illustrate what is to be done. There are three cases:

1. No parameters. Simply follow the procedure name with a terminator (semicolon or dollar sign).

```
procedure abc;
```

When such a procedure is used in an expression or command, `abc()`, with empty parentheses, must be written.

2. One parameter. Enclose it in parentheses *or* just leave at least one space, then follow with a terminator.

```
procedure abc(x);
```

or

```
procedure abc x;
```

3. More than one parameter. Enclose them in parentheses, separated by commas, then follow with a terminator.

```
procedure abc(x,y,z);
```

Referring to the last example, if later in some expression being evaluated the symbols `abc(u,p*q,123)` appear, the operations of the procedure body will be carried out as if `X` had the same value as `U` does, `Y` the same value as `p*q` does, and `Z` the value 123. The values of `X`, `Y`, `Z`, after the procedure body operations are completed are unchanged. So, normally, are the values of `U`, `P`, `Q`, and (of course) 123. (This is technically referred to as call by value.)

The reader will have noted the word *normally* a few lines earlier. The call by value protections can be bypassed if necessary, as described elsewhere.

15.2 Procedure Body

Following the delimiter that ends the procedure heading must be a *single* statement defining the action to be performed or the value to be delivered. A terminator must follow the statement. If it is a semicolon, the name of the procedure just defined is printed. It is not printed if a dollar sign is used.

If the result wanted is given by a formula of some kind, the body is just that formula, using the variables in the procedure heading.

Simple Example:

If $f(x)$ is to mean $(x+5) * (x+6) / (x+7)$, the entire procedure definition could read

```
procedure f x; (x+5) * (x+6) / (x+7) ;
```

Then $f(10)$ would evaluate to $240/17$, $f(a-6)$ to $A * (A-1) / (A+1)$, and so on.

More Complicated Example:

Suppose we need a function $p(n, x)$ that, for any positive integer N , is the Legendre polynomial of order n . We can define this operator using the textbook formula defining these functions:

$$p_n(x) = \frac{1}{n!} \frac{d^n}{dy^n} \frac{1}{(y^2 - 2xy + 1)^{\frac{1}{2}}} \bigg|_{y=0}$$

Put into words, the Legendre polynomial $p_n(x)$ is the result of substituting $y = 0$ in the n^{th} partial derivative with respect to y of a certain fraction involving x and y , then dividing that by $n!$.

This verbal formula can easily be written in REDUCE:

```
procedure p(n, x);
  sub(y=0, df(1/(y^2-2*x*y+1)^(1/2), y, n))
  /(for i:=1:n product i);
```

Having input this definition, the expression evaluation

```
2p(2, w);
```

would result in the output

```
2
3*W - 1 .
```

If the desired process is best described as a series of steps, then a group or compound statement can be used.

Example:

The above Legendre polynomial example can be rewritten as a series of steps instead of a single formula as follows:

```
procedure p(n,x);
begin scalar seed,deriv,top,fact;
  seed:=1/(y^2 - 2*x*y +1)^(1/2);
  deriv:=df(seed,y,n);
  top:=sub(y=0,deriv);
  fact:=for i:=1:n product i;
  return top/fact
end;
```

Procedures may also be defined recursively. In other words, the procedure body can include references to the procedure name itself, or to other procedures that themselves reference the given procedure. As an example, we can define the Legendre polynomial through its standard recurrence relation:

```
procedure p(n,x);
if n<0 then rederr "Invalid argument to P(N,X) "
else if n=0 then 1
else if n=1 then x
else ((2*n-1)*x*p(n-1,x)-(n-1)*p(n-2,x))/n;
```

The operator REDERR in the above example provides for a simple error exit from an algebraic procedure (and also a block). It can take a string as argument.

It should be noted however that all the above definitions of $p(n, x)$ are quite inefficient if extensive use is to be made of such polynomials, since each call effectively recomputes all lower order polynomials. It would be better to store these expressions in an array, and then use say the recurrence relation to compute only those polynomials that have not already been derived. We leave it as an exercise for the reader to write such a definition.

15.3 Using LET Inside Procedures

By using LET instead of an assignment in the procedure body it is possible to bypass the call-by-value protection. If X is a formal parameter or local variable of the procedure (i.e. is in the heading or in a local declaration), and LET is used instead of := to make an assignment to X, e.g.

```
let x = 123;
```

then it is the variable that is the value of x that is changed. This effect also occurs with local variables defined in a block. If the value of x is not a variable, but a more general expression, then it is that expression that is used on the left-hand side of the `LET` statement. For example, if x had the value $p*q$, it is as if `let p*q = 123` had been executed.

15.4 LET Rules as Procedures

The `LET` statement offers an alternative syntax and semantics for procedure definition.

In place of

```
procedure abc(x,y,z); <procedure body>;
```

one can write

```
for all x,y,z let abc(x,y,z) = <procedure body>;
```

There are several differences to note.

If the procedure body contains an assignment to one of the formal parameters, e.g.

```
x := 123;
```

in the `PROCEDURE` case it is a variable holding a copy of the first actual argument that is changed. The actual argument is not changed.

In the `LET` case, the actual argument is changed. Thus, if `ABC` is defined using `LET`, and `abc(u,v,w)` is evaluated, the value of u changes to 123. That is, the `LET` form of definition allows the user to bypass the protections that are enforced by the call by value conventions of standard `PROCEDURE` definitions.

Example: We take our earlier `FACTORIAL` procedure and write it as a `LET` statement.

```
for all n let factorial n =
    begin scalar m,s;
      m:=1; s:=n;
11: if s=0 then return m;
      m:=m*s;
      s:=s-1;
      go to 11
```

```
end;
```

The reader will notice that we introduced a new local variable, *S*, and set it equal to *N*. The original form of the procedure contained the statement $n := n - 1$; . If the user asked for the value of `factorial(5)` then *N* would correspond to, not just have the value of, 5, and REDUCE would object to trying to execute the statement $5 := 5 - 1$.

If *PQR* is a procedure with no parameters,

```
procedure pqr;
  <procedure body>;
```

it can be written as a LET statement quite simply:

```
let pqr = <procedure body>;
```

To call *procedure PQR*, if defined in the latter form, the empty parentheses would not be used: use *PQR* not *PQR()* where a call on the procedure is needed.

The two notations for a procedure with no arguments can be combined. *PQR* can be defined in the standard PROCEDURE form. Then a LET statement

```
let pqr = pqr();
```

would allow a user to use *PQR* instead of *PQR()* in calling the procedure.

A feature available with LET-defined procedures and not with procedures defined in the standard way is the possibility of defining partial functions.

```
for all x such that numberp x let uvw(x)=<procedure body>;
```

Now *UVW* of an integer would be calculated as prescribed by the procedure body, while *UVW* of a general argument, such as *Z* or *p+q* (assuming these evaluate to themselves) would simply stay *uvw(z)* or *uvw(p+q)* as the case may be.

15.5 REMEMBER Statement

Setting the remember option for an algebraic procedure by

```
REMEMBER (PROCNAME:procedure);
```

saves all intermediate results of such procedure evaluations, including recursive calls. Subsequent calls to the procedure can then be determined from the saved results, and thus the number of evaluations (or the complexity) can be reduced. This mode of evaluation costs extra memory, of course. In addition, the procedure must be free of side-effects.

The following examples show the effect of the remember statement on two well-known examples.

```
procedure H(n);          % Hofstadter's function
  if numberp n then
    << cnn := cnn +1;    % counts the calls
    if n < 3 then 1 else H(n-H(n-1))+H(n-H(n-2))>>;
```

```
remember h;
```

```
<< cnn := 0; H(100); cnn>>;
```

```
100
```

```
% H has been called 100 times only.
```

```
procedure A(m,n);      % Ackermann function

  if m=0 then n+1 else
    if n=0 then A(m-1,1) else
      A(m-1,A(m,n-1));
```

```
remember a;
```

```
A(3,3);
```


Chapter 16

User Contributed Packages

The complete REDUCE system includes a number of packages contributed by users that are provided as a service to the user community. Questions regarding these packages should be directed to their individual authors.

All such packages have been precompiled as part of the installation process. However, many must be specifically loaded before they can be used. (Those that are loaded automatically are so noted in their description.) You should also consult the user notes for your particular implementation for further information on whether this is necessary. If it is, the relevant command is `LOAD_PACKAGE`, which takes a list of one or more package names as argument, for example:

```
load_package algint;
```

although this syntax may vary from implementation to implementation.

Nearly all these packages come with separate documentation and test files (except those noted here that have no additional documentation), which is included, along with the source of the package, in the REDUCE system distribution. These items should be studied for any additional details on the use of a particular package.

The packages available in the current release of REDUCE are as follows:

16.1 ALGINT: Integration of square roots

This package, which is an extension of the basic integration package distributed with REDUCE, will analytically integrate a wide range of expressions involving square roots where the answer exists in that class of functions. It is an implementation of the work described in J.H. Davenport, “On the Integration of Algebraic Functions”, LNCS 102, Springer Verlag, 1981. Both this and the source code should be consulted for a more detailed description of this work.

The ALGINT package is loaded automatically when the switch ALGINT is turned on. One enters an expression for integration, as with the regular integrator, for example:

```
int (sqrt (x+sqrt (x**2+1) ) /x, x) ;
```

If one later wishes to integrate expressions without using the facilities of this package, the switch ALGINT should be turned off.

The switches supported by the standard integrator (e.g., TRINT) are also supported by this package. In addition, the switch TRA, if on, will give further tracing information about the specific functioning of the algebraic integrator.

There is no additional documentation for this package.

Author: James H. Davenport.

16.2 APPLYSYM: Infinitesimal symmetries of differential equations

This package provides programs APPLYSYM, QUASILINPDE and DETRAFO for applying infinitesimal symmetries of differential equations, the generalization of special solutions and the calculation of symmetry and similarity variables.

Author: Thomas Wolf.

In this paper the programs APPLYSYM, QUASILINPDE and DETRAFO are described which aim at the utilization of infinitesimal symmetries of differential equations. The purpose of QUASILINPDE is the general solution of quasilinear PDEs. This procedure is used by APPLYSYM for the application of point symmetries for either

- calculating similarity variables to perform a point transformation which lowers the order of an ODE or effectively reduces the number of explicitly occurring independent variables in a PDE(-system) or for
- generalizing given special solutions of ODEs / PDEs with new constant parameters.

The program DETRAFO performs arbitrary point- and contact transformations of ODEs / PDEs and is applied if similarity and symmetry variables have been found. The program APPLYSYM is used in connection with the program LIEPDE for formulating and solving the conditions for point- and contact symmetries which is described in [4]. The actual problem solving is done in all these programs through a call to the package CRACK for solving overdetermined PDE-systems.

16.2.1 Introduction and overview of the symmetry method

The investigation of infinitesimal symmetries of differential equations (DEs) with computer algebra programs attracted considerable attention over the last years. Corresponding programs are available in all major computer algebra systems. In a review article by W. Hereman [1] about 200 references are given, many of them describing related software.

One reason for the popularity of the symmetry method is the fact that Sophus Lie's method [2],[3] is the most widely used method for computing exact solutions of non-linear DEs. Another reason is that the first step in this method, the formulation of the determining equation for the generators of the symmetries, can already be very cumbersome, especially in the case of PDEs of higher order and/or in case of many dependent and independent variables. Also, the formulation of the conditions

is a straight forward task involving only differentiations and basic algebra - an ideal task for computer algebra systems. Less straight forward is the automatic solution of the symmetry conditions which is the strength of the program `LIEPDE` (for a comparison with another program see [4]).

The novelty described in this paper are programs aiming at the final third step: Applying symmetries for

- calculating similarity variables to perform a point transformation which lowers the order of an ODE or effectively reduces the number of explicitly occurring independent variables of a PDE(-system) or for
- generalizing given special solutions of ODEs/PDEs with new constant parameters.

Programs which run on their own but also allow interactive user control are indispensable for these calculations. On one hand the calculations can become quite lengthy, like variable transformations of PDEs (of higher order, with many variables). On the other hand the freedom of choosing the right linear combination of symmetries and choosing the optimal new symmetry- and similarity variables makes it necessary to ‘play’ with the problem interactively.

The focus in this paper is directed on questions of implementation and efficiency, no principally new mathematics is presented.

In the following subsections a review of the first two steps of the symmetry method is given as well as the third, i.e. the application step is outlined. Each of the remaining sections is devoted to one procedure.

The first step: Formulating the symmetry conditions

To obey classical Lie-symmetries, differential equations

$$H_A = 0 \quad (16.1)$$

for unknown functions y^α , $1 \leq \alpha \leq p$ of independent variables x^i , $1 \leq i \leq q$ must be forminvariant against infinitesimal transformations

$$\tilde{x}^i = x^i + \varepsilon \xi^i, \quad \tilde{y}^\alpha = y^\alpha + \varepsilon \eta^\alpha \quad (16.2)$$

in first order of ε . To transform the equations (16.1) by (16.2), derivatives of y^α must be transformed, i.e. the part linear in ε must be determined. The corresponding formulas are (see e.g. [10], [20])

$$\begin{aligned} \tilde{y}_{j_1 \dots j_k}^\alpha &= y_{j_1 \dots j_k}^\alpha + \varepsilon \eta_{j_1 \dots j_k}^\alpha + O(\varepsilon^2) \\ \eta_{j_1 \dots j_{k-1} j_k}^\alpha &= \frac{D \eta_{j_1 \dots j_{k-1}}^\alpha}{D x^k} - y_{i j_1 \dots j_{k-1}}^\alpha \frac{D \xi^i}{D x^k} \end{aligned} \quad (16.3)$$

where D/Dx^k means total differentiation w.r.t. x^k and from now on lower latin indices of functions y^α , (and later u^α) denote partial differentiation w.r.t. the independent variables x^i , (and later v^i). The complete symmetry condition then takes the form

$$XH_A = 0 \mod H_A = 0 \quad (16.4)$$

$$X = \xi^i \frac{\partial}{\partial x^i} + \eta^\alpha \frac{\partial}{\partial y^\alpha} + \eta_m^\alpha \frac{\partial}{\partial y_m^\alpha} + \eta_{mn}^\alpha \frac{\partial}{\partial y_{mn}^\alpha} + \dots + \eta_{mn\dots p}^\alpha \frac{\partial}{\partial y_{mn\dots p}^\alpha} \quad (16.5)$$

where $\mod H_A = 0$ means that the original PDE-system is used to replace some partial derivatives of y^α to reduce the number of independent variables, because the symmetry condition (16.4) must be fulfilled identically in x^i, y^α and all partial derivatives of y^α .

For point symmetries, ξ^i, η^α are functions of x^j, y^β and for contact symmetries they depend on x^j, y^β and y_k^β . We restrict ourself to point symmetries as those are the only ones that can be applied by the current version of the program `APPLYSYM` (see below). For literature about generalized symmetries see [1].

Though the formulation of the symmetry conditions (16.4), (16.5), (16.3) is straightforward and handled in principle by all related programs [1], the computational effort to formulate the conditions (16.4) may cause problems if the number of x^i and y^α is high. This can partially be avoided if at first only a few conditions are formulated and solved such that the remaining ones are much shorter and quicker to formulate.

A first step in this direction is to investigate one PDE $H_A = 0$ after another, as done in [22]. Two methods to partition the conditions for a single PDE are described by Bocharov/Bronstein [9] and Stephani [20].

In the first method only those terms of the symmetry condition $XH_A = 0$ are calculated which contain at least a derivative of y^α of a minimal order m . Setting coefficients of these u -derivatives to zero provides symmetry conditions. Lowering the minimal order m successively then gradually provides all symmetry conditions.

The second method is even more selective. If H_A is of order n then only terms of the symmetry condition $XH_A = 0$ are generated which contain n' th order derivatives of y^α . Furthermore these derivatives must not occur in H_A itself. They can therefore occur in the symmetry condition (16.4) only in $\eta_{j_1\dots j_n}^\alpha$, i.e. in the terms

$$\eta_{j_1\dots j_n}^\alpha \frac{\partial H_A}{\partial y_{j_1\dots j_n}^\alpha}.$$

If only coefficients of n' th order derivatives of y^α need to be accurate to formulate preliminary conditions then from the total derivatives to be taken in (16.3) only that part is performed which differentiates w.r.t. the highest y^α -derivatives. This means, for example, to form only $y_{mnk}^\alpha \partial/\partial y_{mn}^\alpha$ if the expression, which is to be differentiated totally w.r.t. x^k , contains at most second order derivatives of y^α .

The second method is applied in `LIEPDE`. Already the formulation of the remaining conditions is speeded up considerably through this iteration process. These methods can be applied if systems of DEs or single PDEs of at least second order are investigated concerning symmetries.

The second step: Solving the symmetry conditions

The second step in applying the whole method consists in solving the determining conditions (16.4), (16.5), (16.3) which are linear homogeneous PDEs for ξ^i, η^α . The complete solution of this system is not algorithmic any more because the solution of a general linear PDE-system is as difficult as the solution of its non-linear characteristic ODE-system which is not covered by algorithms so far.

Still algorithms are used successfully to simplify the PDE-system by calculating its standard normal form and by integrating exact PDEs if they turn up in this simplification process [4]. One problem in this respect, for example, concerns the optimization of the symbiosis of both algorithms. By that we mean the ranking of priorities between integrating, adding integrability conditions and doing simplifications by substitutions - all depending on the length of expressions and the overall structure of the PDE-system. Also the extension of the class of PDEs which can be integrated exactly is a problem to be pursued further.

The program `LIEPDE` which formulates the symmetry conditions calls the program `CRACK` to solve them. This is done in a number of successive calls in order to formulate and solve some first order PDEs of the overdetermined system first and use their solution to formulate and solve the next subset of conditions as described in the previous subsection. Also, `LIEPDE` can work on DEs that contain parametric constants and parametric functions. An ansatz for the symmetry generators can be formulated. For more details see [4] or [17].

The procedure `LIEPDE` is called through
`LIEPDE (problem,symtype,flist,inequ) ;`
 All parameters are lists.

The first parameter specifies the DEs to be investigated:
`problem` has the form $\{equations, ulist, xlist\}$ where

- `equations` is a list of equations, each has the form $df(ui, \dots) = \dots$ where the LHS (left hand side) $df(ui, \dots)$ is selected such that
 - The RHS (right h.s.) of an equations must not include the derivative on the LHS nor a derivative of it.
 - Neither the LHS nor any derivative of it of any equation may occur in any other equation.
 - Each of the unknown functions occurs on the LHS of exactly one equation.

ulist is a list of function names, which can be chosen freely
xlist is a list of variable names, which can be chosen freely

Equations can be given as a list of single differential expressions and then the program will try to bring them into the 'solved form' $\text{df}(u_i, \dots) = \dots$ automatically. If equations are given in the solved form then the above conditions are checked and execution is stopped if they are not satisfied. An easy way to get the equations in the desired form is to use

```
FIRST SOLVE ({eq1,eq2,...}, {one highest derivative for each function
u})
```

(see the example of the Karpman equations in `LIEPDE.TST`). The example of the Burgers equation in `LIEPDE.TST` demonstrates that the number of symmetries for a given maximal order of the infinitesimal generators depends on the derivative chosen for the LHS.

The second parameter *symtype* of `LIEPDE` is a list `{ }` that specifies the symmetry to be calculated. *symtype* can have the following values and meanings:

<code>{"point"}</code>	Point symmetries with $\xi^i = \xi^i(x^j, u^\beta)$, $\eta^\alpha = \eta^\alpha(x^j, u^\beta)$ are determined.
<code>{"contact"}</code>	Contact symmetries with $\xi^i = 0$, $\eta = \eta(x^j, u, u_k)$ are determined ($u_k = \partial u / \partial x^k$), which is only applicable if a single equation (16.1) with an order > 1 for a single function u is to be investigated. (The <i>symtype</i> <code>{"contact"}</code> is equivalent to <code>{"general", 1}</code> (see below) apart from the additional checks done for <code>{"contact"}</code> .)
<code>{"general", order}</code>	where <i>order</i> is an integer > 0 . Generalized symmetries $\xi^i = 0$, $\eta^\alpha = \eta^\alpha(x^j, u^\beta, \dots, u_K^\beta)$ of a specified order are determined (where K is a multiple index representing <i>order</i> many indices.) NOTE: Characteristic functions of generalized symmetries ($= \eta^\alpha$ if $\xi^i = 0$) are equivalent if they are equal on the solution manifold. Therefore, all dependences of characteristic functions on the substituted derivatives and their derivatives are dropped. For example, if the heat equation is given as $u_t = u_{xx}$ (i.e. u_t is substituted by u_{xx}) then <code>{"general", 2}</code> would not include characteristic functions depending on u_{tx} or u_{xxx} .

THEREFORE:

If you want to find *all* symmetries up to a given order then either
- avoid using $H_A = 0$ to substitute lower order derivatives by expressions involving higher derivatives, or
- increase the order specified in *symtype*.

For an illustration of this effect see the two symmetry determinations of the Burgers equation in the file

```
LIEPDE.TST.
{xi!_xl = ..., ...,
 eta!_ul = ..., ...}
```

LIEPDE.TST.

It is possible to specify an ansatz for the symmetry. Such an ansatz must specify all ξ^i for all independent variables and all η^α for all dependent variables in terms of differential expressions which may involve unknown functions/constants. The dependences of the unknown functions have to be declared in advance by using the `DEPEND` command. For example,

```
DEPEND f, t, x, u$
```

specifies f to be a function of t, x, u . If one wants to have f as a function of derivatives of $u(t, x)$, say f depending on u_{txx} , then one cannot write

```
DEPEND f, df(u, t, x, 2) $
```

but instead must write

```
DEPEND f, u!`1!`2!`2$
```

assuming $xlist$ has been specified as $\{t, x\}$. Because t is the first variable and x is the second variable in $xlist$ and u is differentiated once wrt. t and twice wrt. x we therefore use `u!`1!`2!`2`. The character `!` is the escape character to allow special characters like `'` to occur in an identifier.

For generalized symmetries one usually sets all $\xi^i = 0$. Then the η^α are equal to the characteristic functions.

The third parameter *flist* of `LIEPDE` is a list $\{ \}$ that includes

- all parameters and functions in the equations which are to be determined such that symmetries exist (if any such parameters/functions are specified in *flist* then the symmetry conditions formulated in `LIEPDE` become non-linear conditions which may be much harder for `CRACK` to solve with many cases and subcases to be considered.)
- all unknown functions and constants in the ansatz `xi!_...` and `eta!_...` if that has been specified in *syntype*.

The fourth parameter *inequ* of `LIEPDE` is a list $\{ \}$ that includes all non-vanishing expressions which represent inequalities for the functions in *flist*.

The result of `LIEPDE` is a list with 3 elements, each of which is a list:

$$\{\{con_1, con_2, \dots\}, \{xi_{-...} = \dots, \dots, eta_{-...} = \dots, \dots\}, \{flist\}\}.$$

The first list contains remaining unsolved symmetry conditions con_i . It is the empty list $\{ \}$ if all conditions have been solved. The second list gives the symmetry generators, i.e. expressions for ξ_i and η_j . The last list contains all free constants and functions occurring in the first and second list.

The third step: Application of infinitesimal symmetries

If infinitesimal symmetries have been found then the program `APPLYSYM` can use them for the following purposes:

1. Calculation of one symmetry variable and further similarity variables. After transforming the DE(-system) to these variables, the symmetry variable will not occur explicitly any more. For ODEs this has the consequence that their order has effectively been reduced.
2. Generalization of a special solution by one or more constants of integration.

Both methods are described in the following section.

16.2.2 Applying symmetries with `APPLYSYM`

The first mode: Calculation of similarity and symmetry variables

In the following we assume that a symmetry generator X , given in (16.5), is known such that ODE(s)/PDE(s) $H_A = 0$ satisfy the symmetry condition (16.4). The aim is to find new dependent functions $u^\alpha = u^\alpha(x^j, y^\beta)$ and new independent variables $v^i = v^i(x^j, y^\beta)$, $1 \leq \alpha, \beta \leq p$, $1 \leq i, j \leq q$ such that the symmetry generator $X = \xi^i(x^j, y^\beta)\partial_{x^i} + \eta^\alpha(x^j, y^\beta)\partial_{y^\alpha}$ transforms to

$$X = \partial_{v^1}. \quad (16.6)$$

Inverting the above transformation to $x^i = x^i(v^j, u^\beta)$, $y^\alpha = y^\alpha(v^j, u^\beta)$ and setting $H_A(x^i(v^j, u^\beta), y^\alpha(v^j, u^\beta), \dots) = h_A(v^j, u^\beta, \dots)$ this means that

$$\begin{aligned} 0 &= XH_A(x^i, y^\alpha, y_j^\beta, \dots) \mod H_A = 0 \\ &= Xh_A(v^i, u^\alpha, u_j^\beta, \dots) \mod h_A = 0 \\ &= \partial_{v^1}h_A(v^i, u^\alpha, u_j^\beta, \dots) \mod h_A = 0. \end{aligned}$$

Consequently, the variable v^1 does not occur explicitly in h_A . In the case of an ODE(-system) ($v^1 = v$) the new equations $0 = h_A(v, u^\alpha, du^\beta/dv, \dots)$ are then of lower total order after the transformation $z = z(u^1) = du^1/dv$ with now z, u^2, \dots, u^p as unknown functions and u^1 as independent variable.

The new form (16.6) of X leads directly to conditions for the symmetry variable v^1 and the similarity variables $v^i|_{i \neq 1}, u^\alpha$ (all functions of x^k, y^γ):

$$Xv^1 = 1 = \xi^i(x^k, y^\gamma)\partial_{x^i}v^1 + \eta^\alpha(x^k, y^\gamma)\partial_{y^\alpha}v^1 \quad (16.7)$$

$$Xv^j|_{j \neq 1} = Xu^\beta = 0 = \xi^i(x^k, y^\gamma)\partial_{x^i}v^j + \eta^\alpha(x^k, y^\gamma)\partial_{y^\alpha}v^j \quad (16.8)$$

The general solutions of (16.7), (16.8) involve free functions of $p+q-1$ arguments. From the general solution of equation (16.8), $p+q-1$ functionally independent special solutions have to be selected (v^2, \dots, v^p and u^1, \dots, u^q), whereas from (16.7) only one solution v^1 is needed. Together, the expressions for the symmetry and similarity variables must define a non-singular transformation $x, y \rightarrow u, v$.

Different special solutions selected at this stage will result in different resulting DEs which are equivalent under point transformations but may look quite differently. A transformation that is more difficult than another one will in general only complicate the new DE(s) compared with the simpler transformation. We therefore seek the simplest possible special solutions of (16.7), (16.8). They also have to be simple because the transformation has to be inverted to solve for the old variables in order to do the transformations.

The following steps are performed in the corresponding mode of the program APPLYSYM:

- The user is asked to specify a symmetry by selecting one symmetry from all the known symmetries or by specifying a linear combination of them.
- Through a call of the procedure QUASILINPDE (described in a later section) the two linear first order PDEs (16.7), (16.8) are investigated and, if possible, solved.
- From the general solution of (16.7) 1 special solution is selected and from (16.8) $p+q-1$ special solutions are selected which should be as simple as possible.
- The user is asked whether the symmetry variable should be one of the independent variables (as it has been assumed so far) or one of the new functions (then only derivatives of this function and not the function itself turn up in the new DE(s)).
- Through a call of the procedure DETRAFO the transformation $x^i, y^\alpha \rightarrow v^j, u^\beta$ of the DE(s) $H_A = 0$ is finally done.
- The program returns to the starting menu.

The second mode: Generalization of special solutions

A second application of infinitesimal symmetries is the generalization of a known special solution given in implicit form through $0 = F(x^i, y^\alpha)$. If one knows a symmetry variable v^1 and similarity variables v^r, u^α , $2 \leq r \leq p$ then v^1 can be shifted by a constant c because of $\partial_{v^1} H_A = 0$ and therefore the DEs $0 = H_A(v^r, u^\alpha, u_j^\beta, \dots)$ are unaffected by the shift. Hence from

$$0 = F(x^i, y^\alpha) = F(x^i(v^j, u^\beta), y^\alpha(v^j, u^\beta)) = \bar{F}(v^j, u^\beta)$$

follows that

$$0 = \bar{F}(v^1 + c, v^r, u^\beta) = \bar{F}(v^1(x^i, y^\alpha) + c, v^r(x^i, y^\alpha), u^\beta(x^i, y^\alpha))$$

defines implicitly a generalized solution $y^\alpha = y^\alpha(x^i, c)$.

This generalization works only if $\partial_{v^1} \bar{F} \neq 0$ and if \bar{F} does not already have a constant additive to v^1 .

The method above needs to know $x^i = x^i(u^\beta, v^j)$, $y^\alpha = y^\alpha(u^\beta, v^j)$ and $u^\alpha = u^\alpha(x^j, y^\beta)$, $v^\alpha = v^\alpha(x^j, y^\beta)$ which may be practically impossible. Better is, to integrate x^i, y^α along X :

$$\frac{d\bar{x}^i}{d\varepsilon} = \xi^i(\bar{x}^j(\varepsilon), \bar{y}^\beta(\varepsilon)), \quad \frac{d\bar{y}^\alpha}{d\varepsilon} = \eta^\alpha(\bar{x}^j(\varepsilon), \bar{y}^\beta(\varepsilon)) \quad (16.9)$$

with initial values $\bar{x}^i = x^i, \bar{y}^\alpha = y^\alpha$ for $\varepsilon = 0$. (This ODE-system is the characteristic system of (16.8).)

Knowing only the finite transformations

$$\bar{x}^i = \bar{x}^i(x^j, y^\beta, \varepsilon), \quad \bar{y}^\alpha = \bar{y}^\alpha(x^j, y^\beta, \varepsilon) \quad (16.10)$$

gives immediately the inverse transformation $\bar{x}^i = \bar{x}^i(x^j, y^\beta, \varepsilon)$, $\bar{y}^\alpha = \bar{y}^\alpha(x^j, y^\beta, \varepsilon)$ just by $\varepsilon \rightarrow -\varepsilon$ and renaming $x^i, y^\alpha \leftrightarrow \bar{x}^i, \bar{y}^\alpha$.

The special solution $0 = F(x^i, y^\alpha)$ is generalized by the new constant ε through

$$0 = F(x^i, y^\alpha) = F(x^i(\bar{x}^j, \bar{y}^\beta, \varepsilon), y^\alpha(\bar{x}^j, \bar{y}^\beta, \varepsilon))$$

after dropping the $\bar{}$.

The steps performed in the corresponding mode of the program APPLYSYM show features of both techniques:

- The user is asked to specify a symmetry by selecting one symmetry from all the known symmetries or by specifying a linear combination of them.
- The special solution to be generalized and the name of the new constant have to be put in.
- Through a call of the procedure QUASILINPDE, the PDE (16.7) is solved which amounts to a solution of its characteristic ODE system (16.9) where $v^1 = \varepsilon$.
- QUASILINPDE returns a list of constant expressions

$$c_i = c_i(x^k, y^\beta, \varepsilon), \quad 1 \leq i \leq p + q \quad (16.11)$$

which are solved for $x^j = x^j(c_i, \varepsilon)$, $y^\alpha = y^\alpha(c_i, \varepsilon)$ to obtain the generalized solution through

$$0 = F(x^j, y^\alpha) = F(x^j(c_i(x^k, y^\beta, 0), \varepsilon), y^\alpha(c_i(x^k, y^\beta, 0), \varepsilon)).$$

- The new solution is available for further generalizations w.r.t. other symmetries.

If one would like to generalize a given special solution with m new constants because m symmetries are known, then one could run the whole program m times, each time with a different symmetry or one could run the program once with a linear combination of m symmetry generators which again is a symmetry generator. Running the program once adds one constant but we have in addition $m - 1$ arbitrary constants in the linear combination of the symmetries, so m new constants are added. Usually one will generalize the solution gradually to make solving (16.9) gradually more difficult.

Syntax

The call of `APPLYSYM` is `APPLYSYM({de, fun, var}, {sym, cons});`

- *de* is a single DE or a list of DEs in the form of a vanishing expression or in the form $\dots = \dots$.
- *fun* is the single function or the list of functions occurring in *de*.
- *var* is the single variable or the list of variables in *de*.
- *sym* is a linear combination of all symmetries, each with a different constant coefficient, in form of a list of the ξ^i and η^α : $\{\text{xi}_{\dots} = \dots, \dots, \text{eta}_{\dots} = \dots, \dots\}$, where the indices after ‘xi_’ are the variable names and after ‘eta_’ the function names.
- *cons* is the list of constants in *sym*, one constant for each symmetry.

The list that is the first argument of `APPLYSYM` is the same as the first argument of `LIEPDE` and the second argument is the list that `LIEPDE` returns without its first element (the unsolved conditions). An example is given below.

What `APPLYSYM` returns depends on the last performed modus. After modus 1 the return is

$\{\{newde, newfun, newvar\}, trafo\}$
where

- *newde* lists the transformed equation(s)
- *newfun* lists the new function name(s)
- *newvar* lists the new variable name(s)
- *trafo* lists the transformations $x^i = x^i(v^j, u^\beta)$, $y^\alpha = y^\alpha(v^j, u^\beta)$

After modus 2, `APPLYSYM` returns the generalized special solution.

Example: A second order ODE

Weyl's class of solutions of Einsteins field equations consists of axialsymmetric time independent metrics of the form

$$ds^2 = e^{-2U} \left[e^{2k} (d\rho^2 + dz^2) + \rho^2 d\varphi^2 \right] - e^{2U} dt^2, \quad (16.12)$$

where U and k are functions of ρ and z . If one is interested in generalizing these solutions to have a time dependence then the resulting DEs can be transformed such that one longer third order ODE for U results which contains only ρ derivatives [23]. Because U appears not alone but only as derivative, a substitution

$$g = dU/d\rho \quad (16.13)$$

lowers the order and the introduction of a function

$$h = \rho g - 1 \quad (16.14)$$

simplifies the ODE to

$$0 = 3\rho^2 h h'' - 5\rho^2 h'^2 + 5\rho h h' - 20\rho h^3 h' - 20 h^4 + 16 h^6 + 4 h^2. \quad (16.15)$$

where $' = d/d\rho$. Calling LIEPDE through

```
depend h, r;
prob:={-20*h**4+16*h**6+3*r**2*h*df(h, r, 2)+5*r*h*df(h, r)
        -20*h**3*r*df(h, r)+4*h**2-5*r**2*df(h, r)**2},
        {h}, {r}};
sym:=liepde(prob, {"point"}, {}, {});
end;
```

gives

```
sym := {}, {xi_r= - c10*r3 - c11*r, eta_h=c10*h*r2 }, {c10, c11}}.
```

All conditions have been solved because the first element of `sym` is `{}`. The two existing symmetries are therefore

$$-\rho^3 \partial_\rho + h \rho^2 \partial_h \quad \text{and} \quad \rho \partial_\rho. \quad (16.16)$$

Corresponding finite transformations can be calculated with APPLYSYM through

```
newde:=appliesym(prob, rest sym);
```

The interactive session is given below with the user input following the prompt 'Input:3:' or following '?'. (Empty lines have been deleted.)

Do you want to find similarity and symmetry variables (enter '1;')
 or generalize a special solution with new parameters (enter '2;')
 or exit the program (enter ';')
 Input:3: 1;

We enter '1;' because we want to reduce dependencies by finding similarity variables and one symmetry variable and then doing the transformation such that the symmetry variable does not explicitly occur in the DE.

```
----- The 1. symmetry is:
          3
xi_r= - r
          2
eta_h=h*r
----- The 2. symmetry is:
xi_r= - r
-----
Which single symmetry or linear combination of symmetries
do you want to apply?
Enter an expression with 'sy_(i)' for the i'th symmetry.
sy_(1);
```

We could have entered 'sy_(2);' or a combination of both as well with the calculation running then differently.

```
The symmetry to be applied in the following is
          3          2
{xi_r= - r ,eta_h=h*r }
Enter the name of the new dependent variables:
Input:3: u;
Enter the name of the new independent variables:
Input:3: v;
```

This was the input part, now the real calculation starts.

```
The ODE/PDE (-system) under investigation is :
          2          2 2          3
0 = 3*df(h,r,2)*h*r - 5*df(h,r) *r - 20*df(h,r)*h *r
          6          4          2
+ 5*df(h,r)*h*r + 16*h - 20*h + 4*h
for the function(s) : h.
It will be looked for a new dependent variable u
and an independent variable v such that the transformed
de(-system) does not depend on u or v.
1. Determination of the similarity variable
          2
The quasilinear PDE: 0 = r *(df(u_,h)*h - df(u_,r)*r) .
The equivalent characteristic system:
          3
```

```

0= - df(u_,r)*r
      2
0= - r*(df(h,r)*r + h)
for the functions: h(r)  u_(r).

```

The PDE is equation (16.8).

The general solution of the PDE is given through

```

0 = ff(u_,h*r)
with arbitrary function ff(..).
A suggestion for this function ff provides:
0 = - h*r + u_
Do you like this choice? (Y or N)
?y

```

For the following calculation only a single special solution of the PDE is necessary and this has to be specified from the general solution by choosing a special function `ff`. (This function is called `ff` to prevent a clash with names of user variables/functions.) In principle any choice of `ff` would work, if it defines a non-singular coordinate transformation, i.e. here r must be a function of u_- . If we have q independent variables and p functions of them then `ff` has $p + q$ arguments. Because of the condition $0 = \text{ff}$ one has essentially the freedom of choosing a function of $p + q - 1$ arguments freely. This freedom is also necessary to select $p + q - 1$ different functions `ff` and to find as many functionally independent solutions u_- which all become the new similarity variables. q of them become the new functions u^α and $p - 1$ of them the new variables v^2, \dots, v^p . Here we have $p = q = 1$ (one single ODE).

Though the program could have done that alone, once the general solution `ff(..)` is known, the user can interfere here to enter a simpler solution, if possible.

2. Determination of the symmetry variable

```

      2      3
The quasilinear PDE: 0 = df(u_,h)*h*r - df(u_,r)*r - 1.
The equivalent characteristic system:

```

```

      3
0=df(r,u_) + r
      2
0=df(h,u_) - h*r
for the functions: r(u_)  h(u_)  .
New attempt with a different independent variable
The equivalent characteristic system:
      2
0=df(u_,h)*h*r - 1
      2
0=r*(df(r,h)*h + r)
for the functions: r(h)  u_(h)  .
The general solution of the PDE is given through

```

```

          2 2      2
        - 2*h *r *u_ + h
0 = ff(h*r,-----)
          2
with arbitrary function ff(..).
A suggestion for this function ff(..) yields:
          2      2
        h *( - 2*r *u_ + 1)
0 = -----
          2
Do you like this choice? (Y or N)
?y

```

Similar to above.

The suggested solution of the algebraic system which will do the transformation is:

```

                                sqrt(v)*sqrt(2)
{h=sqrt(v)*sqrt(2)*u,r=-----}
                                2*v

```

Is the solution ok? (Y or N)

?y

In the intended transformation shown above the dependent variable is u and the independent variable is v .

The symmetry variable is v , i.e. the transformed expression will be free of v .

Is this selection of dependent and independent variables ok? (Y or N)

?n

We so far assumed that the symmetry variable is one of the new variables, but, of course we also could choose it to be one of the new functions. If it is one of the functions then only derivatives of this function occur in the new DE, not the function itself. If it is one of the variables then this variable will not occur explicitly.

In our case we prefer (without strong reason) to have the function as symmetry variable. We therefore answered with 'no'. As a consequence, u and v will exchange names such that still all new functions have the name u and the new variables have name v :

```

Please enter a list of substitutions. For example, to
make the variable, which is so far call u1, to an
independent variable v2 and the variable, which is
so far called v2, to an dependent variable u1,
enter: '{u1=v2, v2=u1};'
Input:3: {u=v,v=u};

```

The transformed equation which should be free of u :

```

          3 6      2 3

```



```

0=3*df(u,v,2)*v - 16*df(u,v) *v - 20*df(u,v) *v + 5*df(u,v)
Do you want to find similarity and symmetry variables (enter '1;')
or generalize a special solution with new parameters (enter '2;')
or exit the program (enter ';'')
Input:3: ;

```

We stop here. The following is returned from our APPLYSYM call:

```

          3  6          2  3
{{{3*df(u,v,2)*v - 16*df(u,v) *v - 20*df(u,v) *v + 5*df(u,v)},
 {u},
 {v}},
 sqrt(u)*sqrt(2)
{r=-----, h=sqrt(u)*sqrt(2)*v }}
      2*u

```

The use of APPLYSYM effectively provided us the finite transformation

$$\rho = (2u)^{-1/2}, \quad h = (2u)^{1/2}v. \quad (16.17)$$

and the new ODE

$$0 = 3u''v - 16u'^3v^6 - 20u'^2v^3 + 5u' \quad (16.18)$$

where $u = u(v)$ and $' = d/dv$. Using one symmetry we reduced the 2. order ODE (16.15) to a first order ODE (16.18) for u' plus one integration. The second symmetry can be used to reduce the remaining ODE to an integration too by introducing a variable w through $v^3d/dv = d/dw$, i.e. $w = -1/(2v^2)$. With

$$p = du/dw \quad (16.19)$$

the remaining ODE is

$$0 = 3w \frac{dp}{dw} + 2p(p+1)(4p+1)$$

with solution

$$\tilde{c}w^{-2}/4 = \tilde{c}v^4 = \frac{p^3(p+1)}{(4p+1)^4}, \quad \tilde{c} = const.$$

Writing (16.19) as $p = v^3(du/dp)/(dv/dp)$ we get u by integration and with (16.17) further a parametric solution for ρ, h :

$$\rho = \left(\frac{3c_1^2(2p-1)}{p^{1/2}(p+1)^{1/2}} + c_2 \right)^{-1/2} \quad (16.20)$$

$$h = \frac{(c_2p^{1/2}(p+1)^{1/2} + 6c_1^2p - 3c_1^2)^{1/2}p^{1/2}}{c_1(4p+1)} \quad (16.21)$$

where $c_1, c_2 = const.$ and $c_1 = \tilde{c}^{1/4}$. Finally, the metric function $U(p)$ is obtained as an integral from (16.13), (16.14).

Limitations of `APPLYSYM`

Restrictions of the applicability of the program `APPLYSYM` result from limitations of the program `QUASILINPDE` described in a section below. Essentially this means that symmetry generators may only be polynomially non-linear in x^i, y^α . Though even then the solvability can not be guaranteed, the generators of Lie-symmetries are mostly very simple such that the resulting PDE (16.22) and the corresponding characteristic ODE-system have good chances to be solvable.

Apart from these limitations implied through the solution of differential equations with `CRACK` and algebraic equations with `SOLVE` the program `APPLYSYM` itself is free of restrictions, i.e. if once new versions of `CRACK`, `SOLVE` would be available then `APPLYSYM` would not have to be changed.

Currently, whenever a computational step could not be performed the user is informed and has the possibility of entering interactively the solution of the unsolved algebraic system or the unsolved linear PDE.

16.2.3 Solving quasilinear PDEs

The content of `QUASILINPDE`

The generalization of special solutions of DEs as well as the computation of similarity and symmetry variables involve the general solution of single first order linear PDEs. The procedure `QUASILINPDE` is a general procedure aiming at the general solution of PDEs

$$a_1(w_i, \phi)\phi_{w_1} + a_2(w_i, \phi)\phi_{w_2} + \dots + a_n(w_i, \phi)\phi_{w_n} = b(w_i, \phi) \quad (16.22)$$

in n independent variables $w_i, i = 1 \dots n$ for one unknown function $\phi = \phi(w_i)$.

1. The first step in solving a quasilinear PDE (16.22) is the formulation of the corresponding characteristic ODE-system

$$\frac{dw_i}{d\varepsilon} = a_i(w_j, \phi) \quad (16.23)$$

$$\frac{d\phi}{d\varepsilon} = b(w_j, \phi) \quad (16.24)$$

for ϕ, w_i regarded now as functions of one variable ε .

Because the a_i and b do not depend explicitly on ε , one of the equations (16.23), (16.24) with non-vanishing right hand side can be used to divide all others through it and by that having a system with one less ODE to solve. If the equation to divide through is one of (16.23) then the remaining system

would be

$$\frac{dw_i}{dw_k} = \frac{a_i}{a_k}, \quad i = 1, 2, \dots, k-1, k+1, \dots, n \quad (16.25)$$

$$\frac{d\phi}{dw_k} = \frac{b}{a_k} \quad (16.26)$$

with the independent variable w_k instead of ε . If instead we divide through equation (16.24) then the remaining system would be

$$\frac{dw_i}{d\phi} = \frac{a_i}{b}, \quad i = 1, 2, \dots, n \quad (16.27)$$

with the independent variable ϕ instead of ε .

The equation to divide through is chosen by a subroutine with a heuristic to find the “simplest” non-zero right hand side (a_k or b), i.e. one which

- is constant or
- depends only on one variable or
- is a product of factors, each of which depends only on one variable.

One purpose of this division is to reduce the number of ODEs by one. Secondly, the general solution of (16.23), (16.24) involves an additive constant to ε which is not relevant and would have to be set to zero. By dividing through one ODE we eliminate ε and lose the problem of identifying this constant in the general solution before we would have to set it to zero.

2. To solve the system (16.25), (16.26) or (16.27), the procedure CRACK is called. Although being designed primarily for the solution of overdetermined PDE-systems, CRACK can also be used to solve simple not overdetermined ODE-systems. This solution process is not completely algorithmic. Improved versions of CRACK could be used, without making any changes of QUASILINPDE necessary.

If the characteristic ODE-system can not be solved in the form (16.25), (16.26) or (16.27) then successively all other ODEs of (16.23), (16.24) with non-vanishing right hand side are used for division until one is found such that the resulting ODE-system can be solved completely. Otherwise the PDE can not be solved by QUASILINPDE.

3. If the characteristic ODE-system (16.23), (16.24) has been integrated completely and in full generality to the implicit solution

$$0 = G_i(\phi, w_j, c_k, \varepsilon), \quad i, k = 1, \dots, n+1, \quad j = 1, \dots, n \quad (16.28)$$

then according to the general theory for solving first order PDEs, ε has to be eliminated from one of the equations and to be substituted in the others to

have left n equations. Also the constant that turns up additively to ε is to be set to zero. Both tasks are automatically fulfilled, if, as described above, ε is already eliminated from the beginning by dividing all equations of (16.23), (16.24) through one of them.

On either way one ends up with n equations

$$0 = g_i(\phi, w_j, c_k), \quad i, j, k = 1 \dots n \quad (16.29)$$

involving n constants c_k .

The final step is to solve (16.29) for the c_i to obtain

$$c_i = c_i(\phi, w_1, \dots, w_n) \quad i = 1, \dots, n. \quad (16.30)$$

The final solution $\phi = \phi(w_i)$ of the PDE (16.22) is then given implicitly through

$$0 = F(c_1(\phi, w_i), c_2(\phi, w_i), \dots, c_n(\phi, w_i))$$

where F is an arbitrary function with n arguments.

Syntax

The call of QUASILINPDE is

QUASILINPDE(*de*, *fun*, *varlist*);

- *de* is the differential expression which vanishes due to the PDE $de = 0$ or, *de* may be the differential equation itself in the form $\dots = \dots$.
- *fun* is the unknown function.
- *varlist* is the list of variables of *fun*.

The result of QUASILINPDE is a list of general solutions

$$\{sol_1, sol_2, \dots\}.$$

If QUASILINPDE can not solve the PDE then it returns $\{\}$. Each solution sol_i is a list of expressions

$$\{ex_1, ex_2, \dots\}$$

such that the dependent function (ϕ in (16.22)) is determined implicitly through an arbitrary function F and the algebraic equation

$$0 = F(ex_1, ex_2, \dots).$$

Examples

Example 1:

To solve the quasilinear first order PDE

$$1 = xu_{,x} + uu_{,y} - zu_{,z}$$

for the function $u = u(x, y, z)$, the input would be

```
depend u, x, y, z;
de:=x*df(u,x)+u*df(u,y)-z*df(u,z) - 1;
varlist:={x,y,z};
QUASILINPDE(de,u,varlist);
```

In this example the procedure returns

$$\{\{x/e^u, ze^u, u^2 - 2y\}\},$$

i.e. there is one general solution (because the outer list has only one element which itself is a list) and u is given implicitly through the algebraic equation

$$0 = F(x/e^u, ze^u, u^2 - 2y)$$

with arbitrary function F .

Example 2:

For the linear inhomogeneous PDE

$$0 = yz_{,x} + xz_{,y} - 1, \quad \text{for } z = z(x, y)$$

QUASILINPDE returns the result that for an arbitrary function F , the equation

$$0 = F\left(\frac{x+y}{e^z}, e^z(x-y)\right)$$

defines the general solution for z .

Example 3:

For the linear inhomogeneous PDE (3.8) from [15]

$$0 = xw_{,x} + (y+z)(w_{,y} - w_{,z}), \quad \text{for } w = w(x, y, z)$$

QUASILINPDE returns the result that for an arbitrary function F , the equation

$$0 = F(w, y+z, \ln(x)(y+z) - y)$$

defines the general solution for w , i.e. for any function f

$$w = f(y+z, \ln(x)(y+z) - y)$$

solves the PDE.

Limitations of QUASILINPDE

One restriction on the applicability of QUASILINPDE results from the program CRACK which tries to solve the characteristic ODE-system of the PDE. So far CRACK can be applied only to polynomially non-linear DE's, i.e. the characteristic ODE-system (16.25),(16.26) or (16.27) may only be polynomially non-linear, i.e. in the PDE (16.22) the expressions a_i and b may only be rational in w_j, ϕ .

The task of CRACK is simplified as (16.28) does not have to be solved for w_j, ϕ . On the other hand (16.28) has to be solved for the c_i . This gives a second restriction coming from the REDUCE function SOLVE. Though SOLVE can be applied to polynomial and transzendental equations, again no guarantee for solvability can be given.

16.2.4 Transformation of DEs

The content of DETRAFO

Finally, after having found the finite transformations, the program APPLYSYM calls the procedure DETRAFO to perform the transformations. DETRAFO can also be used alone to do point- or higher order transformations which involve a considerable computational effort if the differential order of the expression to be transformed is high and if many dependent and independent variables are involved. This might be especially useful if one wants to experiment and try out different coordinate transformations interactively, using DETRAFO as standalone procedure.

To run DETRAFO, the old functions y^α and old variables x^i must be known explicitly in terms of algebraic or differential expressions of the new functions u^β and new variables v^j . Then for point transformations the identity

$$dy^\alpha = \left(y^\alpha_{,v^i} + y^\alpha_{,u^\beta} u^\beta_{,v^i} \right) dv^i \quad (16.31)$$

$$= y^\alpha_{,x^j} dx^j \quad (16.32)$$

$$= y^\alpha_{,x^j} \left(x^j_{,v^i} + x^j_{,u^\beta} u^\beta_{,v^i} \right) dv^i \quad (16.33)$$

provides the transformation

$$y^\alpha_{,x^j} = \frac{dy^\alpha}{dv^i} \cdot \left(\frac{dx^j}{dv^i} \right)^{-1} \quad (16.34)$$

with $\det(dx^j/dv^i) \neq 0$ because of the regularity of the transformation which is checked by DETRAFO. Non-regular transformations are not performed.

DETRAFO is not restricted to point transformations. In the case of contact- or higher order transformations, the total derivatives dy^α/dv^i and dx^j/dv^i then only

include all v^i — derivatives of u^β which occur in

$$\begin{aligned} y^\alpha &= y^\alpha(v^i, u^\beta, u^\beta_{,v^j}, \dots) \\ x^k &= x^k(v^i, u^\beta, u^\beta_{,v^j}, \dots). \end{aligned}$$

Syntax

The call of DETRAFO is

DETRAFO($\{ex_1, ex_2, \dots, ex_m\}$,
 $\{ofun_1=fex_1, ofun_2=fex_2, \dots, ofun_p=fex_p\}$,
 $\{ovar_1=vex_1, ovar_2=vex_2, \dots, ovar_q=vex_q\}$,
 $\{nfun_1, nfun_2, \dots, nfun_p\}$,
 $\{nvar_1, nvar_2, \dots, nvar_q\}$);

where m, p, q are arbitrary.

- The ex_i are differential expressions to be transformed.
- The second list is the list of old functions $ofun$ expressed as expressions fex in terms of new functions $nfun$ and new independent variables $nvar$.
- Similarly the third list expresses the old independent variables $ovar$ as expressions vex in terms of new functions $nfun$ and new independent variables $nvar$.
- The last two lists include the new functions $nfun$ and new independent variables $nvar$.

Names for $ofun$, $ovar$, $nfun$ and $nvar$ can be arbitrarily chosen.

As the result DETRAFO returns the first argument of its input, i.e. the list

$$\{ex_1, ex_2, \dots, ex_m\}$$

where all ex_i are transformed.

Limitations of DETRAFO

The only requirement is that the old independent variables x^i and old functions y^α must be given explicitly in terms of new variables v^j and new functions u^β as indicated in the syntax. Then all calculations involve only differentiations and basic algebra.

Bibliography

- [1] W. Hereman, Chapter 13 in vol 3 of the CRC Handbook of Lie Group Analysis of Differential Equations, Ed.: N.H. Ibragimov, CRC Press, Boca Raton, Florida (1995). Systems described in this paper are among others:
 DELiA (Alexei Bocharov et.al.) Pascal
 DIFFGROB2 (Liz Mansfield) Maple
 DIMSYM (James Sherring and Geoff Prince) REDUCE
 HSYM (Vladimir Gerdt) Reduce
 LIE (V. Eliseev, R.N. Fedorova and V.V. Kornyak) Reduce
 LIE (Alan Head) muMath
 Lie (Gerd Baumann) Mathematica
 LIEDF/INFSYM (Peter Gragert and Paul Kersten) Reduce
 Liesymm (John Carminati, John Devitt and Greg Fee) Maple
 MathSym (Scott Herod) Mathematica
 NUSY (Clara Nucci) Reduce
 PDELIE (Peter Vafeades) Macsyma
 SPDE (Fritz Schwarz) Reduce and Axiom
 SYM.DE (Stanly Steinberg) Macsyma
 Symmgroup.c (Dominique Berube and Marc de Montigny) Mathematica
 STANDARD FORM (Gregory Reid and Alan Wittkopf) Maple
 SYMCAL (Gregory Reid) Macsyma and Maple
 SYMMGRP.MAX (Benoit Champagne, Willy Hereman and Pavel Winter-
 nitz) Macsyma
 LIE package (Khai Vu) Maple
 Toolbox for symmetries (Mark Hickman) Maple
 Lie symmetries (Jeffrey Ondich and Nick Coult) Mathematica.
- [2] S. Lie, Sophus Lie's 1880 Transformation Group Paper, Translated by M. Ackerman, comments by R. Hermann, Mathematical Sciences Press, Brookline, (1975).
- [3] S. Lie, Differentialgleichungen, Chelsea Publishing Company, New York, (1967).
- [4] T. Wolf, An efficiency improved program `LIEPDE` for determining Lie - symmetries of PDEs, Proceedings of the workshop on Modern group theory methods in Acireale (Sicily) Nov. (1992)
- [5] C. Riquier, Les systèmes d'équations aux dérivées partielles, Gauthier-Villars, Paris (1910).
- [6] J. Thomas, Differential Systems, AMS, Colloquium publications, v.21, N.Y. (1937).

- [7] M. Janet, *Leçons sur les systèmes d'équations aux dérivées*, Gauthier–Villars, Paris (1929).
- [8] V.L. Topunov, Reducing Systems of Linear Differential Equations to a Passive Form, *Acta Appl. Math.* 16 (1989) 191–206.
- [9] A.V. Bocharov and M.L. Bronstein, Efficiently Implementing Two Methods of the Geometrical Theory of Differential Equations: An Experience in Algorithm and Software Design, *Acta. Appl. Math.* 16 (1989) 143–166.
- [10] P.J. Olver, *Applications of Lie Groups to Differential Equations*, Springer-Verlag New York (1986).
- [11] G.J. Reid, A triangularization algorithm which determines the Lie symmetry algebra of any system of PDEs, *J.Phys. A: Math. Gen.* 23 (1990) L853-L859.
- [12] F.Schwarz, Automatically Determining Symmetries of Partial Differential Equations, *Computing* 34, (1985) 91-106.
- [13] W.I.Fushchich and V.V.Korniyak, Computer Algebra Application for Determining Lie and Lie–Bäcklund Symmetries of Differential Equations, *J. Symb. Comp.* 7 (1989) 611–619.
- [14] E. Kamke, *Differentialgleichungen, Lösungsmethoden und Lösungen, Band 1, Gewöhnliche Differentialgleichungen*, Chelsea Publishing Company, New York, 1959.
- [15] E. Kamke, *Differentialgleichungen, Lösungsmethoden und Lösungen, Band 2, Partielle Differentialgleichungen, 6.Aufl., Teubner, Stuttgart:Teubner, 1979.*
- [16] T. Wolf, An Analytic Algorithm for Decoupling and Integrating systems of Nonlinear Partial Differential Equations, *J. Comp. Phys.*, no. 3, 60 (1985) 437-446 and, *Zur analytischen Untersuchung und exakten Lösung von Differentialgleichungen mit Computeralgebrasystemen*, Dissertation B, Jena (1989).
- [17] T. Wolf, A. Brand, The Computer Algebra Package CRACK for Investigating PDEs, Manual for the package CRACK in the REDUCE network library and in Proceedings of ERCIM School on Partial Differential Equations and Group Theory, April 1992 in Bonn, GMD Bonn.
- [18] M.A.H. MacCallum, F.J. Wright, *Algebraic Computing with REDUCE*, Clarendon Press, Oxford (1991).
- [19] M.A.H. MacCallum, An Ordinary Differential Equation Solver for REDUCE, *Proc. ISAAC'88, Springer Lect. Notes in Comp Sci.* 358, 196–205.

- [20] H. Stephani, Differential equations, Their solution using symmetries, Cambridge University Press (1989).
- [21] V.I. Karpman, Phys. Lett. A 136, 216 (1989)
- [22] B.Champagne, W.Hereman and P.Winternitz, The computer calculation of Lie point symmetries of large systems of differential equations, Comp. Phys. Comm. 66, 319-340 (1991)
- [23] M. Kubitza, private communication

16.3 ARNUM: An algebraic number package

This package provides facilities for handling algebraic numbers as polynomial coefficients in REDUCE calculations. It includes facilities for introducing indeterminates to represent algebraic numbers, for calculating splitting fields, and for factoring and finding greatest common divisors in such domains.

Author: Eberhard Schrüfer.

Algebraic numbers are the solutions of an irreducible polynomial over some ground domain. The algebraic number i (imaginary unit), for example, would be defined by the polynomial $i^2 + 1$. The arithmetic of algebraic number s can be viewed as a polynomial arithmetic modulo the defining polynomial.

Given a defining polynomial for an algebraic number a

$$a^n + p_{n-1}a^{n-1} + \dots + p_0$$

All algebraic numbers which can be built up from a are then of the form:

$$r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + \dots + r_0$$

where the r_j 's are rational numbers.

The operation of addition is defined by

$$(r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + \dots) + (s_{n-1}a^{n-1} + s_{n-2}a^{n-2} + \dots) = (r_{n-1} + s_{n-1})a^{n-1} + (r_{n-2} + s_{n-2})a^{n-2} + \dots$$

Multiplication of two algebraic numbers can be performed by normal polynomial multiplication followed by a reduction of the result with the help of the defining polynomial.

$$\begin{aligned} (r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + \dots) \times (s_{n-1}a^{n-1} + s_{n-2}a^{n-2} + \dots) = \\ r_{n-1}s^{n-1}a^{2n-2} + \dots \text{ modulo } a^n + p_{n-1}a^{n-1} + \dots + p_0 \\ = q_{n-1}a^{n-1} + q_{n-2}a^{n-2} + \dots \end{aligned}$$

Division of two algebraic numbers r and s yields another algebraic number q .

$$\frac{r}{s} = q \text{ or } r = qs.$$

The last equation written out explicitly reads

$$\begin{aligned} (r_{n-1}a^{n-1} + r_{n-2}a^{n-2} + \dots) \\ = (q_{n-1}a^{n-1} + q_{n-2}a^{n-2} + \dots) \times (s_{n-1}a^{n-1} + s_{n-2}a^{n-2} + \dots) \\ \text{modulo}(a^n + p_{n-1}a^{n-1} + \dots) \\ = (t_{n-1}a^{n-1} + t_{n-2}a^{n-2} + \dots) \end{aligned}$$

The t_i are linear in the q_j . Equating equal powers of a yields a linear system for the quotient coefficients q_j .

With this, all field operations for the algebraic numbers are available. The translation into algorithms is straightforward. For an implementation we have to decide on a data structure for an algebraic number. We have chosen the representation REDUCE normally uses for polynomials, the so-called standard form. Since our polynomials have in general rational coefficients, we must allow for a rational number domain inside the algebraic number.

```

< algebraic number > ::=
  :ar: . < univariate polynomial over the rationals >
< univariate polynomial over the rationals > ::=
  < variable > .** < ldeg > .* < rational > .+ < reductum >
< ldeg > ::= integer

< rational > ::=
  :rn: . < integer numerator > . < integer denominator > : integer
< reductum > ::= < univariate polynomial > : < rational > : nil

```

This representation allows us to use the REDUCE functions for adding and multiplying polynomials on the tail of the tagged algebraic number. Also, the routines for solving linear equations can easily be used for the calculation of quotients. We are still left with the problem of introducing a particular algebraic number. In the current version this is done by giving the defining polynomial to the statement **defpoly**. The algebraic number $\sqrt{2}$, for example, can be introduced by

```
defpoly sqrt2**2 - 2;
```

This statement associates a simplification function for the translation of the variable in the defining polynomial into its tagged internal form and also generates a power reduction rule used by the operations **times** and **quotient** for the reduction of their result modulo the defining polynomial. A basis for the representation of an algebraic number is also set up by the statement. In the working version, the basis is a list of powers of the indeterminate of the defining polynomial up to one less than its degree. Experiments with integral bases, however, have been very encouraging, and these bases might be available in a later version. If the defining polynomial is not monic, it will be made so by an appropriate substitution.

Example 1

```
defpoly sqrt2**2-2;
```

```

1/(sqrt2+1);

sqrt2 - 1

(x**2+2*sqrt2*x+2)/(x+sqrt2);

x + sqrt2

on gcd;

(x**3+(sqrt2-2)*x**2-(2*sqrt2+3)*x-3*sqrt2)/(x**2-2);

      2
(x  - 2*x - 3)/(x - sqrt2)

off gcd;

sqrt(x**2-2*sqrt2*x*y+2*y**2);

abs(x - sqrt2*y)

```

Until now we have dealt with only a single algebraic number. In practice this is not sufficient as very often several algebraic numbers appear in an expression. There are two possibilities for handling this: one can use multivariate extensions [?] or one can construct a defining polynomial that contains all specified extensions. This package implements the latter case (the so called primitive representation). The algorithm we use for the construction of the primitive element is the same as given by Trager [?]. In the implementation, multiple extensions can be given as a list of equations to the statement **defpoly**, which, among other things, adds the new extension to the previously defined one. All algebraic numbers are then expressed in terms of the primitive element.

Example 2

```

defpoly sqrt2**2-2,cbrr5**3-5;

*** defining polynomial for primitive element:

      6      4      3      2
a1  - 6*a1  - 10*a1  + 12*a1  - 60*a1 + 17

sqrt2;

      5      4      3      2

```

```
48/1187*a1 + 45/1187*a1 - 320/1187*a1 - 780/1187*a1 +
```

```
735/1187*a1 - 1820/1187
```

```
sqrt2**2;
```

```
2
```

We can provide factorization of polynomials over the algebraic number domain by using Trager's algorithm. The polynomial to be factored is first mapped to a polynomial over the integers by computing the norm of the polynomial, which is the resultant with respect to the primitive element of the polynomial and the defining polynomial. After factoring over the integers, the factors over the algebraic number field are recovered by GCD calculations.

Example 3

```
defpoly a**2-5;

on factor;

x**2 + x - 1;

(x + (1/2*a + 1/2))*(x - (1/2*a - 1/2))
```

We have also incorporated a function **split_field** for the calculation of a primitive element of minimal degree for which a given polynomial splits into linear factors. The algorithm as described in Trager's article is essentially a repeated primitive element calculation.

Example 4

```
split_field(x**3-3*x+7);

*** Splitting field is generated by:


$$a^6 - 18a^4 + 81a^2 + 1215$$



$$\{ \frac{1}{126}a^4 - \frac{5}{42}a^2 - \frac{1}{2}a^2 + \frac{2}{7},$$


$$- (\frac{1}{63}a^4 - \frac{5}{21}a^2 + \frac{4}{7}),$$


$$\frac{1}{126}a^4 - \frac{5}{42}a^2 + \frac{1}{2}a^2 + \frac{2}{7} \}$$

```

```
for each j in ws product (x-j);
```

$$x^3 - 3x + 7$$

A more complete description can be found in [?].

16.4 ASSERT: Dynamic Verification of Assertions on Function Types

ASSERT admits to add to symbolic mode RLISP code assertions (partly) specifying *types* of the arguments and results of RLISP expr procedures. These types can be associated with functions testing the validity of the respective arguments during runtime.

Author: Thomas Sturm.

16.5 ASSIST: Useful utilities for various applications

ASSIST contains a large number of additional general purpose functions that allow a user to better adapt REDUCE to various calculational strategies and to make the programming task more straightforward and more efficient.

Author: Hubert Caprasse.

16.6 AVECTOR: A vector algebra and calculus package

This package provides REDUCE with the ability to perform vector algebra using the same notation as scalar algebra. The basic algebraic operations are supported, as are differentiation and integration of vectors with respect to scalar variables, cross product and dot product, component manipulation and application of scalar functions (e.g. cosine) to a vector to yield a vector result.

Author: David Harper.

16.6.1 Introduction

This package ¹ is written in RLISP (the LISP meta-language) and is intended for use with REDUCE 3.4. It provides REDUCE with the ability to perform vector algebra using the same notation as scalar algebra. The basic algebraic operations are supported, as are differentiation and integration of vectors with respect to scalar variables, cross product and dot product, component manipulation and application of scalar functions (e.g. cosine) to a vector to yield a vector result.

A set of vector calculus operators are provided for use with any orthogonal curvilinear coordinate system. These operators are gradient, divergence, curl and del-squared (Laplacian). The Laplacian operator can take scalar or vector arguments.

Several important coordinate systems are pre-defined and can be invoked by name. It is also possible to create new coordinate systems by specifying the names of the coordinates and the values of the scale factors.

16.6.2 Vector declaration and initialisation

Any name may be declared to be a vector, provided that it has not previously been declared as a matrix or an array. To declare a list of names to be vectors use the VEC command:

```
VEC A, B, C;
```

¹Reference: Computer Physics Communications, **54**, 295-305 (1989)

declares the variables A, B and C to be vectors. If they have already been assigned (scalar) values, these will be lost.

When a vector is declared using the `VEC` command, it does not have an initial value.

If a vector value is assigned to a scalar variable, then that variable will automatically be declared as a vector and the user will be notified that this has happened.

A vector may be initialised using the `AVEC` function which takes three scalar arguments and returns a vector made up from those scalars. For example

```
A := AVEC (A1, A2, A3);
```

sets the components of the vector A to A1, A2 and A3.

16.6.3 Vector algebra

(In the examples which follow, V, V1, V2 *etc* are assumed to be vectors while S, S1, S2 *etc* are scalars.)

The scalar algebra operators +, -, * and / may be used with vector operands according to the rules of vector algebra. Thus multiplication and division of a vector by a scalar are both allowed, but it is an error to multiply or divide one vector by another.

```
V := V1 + V2 - V3;   Addition and subtraction
V := S1*3*V1;        Scalar multiplication
V := V1/S;           Scalar division
V := -V1;            Negation
```

Vector multiplication is carried out using the infix operators `DOT` and `CROSS`. These are defined to have higher precedence than scalar multiplication and division.

```
V := V1 CROSS V2;    Cross product
S := V1 DOT V2;      Dot product
V := V1 CROSS V2 + V3;
V := (V1 CROSS V2) + V3;
```

The last two expressions are equivalent due to the precedence of the `CROSS` operator.

The modulus of a vector may be calculated using the `VMOD` operator.

```
S := VMOD V;
```

A unit vector may be generated from any vector using the `VMOD` operator.

```
V1 := V / (VMOD V);
```

Components may be extracted from any vector using index notation in the same way as an array.

```
V := AVEC (AX, AY, AZ);
V(0);           yields AX
V(1);           yields AY
V(2);           yields AZ
```

It is also possible to set values of individual components. Following from above:

```
V(1) := B;
```

The vector *V* now has components *AX*, *B*, *AZ*.

Vectors may be used as arguments in the differentiation and integration routines in place of the dependent expression.

```
V := AVEC (X**2, SIN(X), Y);
DF(V, X);           yields (2*X, COS(X), 0)
INT(V, X);           yields (X**3/3, -COS(X), Y*X)
```

Vectors may be given as arguments to monomial functions such as *SIN*, *LOG* and *TAN*. The result is a vector obtained by applying the function component-wise to the argument vector.

```
V := AVEC (A1, A2, A3);
SIN(V);              yields (SIN(A1), SIN(A2), SIN(A3))
```

16.6.4 Vector calculus

The vector calculus operators *div*, *grad* and *curl* are recognised. The Laplacian operator is also available and may be applied to scalar and vector arguments.

```
V := GRAD S;         Gradient of a scalar field
S := DIV V;           Divergence of a vector field
V := CURL V1;         Curl of a vector field
S := DELSQ S1;        Laplacian of a scalar field
V := DELSQ V1;        Laplacian of a vector field
```

These operators may be used in any orthogonal curvilinear coordinate system. The user may alter the names of the coordinates and the values of the scale factors. Initially the coordinates are *X*, *Y* and *Z* and the scale factors are all unity.

There are two special vectors : *COORDS* contains the names of the coordinates in the current system and *HFACTORS* contains the values of the scale factors.

The coordinate names may be changed using the *COORDINATES* operator.

```
COORDINATES R, THETA, PHI;
```

This command changes the coordinate names to R, THETA and PHI.

The scale factors may be altered using the SCALEFACTORS operator.

```
SCALEFACTORS (1, R, R*SIN (THETA) ) ;
```

This command changes the scale factors to 1, R and $R \sin(\text{THETA})$.

Note that the arguments of SCALEFACTORS must be enclosed in parentheses. This is not necessary with COORDINATES.

When vector differential operators are applied to an expression, the current set of coordinates are used as the independent variables and the scale factors are employed in the calculation. (See, for example, Batchelor G.K. 'An Introduction to Fluid Mechanics', Appendix 2.)

Several coordinate systems are pre-defined and may be invoked by name. To see a list of valid names enter

```
SYMBOLIC !*CSYSTEMS;
```

and REDUCE will respond with something like

```
(CARTESIAN SPHERICAL CYLINDRICAL)
```

To choose a coordinate system by name, use the command GETCSYSTEM.

To choose the Cartesian coordinate system :

```
GETCSYSTEM 'CARTESIAN;
```

Note the quote which prefixes the name of the coordinate system. This is required because GETCSYSTEM (and its complement PUTCSYSTEM) is a SYMBOLIC procedure which requires a literal argument.

REDUCE responds by typing a list of the coordinate names in that coordinate system. The example above would produce the response

```
(X Y Z)
```

whilst

```
GETCSYSTEM 'SPHERICAL;
```

would produce

```
(R THETA PHI)
```

Note that any attempt to invoke a coordinate system is subject to the same restrictions as the implied calls to `COORDINATES` and `SCALEFACTORS`. In particular, `GETCSYSTEM` fails if any of the coordinate names has been assigned a value and the previous coordinate system remains in effect.

A user-defined coordinate system can be assigned a name using the command `PUTCSYSTEM`. It may then be re-invoked at a later stage using `GETCSYSTEM`.

Example 5

We define a general coordinate system with coordinate names X, Y, Z and scale factors H_1, H_2, H_3 :

```
COORDINATES X, Y, Z;
SCALEFACTORS (H1, H2, H3);
PUTCSYSTEM 'GENERAL;
```

This system may later be invoked by entering

```
GETCSYSTEM 'GENERAL;
```

16.6.5 Volume and Line Integration

Several functions are provided to perform volume and line integrals. These operate in any orthogonal curvilinear coordinate system and make use of the scale factors described in the previous section.

Definite integrals of scalar and vector expressions may be calculated using the `DEFINT` function.

Example 6

To calculate the definite integral of $\sin(x)^2$ between 0 and 2π we enter

```
DEFINT (SIN (X) **2, X, 0, 2*PI);
```

This function is a simple extension of the `INT` function taking two extra arguments, the lower and upper bounds of integration respectively.

Definite volume integrals may be calculated using the `VOLINTEGRAL` function whose syntax is as follows :

VOLINTEGRAL(integrand, vector lower-bound, vector upper-bound);

Example 7

In spherical polar coordinates we may calculate the volume of a sphere by integrating unity over the range $r=0$ to RR , $\theta=0$ to PI , $\phi=0$ to $2*\pi$ as follows :

```
VLB := AVEC (0, 0, 0);      Lower bound
VUB := AVEC (RR, PI, 2*PI); Upper bound in r,  $\theta$ ,  $\phi$  respectively
VOLINTORDER := (0, 1, 2);  The order of integration
VOLINTEGRAL (1, VLB, VUB);
```

Note the use of the special vector VOLINTORDER which controls the order in which the integrations are carried out. This vector should be set to contain the number 0, 1 and 2 in the required order. The first component of VOLINTORDER contains the index of the first integration variable, the second component is the index of the second integration variable and the third component is the index of the third integration variable.

Example 8

Suppose we wish to calculate the volume of a right circular cone. This is equivalent to integrating unity over a conical region with the bounds:

```
z = 0 to H      (H = the height of the cone)
r = 0 to pZ     (p = ratio of base diameter to height)
phi = 0 to 2*PI
```

We evaluate the volume by integrating a series of infinitesimally thin circular disks of constant z-value. The integration is thus performed in the order : $d(\phi)$ from 0 to 2π , dr from 0 to $p*Z$, dz from 0 to H . The order of the indices is thus 2, 0, 1.

```
VOLINTORDER := AVEC (2, 0, 1);
VLB := AVEC (0, 0, 0);
VUB := AVEC (P*Z, H, 2*PI);
VOLINTEGRAL (1, VLB, VUB);
```

(At this stage, we replace $P*H$ by RR , the base radius of the cone, to obtain the result in its more familiar form.)

Line integrals may be calculated using the LINEINT and DEFLINEINT functions. Their general syntax is

```
LINEINT(vector-function, vector-curve, variable);
DEFLINENINT(vector-function, vector-curve, variable, lower-bound,
upper-bound);
```

where

vector-function is any vector-valued expression;

`vector-curve` is a vector expression which describes the path of integration in terms of the independent variable;

`variable` is the independent variable;

`lower-bound`

`upper-bound` are the bounds of integration in terms of the independent variable.

Example 9

In spherical polar coordinates, we may integrate round a line of constant theta ('latitude') to find the length of such a line. The vector function is thus the tangent to the 'line of latitude', (0,0,1) and the path is (0, LAT, PHI) where PHI is the independent variable. We show how to obtain the definite integral *i.e.* from $\phi = 0$ to 2π :

```
DEFLINEINT (AVEC (0, 0, 1) , AVEC (0, LAT, PHI) , PHI, 0, 2*PI) ;
```

16.6.6 Defining new functions and procedures

Most of the procedures in this package are defined in symbolic mode and are invoked by the REDUCE expression-evaluator when a vector expression is encountered. It is not generally possible to define procedures which accept or return vector values in algebraic mode. This is a consequence of the way in which the REDUCE interpreter operates and it affects other non-scalar data types as well : arrays cannot be passed as algebraic procedure arguments, for example.

16.6.7 Acknowledgements

This package was written whilst the author was the U.K. Computer Algebra Support Officer at the University of Liverpool Computer Laboratory.

16.7 BIBASIS: A Package for Calculating Boolean Involutive Bases

Authors: Yuri A. Blinkov and Mikhail V. Zinin

16.7.1 Introduction

Involutive polynomial bases are redundant Gröbner bases of special structure with some additional useful features in comparison with reduced Gröbner bases [1]. Apart from numerous applications of involutive bases [2] the involutive algorithms [3] provide an efficient method for computing reduced Gröbner bases. A reduced Gröbner basis is a well-determined subset of an involutive basis and can be easily extracted from the latter without any extra reductions. All this takes place not only in rings of commutative polynomials but also in Boolean rings.

Boolean Gröbner basis already have already revealed their value and usability in practice. The first impressive demonstration of practicability of Boolean Gröbner bases was breaking the first HFE (Hidden Fields Equations) challenge in the public key cryptography done in [4] by computing a Boolean Gröbner basis for the system of quadratic polynomials in 80 variables. Since that time the Boolean Gröbner bases application area has widen drastically and nowadays there is also a number of quite successful examples of using Gröbner bases for solving SAT problems.

During our research we had developed [5, 6, 7] Boolean involutive algorithms based on Janet and Pommaret divisions and applied them to computation of Boolean Gröbner bases. Our implementation of both divisions has experimentally demonstrated computational superiority of the Pommaret division implementation. This package BIBASIS is the result of our thorough research in the field of Boolean Gröbner bases. BIBASIS implements the involutive algorithm based on Pommaret division in a multivariate Boolean ring.

In section 2 the Boolean ring and its peculiarities are shortly introduced. In section 3 we briefly argue why the involutive algorithm and Pommaret division are good for Boolean ring while the Buhberger's algorithm is not. And finally in section 4 we give the full description of BIBASIS package capabilities and illustrate it by examples.

16.7.2 Boolean Ring

Boolean ring perfectly goes with its name, it is a ring of *Boolean functions* of n variables, i.e mappings from $\{0, 1\}^n$ to $\{0, 1\}^n$. Considering these variables are $\mathbf{X} := \{x_1, \dots, x_n\}$ and \mathbb{F}_2 is the finite field of two elements $\{0, 1\}$, Boolean ring

can be regarded as the quotient ring

$$\mathbb{B}[\mathbf{X}] := \mathbb{F}_2[\mathbf{X}] / \langle x_1^2 + x_1, \dots, x_n^2 + x_n \rangle.$$

Multiplication in $\mathbb{B}[\mathbf{X}]$ is *idempotent* and addition is *nilpotent*

$$\forall b \in \mathbb{B}[\mathbf{X}] : b^2 = b, b + b = 0.$$

Elements in $\mathbb{B}[\mathbf{X}]$ are *Boolean polynomials* and can be represented as finite sums

$$\sum_j \prod_{x \in \Omega_j \subseteq \mathbf{X}} x$$

of *Boolean monomials*. Each monomial is a conjunction. If set Ω is empty, then the corresponding monomial is the unity Boolean function 1. The sum of zero monomials corresponds to zero polynomial, i.e. is zero Boolean function 0.

16.7.3 Pommaret Involutive Algorithm

Detailed description of involutive algorithm can found in [3]. Here we note that result of both involutive and Buhberger's algorithms depend on chosen monomial ordering. At that the ordering must be admissible, i.e.

$$m \neq 1 \iff m \succ 1, \quad m_1 \succ m_2 \iff m_1 m \succ m_2 m \quad \forall m, m_1, m_2.$$

But as one can easily check the second condition of admissibility does not hold for any monomial ordering in Boolean ring:

$$x_1 \succ x_2 \xrightarrow{*x_1} x_1 * x_1 \succ x_2 * x_2 \longrightarrow x_1 \prec x_1 x_2$$

Though $\mathbb{B}[\mathbf{X}]$ is a principal ideal ring, boolean singleton $\{p\}$ is not necessarily a Gröbner basis of ideal $\langle p \rangle$, for example:

$$x_1, x_2 \in \langle x_1 x_2 + x_1 + x_2 \rangle \subset \mathbb{B}[x_1, x_2].$$

That the reason why one cannot apply the Buhberger's algorithm directly in a Boolean ring, using instead a ring $\mathbb{F}_2[\mathbf{X}]$ and the *field binomials* $x_1^2 + x_1, \dots, x_n^2 + x_n$.

The involutive algorithm based on Janet division has the same disadvantage unlike the Pommaret division algorithm as shown in [5]. The Pommaret division algorithm can be applied directly in a Boolean ring and admits effective data structures for monomial representation.

16.7.4 BIBASIS Package

The package BIBASIS implements the Pommaret division algorithm in a Boolean ring. The first step to using the package is to load it:

```
1: load_package bibasis;
```

The current version of the BIBASIS user interface consists only of 2 functions: `bibasis` and `bibasis.print_statistics`.

The `bibasis` is the function that performs all the computation and has the following syntax:

```
bibasis(initial_polynomial_list, variables_list,
        monomial_ordering, reduce_to_groebner);
```

Input:

- `initial_polynomial_list` is the list of polynomials containing the known basis of initial Boolean ideal. All given polynomials are treated modulo 2. See Example 1.
- `variables_list` is the list of independent variables in decreasing order.
- `monomial_ordering` is a chosen monomial ordering and the supported ones are:

```
lex – pure lexicographical ordering;
deglex – degree lexicographic ordering;
degrevlex – degree reverse lexicographic.
```

See Examples 2—4 to check that Gröbner (as well as involutive) basis depends on monomial ordering.

- `reduce_to_groebner` is a Boolean value, if it is `t` the output is the reduced Boolean Gröbner basis, if `nil`, then the reduced Boolean Pommaret basis. Examples 5,6 show distinctions between these two outputs.

Output:

- The list of polynomials which constitute the reduced Boolean Gröbner or Pommaret basis.

The syntax of `bibasis_print_statistics` is simple:

```
bibasis_print_statistics();
```

This function prints out a brief statistics for the last invocation of `bibasis` function. See Example 7.

16.7.5 Examples

Example 1:

```
1: load_package bibasis;
2: bibasis({x+2*y}, {x,y}, lex, t);
{x}
```

Example 2:

```
1: load_package bibasis;
2: variables := {x0,x1,x2,x3,x4}$
3: polynomials := {x0*x3+x1*x2,x2*x4+x0}$
4: bibasis(polynomials, variables, lex, t);
{x0 + x2*x4, x2*(x1 + x3*x4)}
```

Example 3:

```
1: load_package bibasis;
2: variables := {x0,x1,x2,x3,x4}$
3: polynomials := {x0*x3+x1*x2,x2*x4+x0}$
4: bibasis(polynomials, variables, deglex, t);
{x1*x2*(x3 + 1),
 x1*(x0 + x2),
 x0*(x2 + 1),
 x0*x3 + x1*x2,
 x0*(x4 + 1),
 x2*x4 + x0}
```

Example 4:

```
1: load_package bibasis;
2: variables := {x0,x1,x2,x3,x4}$
```

```

3: polynomials := {x0*x3+x1*x2,x2*x4+x0}$
4: bibasis(polynomials, variables, degrevlex, t);
{x0*(x1 + x3),
 x0*(x2 + 1),
 x1*x2 + x0*x3,
 x0*(x4 + 1),
 x2*x4 + x0}

```

Example 5:

```

1: load_package bibasis;
2: variables := {x, y, z}$
3: polinomials := {x, z}$
4: bibasis(polinomials, variables, degrevlex, t);
{x, z}

```

Example 6:

```

1: load_package bibasis;
2: variables := {x, y, z}$
3: polinomials := {x, z}$
4: bibasis(polinomials, variables, degrevlex, nil);
{x, z, y*z}

```

Example 7:

```

1: load_package bibasis;
2: variables := {u0, u1, u2, u3, u4, u5, u6, u7, u8, u9}$
3: polinomials := {u0*u1+u1*u2+u1+u2*u3+u3*u4+u4*u5+u5*u6+u6*u7+u7*u8+
3: u0*u2+u1+u1*u3+u2*u4+u2+u3*u5+u4*u6+u5*u7+u6*u8+u7*
3: u0*u3+u1*u2+u1*u4+u2*u5+u3*u6+u3+u4*u7+u5*u8+u6*u9,
3: u0*u4+u1*u3+u1*u5+u2+u2*u6+u3*u7+u4*u8+u4+u5*u9,
3: u0*u5+u1*u4+u1*u6+u2*u3+u2*u7+u3*u8+u4*u9+u5,
3: u0*u6+u1*u5+u1*u7+u2*u4+u2*u8+u3+u3*u9+u6,
3: u0*u7+u1*u6+u1*u8+u2*u5+u2*u9+u3*u4+u7,
3: u0*u8+u1*u7+u1*u9+u2*u6+u3*u5+u4+u8,
3: u0+u1+u2+u3+u4+u5+u6+u7+u8+u9+1}$
4: bibasis(polinomials, variables, degrevlex, t);
{u3*u6,
u3*u7,
u7*(u6 + 1),
u3*u8,
u6*u8 + u6 + u7,
u7*u8,
u3*(u9 + 1),
u6*u9 + u7,
u7*(u9 + 1),
u8*u9 + u6 + u7 + u8,
u0 + u3 + u6 + u9 + 1,
u1 + u7,

```

```

u2 + u7 + u8,
u4 + u6 + u8,
u5 + u6 + u7 + u8}
5: bibasis_print_statistics();
    Variables order = u0 > u1 > u2 > u3 > u4 > u5 > u6 > u7 > u8 > u9
Normal forms calculated = 216
    Non-zero normal forms = 85
    Reductions made = 4488
Time: 270 ms
GC time: 0 ms

```

Bibliography

- [1] V.P.Gerdt and Yu.A.Blinkov. *Involutive Bases of Polynomial Ideals*. Mathematics and Computers in Simulation, 45, 519–542, 1998; *Minimal Involutive Bases*, ibid. 543–560.
- [2] W.M.Seiler. *Involution: The Formal Theory of Differential Equations and its Applications in Computer Algebra*. Algorithms and Computation in Mathematics, 24, Springer, 2010. arXiv:math.AC/0501111
- [3] Vladimir P. Gerdt. *Involutive Algorithms for Computing Gröbner Bases*. Computational Commutative and Non-Commutative Algebraic Geometry. IOS Press, Amsterdam, 2005, pp.199–225.
- [4] J.-C.Faugère and A.Joux. Algebraic Cryptanalysis of Hidden Field Equations (HFE) Using Gröbner Bases. *LNCS 2729*, Springer-Verlag, 2003, pp.44–60.
- [5] V.P.Gerdt and M.V.Zinin. A Pommaret Division Algorithm for Computing Gröbner Bases in Boolean Rings. *Proceedings of ISSAC 2008*, ACM Press, 2008, pp.95–102.
- [6] V.P.Gerdt and M.V.Zinin. Involutive Method for Computing Gröbner Bases over F_2 . *Programming and Computer Software*, Vol.34, No. 4, 2008, 191–203.
- [7] Vladimir Gerdt, Mikhail Zinin and Yuri Blinkov. On computation of Boolean involutive bases, Proceedings of International Conference Polynomial Computer Algebra 2009, pp. 17-24 (International Euler Institute, April 7-12, 2009, St. Peterburg, Russia)

16.8 BOOLEAN: A package for boolean algebra

This package supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions connected by the infix boolean operators **and**, **or**, **implies**, **equiv**, and the unary prefix operator **not**. **Boolean** allows you to simplify expressions built from these operators, and to test properties like equivalence, subset property etc.

Author: Herbert Melenk.

16.8.1 Introduction

The package **Boolean** supports the computation with boolean expressions in the propositional calculus. The data objects are composed from algebraic expressions (“atomic parts”, “leafs”) connected by the infix boolean operators **and**, **or**, **implies**, **equiv**, and the unary prefix operator **not**. **Boolean** allows you to simplify expressions built from these operators, and to test properties like equivalence, subset property etc. Also the reduction of a boolean expression by a partial evaluation and combination of its atomic parts is supported.

16.8.2 Entering boolean expressions

In order to distinguish boolean data expressions from boolean expressions in the REDUCE programming language (e.g. in an **if** statement), each expression must be tagged explicitly by an operator **boolean**. Otherwise the boolean operators are not accepted in the REDUCE algebraic mode input. The first argument of **boolean** can be any boolean expression, which may contain references to other boolean values.

```
boolean (a and b or c);
q := boolean(a and b implies c);
boolean(q or not c);
```

Brackets are used to override the operator precedence as usual. The leafs or atoms of a boolean expression are those parts which do not contain a leading boolean operator. These are considered as constants during the boolean evaluation. There are two pre-defined values:

- **true**, **t** or **1**
- **false**, **nil** or **0**

These represent the boolean constants. In a result form they are used only as **1** and **0**.

By default, a **boolean** expression is converted to a disjunctive normal form, that is a form where terms are connected by **or** on the top level and each term is set of leaf expressions, eventually preceded by **not** and connected by **and**. An operators **or** or **and** is omitted if it would have only one single operand. The result of the transformation is again an expression with leading operator **boolean** such that the boolean expressions remain separated from other algebraic data. Only the boolean constants **0** and **1** are returned untagged.

On output, the operators **and** and **or** are represented as `/\` and `\/`, respectively.

```
boolean(true and false);      ->    0
boolean(a or not(b and c)); -> boolean(not(b) \/ not(c) \/ a)
boolean(a equiv not c);       -> boolean(not(a) /\ c \/ a /\ not(c))
```

16.8.3 Normal forms

The **disjunctive** normal form is used by default. It represents the “natural” view and allows us to represent any form free or parentheses. Alternatively a **conjunctive** normal form can be selected as simplification target, which is a form with leading operator **and**. To produce that form add the keyword **and** as an additional argument to a call of **boolean**.

```
boolean (a or b implies c);
      ->
      boolean(not(a) /\ not(b) \/ c)

boolean (a or b implies c, and);
      ->
      boolean((not(a) \/ c) /\ (not(b) \/ c))
```

Usually the result is a fully reduced disjunctive or conjunctive normal form, where all redundant elements have been eliminated following the rules

$$a \wedge b \vee \neg a \wedge b \longleftrightarrow b$$

$$a \vee b \wedge \neg a \vee b \longleftrightarrow b$$

Internally the full normal forms are computed as intermediate result; in these forms each term contains all leaf expressions, each one exactly once. This unreduced form is returned when you set the additional keyword **full**:

```
boolean (a or b implies c, full);
      ->
boolean(a /\ b /\ c \/ a /\ not(b) /\ c \/ not(a) /\ b /\ c \/ not(a) /\ not(b) /\ c
```

```
\ / not (a) /\not (b) /\not (c) )
```

The keywords **full** and **and** may be combined.

16.8.4 Evaluation of a boolean expression

If the leafs of the boolean expression are algebraic expressions which may evaluate to logical values because the environment has changed (e.g. variables have been bound), you can re-investigate the expression using the operator `testbool` with the boolean expression as argument. This operator tries to evaluate all leaf expressions in REDUCEboolean style. As many terms as possible are replaced by their boolean values; the others remain unchanged. The resulting expression is contracted to a minimal form. The result **1** (= true) or **0** (=false) signals that the complete expression could be evaluated.

In the following example the leafs are built as numeric greater test. For using `>` in the expressions the greater sign must be declared operator first. The error messages are meaningless.

```
operator >;
fm:=boolean(x>v or not (u>v));
      ->
      fm := boolean(not (u>v) \ / x>v)

v:=10$

testbool fm;

      ***** u - 10 invalid as number
      ***** x - 10 invalid as number

      ->
      boolean(not (u>10) \ / x>10)

x:=3$
testbool fm;

      ***** u - 10 invalid as number

      ->
      boolean(not (u>10) )

x:=17$
```

```
testbool fm;
```

```
***** u - 10 invalid as number
```

```
    ->
```

```
1
```

16.9 CALI: A package for computational commutative algebra

This package contains algorithms for computations in commutative algebra closely related to the Gröbner algorithm for ideals and modules. Its heart is a new implementation of the Gröbner algorithm that also allows for the computation of syzygies. This implementation is also applicable to submodules of free modules with generators represented as rows of a matrix.

Author: Hans-Gert Gräbe.

16.10 CAMAL: Calculations in celestial mechanics

This package implements in REDUCE the Fourier transform procedures of the CAMAL package for celestial mechanics.

Author: John P. Fitch.

It is generally accepted that special purpose algebraic systems are more efficient than general purpose ones, but as machines get faster this does not matter. An experiment has been performed to see if using the ideas of the special purpose algebra system CAMAL(F) it is possible to make the general purpose system REDUCE perform calculations in celestial mechanics as efficiently as CAMAL did twenty years ago. To this end a prototype Fourier module is created for REDUCE, and it is tested on some small and medium-sized problems taken from the CAMAL test suite. The largest calculation is the determination of the Lunar Disturbing Function to the sixth order. An assessment is made as to the progress, or lack of it, which computer algebra has made, and how efficiently we are using modern hardware.

16.10.1 Introduction

A number of years ago there emerged the divide between general-purpose algebra systems and special purpose one. Here we investigate how far the improvements in software and more predominantly hardware have enabled the general systems to perform as well as the earlier special ones. It is similar in some respects to the Possion program for MACSYMA [?] which was written in response to a similar challenge.

The particular subject for investigation is the Fourier series manipulator which had its origins in the Cambridge University Institute for Theoretical Astronomy, and later became the F subsystem of CAMAL [?, ?]. In the late 1960s this system was used for both the Delaunay Lunar Theory [?, ?] and the Hill Lunar Theory [?], as well as other related calculations. Its particular area of application had a number of peculiar operations on which the general speed depended. These are outlined below in the section describing how CAMAL worked. There have been a number of subsequent special systems for celestial mechanics, but these tend to be restricted to the group of the originator.

The main body of the paper describes an experiment to create within the REDUCE system a sub-system for the efficient manipulation of Fourier series. This prototype program is then assessed against both the normal (general) REDUCE and the extant CAMAL results. The tests are run on a number of small problems typical of those for which CAMAL was used, and one medium-sized problem, the calculation of the Lunar Disturbing Function. The mathematical background to this problem is also presented for completeness. It is important as a problem as it is the first stage

in the development of a Delaunay Lunar Theory.

The paper ends with an assessment of how close the performance of a modern REDUCE on modern equipment is to the (almost) defunct CAMAL of eighteen years ago.

16.10.2 How CAMAL Worked

The Cambridge Algebra System was initially written in assembler for the Titan computer, but later was rewritten a number of times, and matured in BCPL, a version which was ported to IBM mainframes and a number of microcomputers. In this section a brief review of the main data structures and special algorithms is presented.

CAMAL Data Structures

CAMAL is a hierarchical system, with the representation of polynomials being completely independent of the representations of the angular parts.

The angular part had to represent a polynomial coefficient, either a sine or cosine function and a linear sum of angles. In the problems for which CAMAL was designed there are 6 angles only, and so the design restricted the number, initially to six on the 24 bit-halfword TITAN, and later to eight angles on the 32-bit IBM 370, each with fixed names (usually *u* through *z*). All that is needed is to remember the coefficients of the linear sum. As typical problems are perturbations, it was reasonable to restrict the coefficients to small integers, as could be represented in a byte with a guard bit. This allowed the representation to pack everything into four words.

```
[ NextTerm, Coefficient, Angles0-3, Angles4-7 ]
```

The function was coded by a single bit in the `Coefficient` field. This gives a particularly compact representation. For example the Fourier term $\sin(u - 2v + w - 3x)$ would be represented as

```
[ NULL, "1"|0x1, 0x017e017d, 0x00000000 ]
or
[ NULL, "1"|0x1, 1:-2:1:-3, 0:0:0:0 ]
```

where "1" is a pointer to the representation of the polynomial 1. In all this representation of the term took 48 bytes. As the complexity of a term increased the store requirements to no grow much; the expression $(7/4)ae^3f^5 \cos(u - 2v + 3w - 4x + 5y + 6z)$ also takes 48 bytes. There is a canonicalisation operation to ensure that the leading angle is positive, and $\sin(0)$ gets removed. It should be noted that

$\cos(0)$ is a valid and necessary representation.

The polynomial part was similarly represented, as a chain of terms with packed exponents for a fixed number of variables. There is no particular significance in this except that the terms were held in *increasing* total order, rather than the decreasing order which is normal in general purpose systems. This had a number of important effects on the efficiency of polynomial multiplication in the presence of a truncation to a certain order. We will return to this point later. Full details of the representation can be found in [?].

The space administration system was based on explicit return rather than garbage collection. This meant that the system was sometimes harder to write, but it did mean that much attention was focussed on efficient reuse of space. It was possible for the user to assist in this by marking when an expression was needed no longer, and the compiler then arranged to recycle the space as part of the actual operation. This degree of control was another assistance in running of large problems on relatively small machines.

Automatic Linearisation

In order to maintain Fourier series in a canonical form it is necessary to apply the transformations for linearising products of sine and cosines. These will be familiar to readers of the REDUCE test program as

$$\cos \theta \cos \phi \Rightarrow (\cos(\theta + \phi) + \cos(\theta - \phi))/2, \quad (16.35)$$

$$\cos \theta \sin \phi \Rightarrow (\sin(\theta + \phi) - \sin(\theta - \phi))/2, \quad (16.36)$$

$$\sin \theta \sin \phi \Rightarrow (\cos(\theta - \phi) - \cos(\theta + \phi))/2, \quad (16.37)$$

$$\cos^2 \theta \Rightarrow (1 + \cos(2\theta))/2, \quad (16.38)$$

$$\sin^2 \theta \Rightarrow (1 - \cos(2\theta))/2. \quad (16.39)$$

In CAMAL these transformations are coded directly into the multiplication routines, and no action is necessary on the part of the user to invoke them. Of course they cannot be turned off either.

Differentiation and Integration

The differentiation of a Fourier series with respect to an angle is particularly simple. The integration of a Fourier series is a little more interesting. The terms like $\cos(nu + \dots)$ are easily integrated with respect to u , but the treatment of terms independent of the angle would normally introduce a secular term. By convention in Fourier series these secular terms are ignored, and the constant of integration is taken as just the terms independent of the angle in the integrand. This is equivalent

to the substitution rules

$$\begin{aligned}\sin(n\theta) &\Rightarrow -(1/n) \cos(n\theta) \\ \cos(n\theta) &\Rightarrow (1/n) \sin(n\theta)\end{aligned}$$

In CAMAL these operations were coded directly, and independently of the differentiation and integration of the polynomial coefficients.

Harmonic Substitution

An operation which is of great importance in Fourier operations is the *harmonic substitution*. This is the substitution of the sum of some angles and a general expression for an angle. In order to preserve the format, the mechanism uses the translations

$$\begin{aligned}\sin(\theta + A) &\Rightarrow \sin(\theta) \cos(A) + \cos(\theta) \sin(A) \\ \cos(\theta + A) &\Rightarrow \cos(\theta) \cos(A) - \sin(\theta) \sin(A)\end{aligned}$$

and then assuming that the value A is small it can be replaced by its expansion:

$$\begin{aligned}\sin(\theta + A) &\Rightarrow \sin(\theta) \{1 - A^2/2! + A^4/4! \dots\} + \\ &\quad \cos(\theta) \{A - A^3/3! + A^5/5! \dots\} \\ \cos(\theta + A) &\Rightarrow \cos(\theta) \{1 - A^2/2! + A^4/4! \dots\} - \\ &\quad \sin(\theta) \{A - A^3/3! + A^5/5! \dots\}\end{aligned}$$

If a truncation is set for large powers of the polynomial variables then the series will terminate. In CAMAL the `HSUB` operation took five arguments; the original expression, the angle for which there is a substitution, the new angular part, the expression part (A in the above), and the number of terms required.

The actual coding of the operation was not as expressed above, but by the use of Taylor's theorem. As has been noted above the differentiation of a harmonic series is particularly easy.

Truncation of Series

The main use of Fourier series systems is in generating perturbation expansions, and this implies that the calculations are performed to some degree of the small quantities. In the original CAMAL all variables were assumed to be equally small (a restriction removed in later versions). By maintaining polynomials in increasing

maximum order it is possible to truncate the multiplication of two polynomials. Assume that we are multiplying the two polynomials

$$\begin{aligned} A &= a_0 + a_1 + a_2 + \dots \\ B &= b_0 + b_1 + b_2 + \dots \end{aligned}$$

If we are generating the partial answer

$$a_i(b_0 + b_1 + b_2 + \dots)$$

then if for some j the product $a_i b_j$ vanishes, then so will all products $a_i b_k$ for $k > j$. This means that the later terms need not be generated. In the product of $1 + x + x^2 + x^3 + \dots + x^{10}$ and $1 + y + y^2 + y^3 + \dots + y^{10}$ to a total order of 10 instead of generating 100 term products only 55 are needed. The ordering can also make the merging of the new terms into the answer easier.

16.10.3 Towards a CAMAL Module

For the purposes of this work it was necessary to reproduce as many of the ideas of CAMAL as feasible within the REDUCE framework and philosophy. It was not intended at this stage to produce a complete product, and so for simplicity a number of compromises were made with the “no restrictions” principle in REDUCE and the space and time efficiency of CAMAL. This section describes the basic design decisions.

Data Structures

In a fashion similar to CAMAL a two level data representation is used. The coefficients are the standard quotients of REDUCE, and their representation need not concern us further. The angular part is similar to that of CAMAL, but the ability to pack angle multipliers and use a single bit for the function are not readily available in Standard LISP, so instead a longer vector is used. Two versions were written. One used a balanced tree rather than a linear list for the Fourier terms, this being a feature of CAMAL which was considered but never coded. The other uses a simple linear representation for sums. The angle multipliers are held in a separate vector in order to allow for future flexibility. This leads to a representation as a vector of length 6 or 4;

```
Version1: [ BalanceBits, Coeff, Function, Angles, LeftTree, RightTree ]
Version2: [ Coeff, Function, Angles, Next ]
```

where the `Angles` field is a vector of length 8, for the multipliers. It was decided to forego packing as for portability we do not know how many to pack into a small

integer. The tree system used is AVL, which needs 2 bits to maintain balance information, but these are coded as a complete integer field in the vector. We can expect the improvements implicit in a binary tree to be advantageous for large expressions, but the additional overhead may reduce its utility for smaller expressions.

A separate vector is kept relating the position of an angle to its print name, and on the property list of each angle the allocation of its position is kept. So long as the user declares which variables are to be treated as angles this mechanism gives flexibility which was lacking in CAMAL.

Linearisation

As in the CAMAL system the linearisation of products of sines and cosines is done not by pattern matching but by direct calculation at the heart of the product function, where the transformations (1) through (3) are made in the product of terms function. A side effect of this is that there are no simple relations which can be used from within the Fourier multiplication, and so a full addition of partial products is required. There is no need to apply linearisations elsewhere as a special case. Addition, differentiation and integration cannot generate such products, and where they can occur in substitution the natural algorithm uses the internal multiplication function anyway.

Substitution

Substitution is the main operation of Fourier series. It is useful to consider three different cases of substitutions.

1. Angle Expression for Angle:
2. Angle Expression + Fourier Expression for Angle:
3. Fourier Expression for Polynomial Variable.

The first of these is straightforward, and does not require any further comment. The second substitution requires a little more care, but is not significantly difficult to implement. The method follows the algorithm used in CAMAL, using TAYLOR series. Indeed this is the main special case for substitution.

The problem is the last case. Typically many variables used in a Fourier series program have had a WEIGHT assigned to them. This means that substitution must take account of any possible WEIGHTs for variables. The standard code in REDUCE does this in effect by translating the expression to prefix form, and recalculating the value. A Fourier series has a large number of coefficients, and so this operations are repeated rather too often. At present this is the largest problem area

with the internal code, as will be seen in the discussion of the Disturbing Function calculation.

16.10.4 Integration with REDUCE

The Fourier module needs to be seen as part of REDUCE rather than as a separate language. This can be seen as having internal and external parts.

Internal Interface

The Fourier expressions need to co-exist with the normal REDUCE syntax and semantics. The prototype version does this by (ab)using the module method, based in part on the TPS code [?]. Of course Fourier series are not constant, and so are not really domain elements. However by asserting that Fourier series form a ring of constants REDUCE can arrange to direct basic operations to the Fourier code for addition, subtraction, multiplication and the like.

The main interface which needs to be provided is a simplification function for Fourier expressions. This needs to provide compilation for linear sums of angles, as well as constructing sine and cosine functions, and creating canonical forms.

User Interface

The creation of `HDIFF` and `HINT` functions for differentiation disguises this. An unsatisfactory aspect of the interface is that the tokens `SIN` and `COS` are already in use. The prototype uses the operator form

```
fourier sin(u)
```

to introduce harmonically represented sine functions. An alternative of using the tokens `F_SIN` and `F_COS` is also available.

It is necessary to declare the names of the angles, which is achieved with the declaration

```
harmonic theta, phi;
```

At present there is no protection against using a variable as both an angle and a polynomial variable. This will need to be done in a user-oriented version.

16.10.5 The Simple Experiments

The REDUCE test file contains a simple example of a Fourier calculation, determining the value of $(a_1 \cos(wt) + a_3 \cos(3wt) + b_1 \sin(wt) + b_3 \sin(3wt))^3$. For the purposes of this system this is too trivial to do more than confirm the correct answers.

The simplest non-trivial calculation for a Fourier series manipulator is to solve Kepler's equation for the eccentric anomaly E in terms of the mean anomaly u , and the eccentricity of an orbit e , considered as a small quantity

$$E = u + e \sin E$$

The solution proceeds by repeated approximation. Clearly the initial approximation is $E_0 = u$. The n^{th} approximation can be written as $u + A_n$, and so A_n can be calculated by

$$A_k = e \sin(u + A_{k-1})$$

This is of course precisely the case for which the HSUB operation is designed, and so in order to calculate $E_n - u$ all one requires is the code

```

bige := fourier 0;
for k:=1:n do <<
    wtlevel k;
    bige:=fourier e * hsub(fourier(sin u), u, u, bige, k);
>>;
write "Kepler Eqn solution:", bige$

```

It is possible to create a regular REDUCE program to simulate this (as is done for example in Barton and Fitch[?], page 254). Comparing these two programs indicates substantial advantages to the Fourier module, as could be expected.

Solving Kepler's Equation

Order	REDUCE	Fourier Module
5	9.16	2.48
6	17.40	4.56
7	33.48	8.06
8	62.76	13.54
9	116.06	21.84
10	212.12	34.54
11	381.78	53.94
12	692.56	82.96
13	1247.54	125.86
14	2298.08	187.20
15	4176.04	275.60
16	7504.80	398.62
17	13459.80	569.26
18	***	800.00
19	***	1116.92
20	***	1536.40

These results were with the linear representation of Fourier series. The tree representation was slightly slower. The ten-fold speed-up for the 13th order is most satisfactory.

16.10.6 A Medium-Sized Problem

Fourier series manipulators are primarily designed for large-scale calculations, but for the demonstration purposes of this project a medium problem is considered. The first stage in calculating the orbit of the Moon using the Delaunay theory (of perturbed elliptic motion for the restricted 3-body problem) is to calculate the energy of the Moon's motion about the Earth — the Hamiltonian of the system. This is the calculation we use for comparisons.

Mathematical Background

The full calculation is described in detail in [?], but a brief description is given here for completeness, and to grasp the extent of the calculation.

Referring to the figure 1 which gives the coordinate system, the basic equations are

$$\begin{aligned} S &= (1 - \gamma^2) \cos(f + g + h - f' - g' - h') + \gamma^2 \cos(f + g - h + f' + g' - h') \\ r &= a(1 - e \cos E) \end{aligned} \quad (16.41)$$

$$l = E - e \sin E \quad (16.42)$$

$$a = \frac{r \mathbf{d}E}{\mathbf{d}l} \quad (16.43)$$

$$\frac{r^2 \mathbf{d}f}{\mathbf{d}l} = a^2 (1 - e^2)^{\frac{1}{2}} \quad (16.44)$$

$$R = m' \frac{a^2}{a'^3} \frac{a'}{r'} \left\{ \left(\frac{r}{a} \right)^2 \left(\frac{a'}{r'} \right)^2 P_2(S) + \left(\frac{a}{a'} \right) \left(\frac{r}{a} \right)^3 \left(\frac{a'}{r'} \right)^3 P_3(S) + \dots \right\} \quad (16.45)$$

There are similar equations to (7) to (10) for the quantities r' , a' , e' , l' , E' and f' which refer to the position of the Sun rather than the Moon. The problem is to calculate the expression R as an expansion in terms of the quantities e , e' , γ , a/a' , l , g , h , l' , g' and h' . The first three quantities are small quantities of the first order, and a/a' is of second order.

The steps required are

1. Solve the Kepler equation (8)
2. Substitute into (7) to give r/a in terms of e and l .
3. Calculate a/r from (9) and f from (10)
4. Substitute for f and f' into S using (6)
5. Calculate R from S , a'/r' and r/a

The program is given in the Appendix.

Results

The Lunar Disturbing function was calculated by a direct coding of the previous sections' mathematics. The program was taken from Barton and Fitch [?] with just small changes to generalise it for any order, and to make it acceptable for Reduce3.4. The Fourier program followed the same pattern, but obviously used the HSUB operation as appropriate and the harmonic integration. It is very similar to the CAMAL program in [?].

The disturbing function was calculated to orders 2, 4 and 6 using Cambridge LISP on an HLH Orion 1/05 (Intergraph Clipper), with the three programs α) Reduce3.4, β) Reduce3.4 + Camal Linear Module and γ) Reduce3.4 + Camal AVL Module.

The timings for CPU seconds (excluding garbage collection time) are summarised the following table:

Order of DDF	Reduce	Camal Linear	Camal Tree
2	23.68	11.22	12.9
4	429.44	213.56	260.64
6	>7500	3084.62	3445.54

If these numbers are normalised so REDUCE calculating the DDF is 100 units for each order the table becomes

Order of DDF	Reduce	Camal Linear	Camal Tree
2	100	47.38	54.48
4	100	49.73	60.69
6	100	<41.13	<45.94

From this we conclude that a doubling of speed is about correct, and although the balanced tree system is slower as the problem size increases the gap between it and the simpler linear system is narrowing.

It is disappointing that the ratio is not better, nor the absolute time less. It is worth noting in this context that Jefferys claimed that the sixth order DDF took 30s on a CDC6600 with TRIGMAN in 1970 [?], and Barton and Fitch took about 1s for the second order DDF on TITAN with CAMAL [?]. A closer look at the relative times for individual sections of the program shows that the substitution case of replacing a polynomial variable by a Fourier series is only marginally faster than the simple REDUCE program. In the DDF program this operation is only used once in a major form, substituting into the Legendre polynomials, which have been previously calculated by Rodrigues formula. This suggests that we replace this with the recurrence relationship.

Making this change actually slows down the normal REDUCE by a small amount but makes a significant change to the Fourier module; it reduces the run time for the 6th order DDF from 3084.62s to 2002.02s. This gives some indication of the problems with benchmarks. What is clear is that the current implementation of substitution of a Fourier series for a polynomial variable is inadequate.

16.10.7 Conclusion

The Fourier module is far from complete. The operations necessary for the solution of Duffing's and Hill's equations are not yet written, although they should not cause much problem. The main deficiency is the treatment of series truncation; at present it relies on the REDUCE WTLEVEL mechanism, and this seems too

coarse for efficient truncation. It would be possible to re-write the polynomial manipulator as well, while retaining the REDUCE syntax, but that seems rather more than one would hope.

The real failure so far is the large time lag between the REDUCE-based system on a modern workstation against a mainframe of 25 years ago running a special system. The CAMAL Disturbing function program could calculate the tenth order with a maximum of 32K words (about 192Kbytes) whereas this system failed to calculate the eighth order in 4Mbytes (taking 2000s before failing). I have in my archives the output from the standard CAMAL test suite, which includes a sixth order DDF on an IBM 370/165 run on 2 June 1978, taking 22.50s and using a maximum of 15459 words of memory for heap — or about 62Kbytes. A rough estimate is that the Orion 1/05 is comparable in speed to the 360/165, but with more real memory and virtual memory.

However, a simple Fourier manipulator has been created for REDUCE which performs between twice and three times the speed of REDUCE using pattern matching. It has been shown that this system is capable of performing the calculations of celestial mechanics, but it still seriously lags behind the efficiency of the specialist systems of twenty years before. It is perhaps fortunate that it was not been possible to compare it with a modern specialist system.

There is still work to do to provide a convenient user interface, but it is intended to develop the system in this direction. It would be pleasant to have again a system of the efficiency of CAMAL(F).

I would like to thank Codemist Ltd for the provision of computing resources for this project, and David Barton who taught me so much about Fourier series and celestial mechanics. Thanks are also due to the National Health Service, without whom this work and paper could not have been produced.

Appendix: The DDF Function

```
array p(n/2+2);
harmonic u,v,w,x,y,z;
weight e=1, b=1, d=1, a=1;

%% Generate Legendre Polynomials to sufficient order
for i:=2:n/2+2 do <<
  p(i):=(h*h-1)^i;
  for j:=1:i do p(i):=df(p(i),h)/(2j)
>>;

%%%%%%%%%%%%%% Step1: Solve Kepler equation
bige := fourier 0;
```



```

for k:=1:n do <<
  wtlevel k;
  bige:=fourier e * hsub(fourier(sin u), u, u, bige, k);
>>;

%% Ensure we do not calculate things of too high an order
wtlevel n;

%%%%%%%%%%%%%% Step 2: Calculate r/a in terms of e and l
dd:=-e*e; hh:=3/2; j:=1; cc := 1;
for i:=1:n/2 do <<
  j:=i*j; hh:=hh-1; cc:=cc+hh*(dd^i)/j
>>;
bb:=hsub(fourier(1-e*cos u), u, u, bige, n);
aa:=fourier 1+hdiff(bige,u); ff:=hint(aa*aa*fourier cc,u);

%%%%%%%%%%%%%% Step 3: a/r and f
uu := hsub(bb,u,v); uu:=hsub(uu,e,b);
vv := hsub(aa,u,v); vv:=hsub(vv,e,b);
ww := hsub(ff,u,v); ww:=hsub(ww,e,b);

%%%%%%%%%%%%%% Step 4: Substitute f and f' into S
yy:=ff-ww; zz:=ff+ww;
xx:=hsub(fourier((1-d*d)*cos(u)),u,u-v+w-x-y+z,yy,n)+
  hsub(fourier(d*d*cos(v)),v,u+v+w+x+y-z,zz,n);

%%%%%%%%%%%%%% Step 5: Calculate R
zz:=bb*vv; yy:=zz*zz*vv;

on fourier;
for i := 2:n/2+2 do <<
  wtlevel n+4-2i; p(i) := hsub(p(i), h, xx) >>;

wtlevel n;
for i:=n/2+2 step -1 until 3 do
  p(n/2+2):=fourier(a*a)*zz*p(n/2+2)+p(i-1);
yy*p(n/2+2);

```

16.11 CDIFF: A package for the geometry of Differential Equations

Authors: P. Gragert, P.H.M. Kersten, G. Post and G. Roelofs, R. Vitolo.

16.12 CHANGEVR: Change of Independent Variable(s) in DEs

This package provides facilities for changing the independent variables in a differential equation. It is basically the application of the chain rule.

Author: G. Üçoluk.

16.12.1 Introduction

The mathematics behind the change of independent variable(s) in differential equations is quite straightforward. It is basically the application of the chain rule. If the dependent variable of the differential equation is F , the independent variables are x_i and the new independent variables are u_i (where $i=1\dots n$) then the first derivatives are:

$$\frac{\partial F}{\partial x_i} = \frac{\partial F}{\partial u_j} \frac{\partial u_j}{\partial x_i}$$

We assumed Einstein's summation convention. Here the problem is to calculate the $\partial u_j / \partial x_i$ terms if the change of variables is given by

$$x_i = f_i(u_1, \dots, u_n)$$

The first thought might be solving the above given equations for u_j and then differentiating them with respect to x_i , then again making use of the equations above, substituting new variables for the old ones in the calculated derivatives. This is not always a preferable way to proceed. Mainly because the functions f_i may not always be easily invertible. Another approach that makes use of the Jacobian is better. Consider the above given equations which relate the old variables to the new ones. Let us differentiate them:

$$\begin{aligned} \frac{\partial x_j}{\partial x_i} &= \frac{\partial f_j}{\partial x_i} \\ \delta_{ij} &= \frac{\partial f_j}{\partial u_k} \frac{\partial u_k}{\partial x_i} \end{aligned}$$

The first derivative is nothing but the (j, k) th entry of the Jacobian matrix.

So if we speak in matrix language

$$\mathbf{1} = \mathbf{J} \cdot \mathbf{D}$$

where we defined the Jacobian

$$\mathbf{J}_{ij} \triangleq \frac{\partial f_i}{\partial u_j}$$

and the matrix of the derivatives we wanted to obtain as

$$\mathbf{D}_{ij} \triangleq \frac{\partial u_i}{\partial x_j}.$$

If the Jacobian has a non-vanishing determinant then it is invertible and we are able to write from the matrix equation above:

$$\mathbf{D} = \mathbf{J}^{-1}$$

so finally we have what we want

$$\frac{\partial u_i}{\partial x_j} = [\mathbf{J}^{-1}]_{ij}$$

The higher derivatives are obtained by the successive application of the chain rule and using the definitions of the old variables in terms of the new ones. It

can be easily verified that the only derivatives that are needed to be calculated are the first order ones which are obtained above.

16.12.2 How to Use CHANGEVR

This facility requires the matrix package to be present in the session. So if it is not autoloaded in your REDUCE implementation, say

```
LOAD_PACKAGE MATRIX;
```

in the REDUCE environment. Then load CHANGEVR by the statement:

```
LOAD_PACKAGE CHANGEVR$
```

Now the REDUCE function CHANGEVAR is ready to use. **Note: The package is named CHANGEVR, but the function has the name CHANGEVAR.** The function CHANGEVAR has (at least) four different arguments. Here we give a list them:

- **FIRST ARGUMENT**

Is a list of the dependent variables of the differential equation. They shall be enclosed in a pair of curly braces and separated by commas. If there is only one dependent variable there is no need for the curly braces.

- **SECOND ARGUMENT**

Is a list of the **new** independent variables. Similar to what is said for the first argument, these shall also be separated by commas, enclosed in curly braces and the curly braces can be omitted if there is only one new variable.

- **THIRD ARGUMENT**

Is a list of equations separated by commas, where each of the equation is of the form

$$\text{old variable} = \text{a function in new variables}$$

The left hand side cannot be a non-kernel structure. In this argument the functions which give the old variables in terms of the new ones are introduced. It is possible to omit totally the curly braces which enclose the list. **Please note that only for this argument it is allowed to omit the curly braces even if the list has more than one items.**

- **LAST ARGUMENT**

Is a list of algebraic expressions which evaluates to differential equations, separated by commas, enclosed in curly braces. So, variables in which differential equations are already stored may be used freely. Again it is possible to omit the curly braces if there is only **one** differential equation.

If the last argument is a list then the result of CHANGEVAR is also a list.

It is possible to display the entries of the inverse Jacobian, explained in the introduction. To do so, turn ON the flag DISPJACOBIAN by a statement:

```
ON DISPJACOBIAN;
```

16.12.3 AN EXAMPLE.....The 2-dim. Laplace Equation

The 2-dimensional Laplace equation in cartesian coordinates is:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

Now assume we want to obtain the polar coordinate form of Laplace equation. The change of variables is:

$$x = r \cos \theta, \quad y = r \sin \theta$$

The solution using CHANGEVAR (of course after it is properly loaded) is as follows

```
CHANGEVAR ({u}, {r, theta}, {x=r*cos theta, y=r*sin theta},
           {df(u(x,y), x, 2) + df(u(x,y), y, 2)} )
```

Here we could omit the curly braces in the first and last arguments (because those lists have only one member) and the curly braces in the third argument (because they are optional), but you cannot leave off the curly braces in the second argument. So one could equivalently write

```
CHANGEVAR(u, {r, theta}, x=r*cos theta, y=r*sin theta,
          df(u(x, y), x, 2) + df(u(x, y), y, 2) )
```

If you have tried out the above example, you will notice that the denominator contains a $\cos^2 \theta + \sin^2 \theta$ which is actually equal to 1. This has of course nothing to do with the `CHANGEVAR` facility introduced here. One has to overcome these pattern matching problems by the conventional methods `REDUCE` provides (a `LET` statement, for example, will fix it).

Secondly you will notice that your $u(x, y)$ operator has changed to $u(r, \theta)$ in the result. Nothing magical about this. That is just what we do with pencil and paper. $u(r, \theta)$ represents the transformed dependent variable.

16.12.4 ANOTHER EXAMPLE.....An Euler Equation

Consider a differential equation which is of Euler type, for instance:

$$x^3 y''' - 3x^2 y'' + 6xy' - 6y = 0$$

Where prime denotes differentiation with respect to x . As is well known, Euler type of equations are solved by a change of variable:

$$x = e^u$$

So our `CHANGEVAR` call reads as follows:

```
CHANGEVAR(y, u, x=e**u, x**3*df(y(x), x, 3) -
          3*x**2*df(y(x), x, 2) + 6*x*df(y(x), x) - 6*y(x) )
```

16.13 COMPACT: Package for compacting expressions

COMPACT is a package of functions for the reduction of a polynomial in the presence of side relations. COMPACT applies the side relations to the polynomial so that an equivalent expression results with as few terms as possible. For example, the evaluation of

```
compact (s*(1-sin x^2)+c*(1-cos x^2)+sin x^2+cos x^2,
        {cos x^2+sin x^2=1});
```

yields the result

$$\text{SIN}(X)^2 * C + \text{COS}(X)^2 * S + 1.$$

Author: Anthony C. Hearn.

16.14 CRACK: Solving overdetermined systems of PDEs or ODEs

CRACK is a package for solving overdetermined systems of partial or ordinary differential equations (PDEs, ODEs). Examples of programs which make use of CRACK (finding symmetries of ODEs/PDEs, first integrals, an equivalent Lagrangian or a "differential factorization" of ODEs) are included. The application of symmetries is also possible by using the APPLYSYM package.

Authors: Andreas Brand, Thomas Wolf.

16.15 CVIT: Fast calculation of Dirac gamma matrix traces

This package provides an alternative method for computing traces of Dirac gamma matrices, based on an algorithm by Cvitanovich that treats gamma matrices as 3-j symbols.

Authors: V.Ilyin, A.Kryukov, A.Rodionov, A.Taranov.

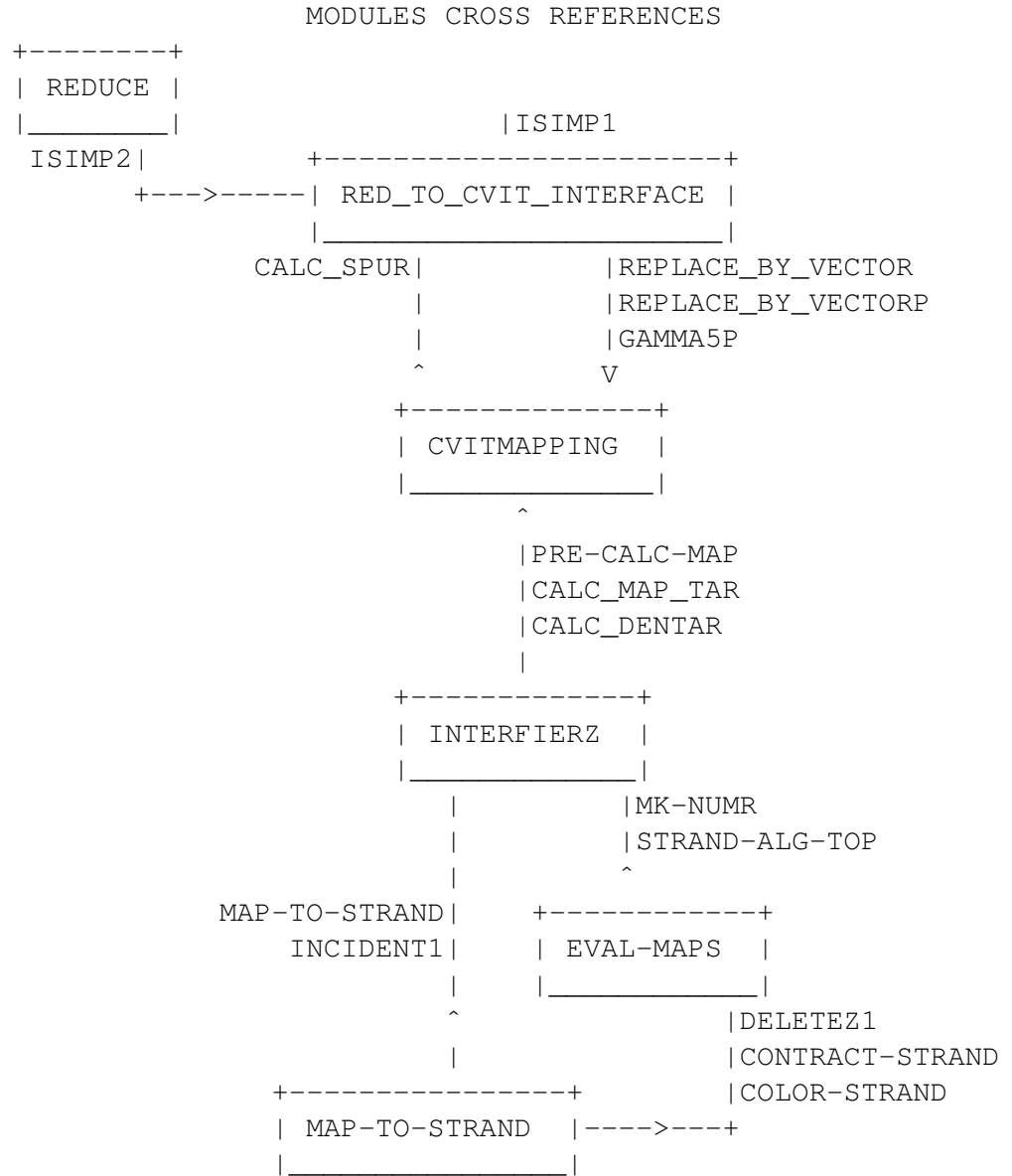
Abstract

In modern high energy physics the calculation of Feynman diagrams are still very important. One of the difficulties of these calculations are trace calculations. So the calculation of traces of Dirac's γ -matrices were one of first task of computer algebra systems. All available algorithms are based on the fact that gamma-matrices constitute a basis of a Clifford algebra:

$$\{G_m, G_n\} = 2g_{mn}.$$

We present the implementation of an alternative algorithm based on treating of gamma-matrices as 3-j symbols (details may be found in [1,2]).

The program consists of 5 modules described below.



Requires of REDUCE version: 3.2, 3.3.

Module RED_TO_CVIT_INTERFACE

Author: A.P.Kryukov

Purpose: interface REDUCE and CVIT package

RED_TO_CVIT_INTERFACE module is intended for connection of REDUCE with main module of CVIT package. The main idea is to preserve standard REDUCE syntax for high energy calculations. For realization of this we redefine SYMBOLIC PROCEDURE ISIMP1 from HEPHys module of REDUCE system.

After loading CVIT package user may use switch CVIT which is ON by default. If switch CVIT is OFF then calculations of Diracs matrices traces are performed using standard REDUCE facilities. If CVIT switch is ON then CVIT package will be active.

RED_TO_CVIT_INTERFACE module performs some primitive simplification and control input data independently. For example it remove $G_m G_m$, check parity of the number of Dirac matrices in each trace *etc.* There is one principal restriction concerning G5-matrix. There are no closed form for trace in non-integer dimension case when trace include G5-matrix. The next restriction is that if the space-time dimension is integer then it must be even (2,4,6,...). If these and other restrictions are violated then the user get corresponding error message. List of messages is included.

LIST OF IMPORTED FUNCTIONS

Function	From module
ISIMP2	HEPHys
CALC_SPUR	CVITMAPPING

LIST OF EXPORTED FUNCTION

Function	To module
ISIMP1	HEPHys (redefine)
REPLACE_BY_VECTOR	EVAL_MAP
REPLACE_BY_VECTORP	EVAL__MAP
GAMMA5P	CVITMAPPING, EVAL_MAP

Module CVITMAPPING

Author: A.Ya.Rodionov

Purpose: graphs reduction

CVITMAPPING module is intended for diagrams calculation according to Cvitanovic - Kennedy algorithm. The top function of this module CALC_SPUR is

called from RED_TO_CVIT_INTERFACE interface module. The main idea of the algorithm consists in diagram simplification according to rules (1.9') and (1.14) from [1]. The input data - trace of Diracs gamma matrices (G-matrices) has a form of a list of identifiers lists with cyclic order. Some of identifiers may be identical. In this case we assume summation over dummy indices. So trace $\text{Sp}(\text{GbGr}).\text{Sp}(\text{GwGbGcGwGcGr})$ is represented as list ((b r) (w b c w c r)).

The first step is to transform the input data to “map” structure and then to reduce the map to a “simple” one. This transformation is made by function TRANSFORM_MAP_ (top function). Transformation is made in three steps. At the first step the input data are transformed to the internal form - a map (by function PREPARE_MAP_). At the second step a map is subjected to Fierz transformations (1.14) (function MK_SIMPLE_MAP_). At this step of optimization can be maid (if switch CVITOP is on) by function MK_FIRZ_OP. In this case Fierzing starts with linked vertices with minimal distance (number of vertices) between them. After Fierz transformations map is further reduced by vertex simplification routine MK_SIMPLE_VERTEX using (1.9'). Vertices reduced to primitive ones, that is to vertices with three or less edges. This is the last (third) step in transformation from input to internal data.

The next step is optional. If switch CVITBTR is on factorisation of bubble (function FIND_BUBBLES1) and triangle (function FIND_TRIANGLES1) submaps is made. This factorisation is very efficient for “wheel” diagrams and unnecessary for “lattice” diagrams. Factorisation is made recursively by substituting composed edges for bubbles and composed vertices for triangles. So check (function SORT_ATLAS) must be done to test possibility of future marking procedure. If the check fails then a new attempt to reorganize atlas (so we call complicated structure with consists of MAP, COEFFicient and DENOMinator) is made. This cause backtracking (but very seldom). Backtracking can be traced by turning on switch CVITRACE. FIND_BUBLTR is the top function of this program's branch.

Then atlases must be prepared (top function WORLD_FROM_ATLAS) for final algebraic calculations. The resulted object called “world” consists of edges names list (EDGELIST), their marking variants (VARIANTS) and WORLD1 structure. WORLD1 structure differs from WORLD structure in one point. It contains MAP2 structure instead of MAP structure. MAP2 is very complicated structure and consist of VARIANTS, marking plan and GSTRAND. (GSTRAND constructed by PRE!-CALC!-MAP_ from INTERFIERZ module.) By marking we understand marking of edges with numbers according to Cvitanovic - Kennedy algorithm.

The last step is performed by function CALC_WORLD. At this step algebraic calculations are done. Two functions CALC_MAP_TAR and CALC_DENTAR from INTERFIERZ module make algebraic expressions in the prefix form. This expressions are further simplified by function REVAL. This is the REDUCE system general function for algebraic expressions simplification. REVAL and SIMP ! * are the only REDUCE functions used in this module.

There are also some functions for printing several internal structures: PRINT_ATLAS, PRINT_VERTEX, PRINT_EDGE, PRINT_COEFF, PRINT_DENOM. These functions can be used for debugging.

If an error occurs in module CVITMAPPING the error message “ERROR IN MAP_CREATING_ROUTINES” is displayed. Error has number 55. The switch CVITERROR allows to give full information about error: name of function where error occurs and names and values of function’s arguments. If CVITERROR switch is on and backtracking fails message about error in SORT_ATLAS function is printed. The result of computation however will be correct because in this case factorized structure is not used. This happens extremely seldom.

List of imported function

function	from module
REVAL	REDUCE
SIMP!*	REDUCE
CALC_MAP_TAR	INTERFIERZ
CALC_DENTAR	INTERFIERZ
PRE!-CALC!-MAP_	INTERFIERZ
GAMMA5P	RED_TO_CVIT_INTERFACE

List of exported function

function	to module
CALC_SPUR	REDUCE - CVIT interface

Data structure

WORLD	::=	(EDGELIST, VARIANTS, WORLD1)
WORLD1	::=	(MAP2, COEFF, DENOM)
MAP2	::=	(MAPS, VARIANTS, PLAN)
MAPS	::=	(EDGEPAIR . GSTRAND)
MAP1	::=	(EDGEPAIR . MAP)
MAP	::=	list of VERTICES (unordered)
EDGEPAIR	::=	(OLDEDGELIST . NEWEDGELIST)
COEFF	::=	list of WORLDS (unordered)
ATLAS	::=	(MAP, COEFF, DENOM)
GSTRAND	::=	(STRAND*, MAP, TADPOLES, DELTAS)
VERTEX	::=	list of EDGES (with cyclic order)
EDGE	::=	(NAME, PROPERTY, TYPE)

```

NAME      ::=  ATOM
PROPERTY  ::=  (FIRSTPAIR . SECONDPAIR)
TYPE      ::=  T or NIL
-----
*Define in module MAP!-TO!-STRAND.

```

Modules INTERFIERZ, EVAL_MAPS, AND MAP-TO-STRAND.

Author: A.Taranov

Purpose: evaluate single Map

Module INTERFIERZ exports to module CVITMAPPING three functions: PRE-CALC-MAP_, CALC-MAP_TAR, CALC-DENTAR.

Function PRE-CALC-MAP_ is used for preliminary processing of a map. It returns a list of the form (STRAND NEWMAP TADEPOLES DELTAS) where STRAND is strand structure described in MAP-TO-STRAND module. NEWMAP is a map structure without “tadepoles” and “deltas”. “Tadepole” is a loop connected with map with only one line (edge). “Delta” is a single line disconnected from a map. TADEPOLES is a list of “tadepole” submaps. DELTAS is a list (CONS E1 E2) where E1 and E2 are

Function CALC_MAP_TAR takes a list of the same form as returned by PRE-CALC-MAP_, a-list, of the form (... edge . weight ...) and returns a prefix form of algebraic expression corresponding to the map numerator.

Function CALC-DENTAR returns a prefix form of algebraic expression corresponding to the map denominator.

Module EVAL-MAP exports to module INTERFIERZ functions MK-NUMR and STRAND-ALG-TOP.

Function MK-NUMR returns a prefix form for some combinatorial coefficient (Pohhammer symbol).

Function STRAND-ALG-TOP performs an actual computation of a prefix form of algebraic expression corresponding to the map numerator. This computation is based on a “strand” structure constructed from the “map” structure.

Module MAP-TO-STRAND exports functions MAP-TO-STRAND, INCIDENT1 to module INTERFIERZ and functions DELETEZ1, CONTRACT-STRAND, COLOR-STRAND to module EVAL-MAPS.

Function INCIDENT1 is a selector in “strand” structure. DELETEZ1 performs auxiliary optimization of “strand”. MAP-TO-STRAND transforms “map” to “strand” structure. The latter is describe in program module.

CONTRACT-STRAND do strand vertex simplifications of “strand” and COLOR-STRAND finishes strand generation.

```

                Description of STRAND  data structure.
STRAND ::= <LIST OF VERTEX>
VERTEX ::= <NAME> . (<LIST OF ROAD> <LIST OF ROAD>)
ROAD   ::= <ID> . NUMBER
NAME   ::= NUMBER

```

LIST OF MESSAGES

- **CALC_SPUR: <vecdim> IS NOT EVEN SPACE-TIME DIMENSION** The dimension of space-time <vecdim> is integer but not even. Only even numeric dimensions are allowed.
- **NOSPUR NOT YET IMPLEMENTED** Attempt to calculate trace when NOSPUR switch is on. This facility is not implemented now.
- **G5 INVALID FOR VECDIM NEQ 4** Attempt to calculate trace with gamma5-matrix for space-time dimension not equal to 4.
- **CALC_SPUR: <expr> HAS NON-UNIT DENOMINATOR** The $\langle \text{expr} \rangle$ has non-unit denominator.
- **THREE INDICES HAVE NAME <name>** There are three indices with equal names in evaluated expression.

List of switches		
switch	default	comment
CVIT	ON	If it is on then use Kennedy-Cvitanovic algorithm else use standard facilities.
CVITOP	OFF	Fierz optimization switch
CVITBTR	ON	Bubbles and triangles factorisation switch
CVITRACE	OFF	Backtracking tracing switch

Functions cross references*.

CALC_SPUR

```

|
+-->SIMP!* (REDUCE)
|
+-->CALC_SPUR0
|
|--->TRANSFORM_MAP_
|   |
|   |--->MK_SIMPLE_VERTEX
|   +--->MK_SIMPLE_MAP_
|       |
|       +--->MK_SIMPLE_MAP_1
|           |
|           +--->MK_FIERS_OP
|
|--->WORLD_FROM_ATLAS
|   |
|   +--->CONSTR_WORLDS
|       |
|       +----->MK_WORLD1
|           |
|           +--->MAP_2_FROM_MAP_1
|               |
|               |--->MARK_EDGES
|               +--->MAP_1_TO_STRAND
|                   |
|                   +--->PRE!-CALC!-MAP_
|                           (INTERFIRZ)
|
|--->CALC_WORLD
|   |
|   |--->CALC!-MAP_TAR (INTERFIRZ)
|   |--->CALC!-DENTAR (INTERFIRZ)
|   +--->REVAL (REDUCE)
|
+---->FIND_BUBLTR
|
+---->FIND_BUBLTR0
|
|--->SORT_ATLAS
+---->FIND_BUBLTR1
|
|--->FIND_BUBLES1
+---->FIND_TRIANGLES1

```

*Unmarked functions are from CVITMPPING module.

References

- 1. Ilyin V.A., Kryukov A.P., Rodionov A.Ya., Taranov A.Yu. Fast algorithm for calculation of Diracs gamma-matrices traces. SIGSAM Bull., 1989, v.23, no.4, pp.15-24.
- 2. Kennedy A.D. Phys.Rev., 1982, D26, p.1936.

Keywords

REDUCE, GAMMA-MATRIX, TRACE, SPACE-TIME DIMENSION, HIGH ENERGY PHYSICS.

16.16 DEFINT: A definite integration interface

This package finds the definite integral of an expression in a stated interval. It uses several techniques, including an innovative approach based on the Meijer G-function, and contour integration.

Authors: Kerry Gaskell, Stanley M. Kameny, Winfried Neun.

16.16.1 Introduction

This documentation describes part of REDUCE's definite integration package that is able to calculate the definite integrals of many functions, including several special functions. There are other parts of this package, such as Stan Kameny's code for contour integration, that are not included here. The integration process described here is not the more normal approach of initially calculating the indefinite integral, but is instead the rather unusual idea of representing each function as a Meijer G-function (a formal definition of the Meijer G-function can be found in [1]), and then calculating the integral by using the following Meijer G integration formula.

$$\int_0^\infty x^{\alpha-1} G_{uv}^{st} \left(\sigma x \left| \begin{matrix} (c_u) \\ (d_v) \end{matrix} \right. \right) G_{pq}^{mn} \left(\omega x^{l/k} \left| \begin{matrix} (a_p) \\ (b_q) \end{matrix} \right. \right) dx = k G_{kl}^{ij} \left(\xi \left| \begin{matrix} (g_k) \\ (h_l) \end{matrix} \right. \right) \quad (16.46)$$

The resulting Meijer G-function is then retransformed, either directly or via a hypergeometric function simplification, to give the answer. A more detailed account of this theory can be found in [2].

16.16.2 Integration between zero and infinity

As an example, if one wishes to calculate the following integral

$$\int_0^\infty x^{-1} e^{-x} \sin(x) dx$$

then initially the correct Meijer G-functions are found, via a pattern matching process, and are substituted into eq. 16.46 to give

$$\sqrt{\pi} \int_0^\infty x^{-1} G_{01}^{10} \left(x \left| \begin{matrix} \cdot \\ 0 \end{matrix} \right. \right) G_{02}^{10} \left(\frac{x^2}{4} \left| \begin{matrix} \cdot \cdot \\ \frac{1}{2} 0 \end{matrix} \right. \right) dx$$

The cases for validity of the integral are then checked. If these are found to be satisfactory then the formula is calculated and we obtain the following Meijer G-function

$$G_{22}^{12} \left(1 \left| \begin{matrix} \frac{1}{2} & 1 \\ \frac{1}{2} & 0 \end{matrix} \right. \right)$$

This is reduced to the following hypergeometric function

$${}_2F_1\left(\frac{1}{2}, 1; \frac{3}{2}; -1\right)$$

which is then calculated to give the correct answer of

$$\frac{\pi}{4}$$

The above formula (1) is also true for the integration of a single Meijer G-function by replacing the second Meijer G-function with a trivial Meijer G-function.

A list of numerous particular Meijer G-functions is available in [1].

16.16.3 Integration over other ranges

Although the description so far has been limited to the computation of definite integrals between 0 and infinity, it can also be extended to calculate integrals between 0 and some specific upper bound, and by further extension, integrals between any two bounds. One approach is to use the Heaviside function, i.e.

$$\int_0^\infty x^2 e^{-x} H(1-x) dx = \int_0^1 x^2 e^{-x} dx$$

Another approach, again not involving the normal indefinite integration process, again uses Meijer G-functions, this time by means of the following formula

$$\int_0^y x^{\alpha-1} G_{pq}^{mn} \left(\sigma x \left| \begin{matrix} (a_u) \\ (b_v) \end{matrix} \right. \right) dx = y^\alpha G_{p+1\ q+1}^{m\ n+1} \left(\sigma y \left| \begin{matrix} (a_1..a_n, 1-\alpha, a_{n+1}..a_p) \\ (b_1..b_m, -\alpha, b_{m+1}..b_q) \end{matrix} \right. \right) \quad (16.47)$$

For a more detailed look at the theory behind this see [2].

For example, if one wishes to calculate the following integral

$$\int_0^y \sin(2\sqrt{x}) dx$$

then initially the correct Meijer G-function is found, by a pattern matching process, and is substituted into eq. 16.47 to give

$$\int_0^y G_{02}^{10} \left(x \left| \begin{matrix} \cdot \\ \frac{1}{2} \end{matrix} \right. 0 \right) dx$$

which then in turn gives

$$y G_{13}^{11} \left(y \left| \begin{matrix} 0 \\ \frac{1}{2} -1 \end{matrix} \right. 0 \right) dx$$

and returns the result

$$\frac{\sqrt{\pi} J_{3/2}(2\sqrt{y}) y}{y^{1/4}}$$

16.16.4 Using the definite integration package

To use this package, you must first load it by the command

```
load_package defint;
```

Definite integration is then possible using the `int` command with the syntax:

```
INT(EXPRN:algebraic,VAR:kernel,LOW:algebraic,UP:algebraic)
:algebraic.
```

where `LOW` and `UP` are the lower and upper bounds respectively for the definite integration of `EXPRN` with respect to `VAR`.

Examples

$$\int_0^\infty e^{-x} dx$$

```
int(e^(-x),x,0,infinity);
```

1

$$\int_0^\infty x \sin(1/x) dx$$

```
int (x*sin (1/x) , x, 0, infinity) ;
```

```

      1
INT (X*SIN (---) , X, 0, INFINITY)
      X

```

$$\int_0^{\infty} x^2 \cos(x) e^{-2x} dx$$

```
int (x^2*cos (x) *e^ (-2*x) , x, 0, infinity) ;
```

```

      4
-----
    125

```

$$\int_0^{\infty} x e^{-1/2x} H(1-x) dx = \int_0^1 x e^{-1/2x} dx$$

```
int (x*e^ (-1/2x) *Heaviside (1-x) , x, 0, infinity) ;
```

```

      2*(2*SQRT(E)  - 3)
-----
      SQRT(E)

```

$$\int_0^1 x \log(1+x) dx$$

```
int (x*log (1+x) , x, 0, 1) ;
```

```

      1
-----
      4

```

$$\int_0^y \cos(2x) dx$$

```
int (cos (2x) , x, y, 2y) ;
```

```

      SIN(4*Y)  -  SIN(2*Y)
-----

```

16.16.5 Integral Transforms

A useful application of the definite integration package is in the calculation of various integral transforms. The transforms available are as follows:

- Laplace transform
- Hankel transform
- Y-transform
- K-transform
- StruveH transform
- Fourier sine transform
- Fourier cosine transform

Laplace transform

The Laplace transform

$$f(s) = \mathcal{L} \{F(t)\} = \int_0^{\infty} e^{-st} F(t) dt$$

can be calculated by using the `laplace_transform` command.

This requires as parameters

- the function to be integrated
- the integration variable.

For example

$$\mathcal{L} \{e^{-at}\}$$

is entered as

```
laplace_transform(e^(-a*x), x);
```

and returns the result

$$\frac{1}{s + a}$$

Hankel transform

The Hankel transform

$$f(\omega) = \int_0^{\infty} F(t) J_{\nu}(2\sqrt{\omega t}) dt$$

can be calculated by using the `hankel_transform` command e.g.

```
hankel_transform(f(x), x);
```

This is used in the same way as the `laplace_transform` command.

Y-transform

The Y-transform

$$f(\omega) = \int_0^{\infty} F(t) Y_{\nu}(2\sqrt{\omega t}) dt$$

can be calculated by using the `Y_transform` command e.g.

```
Y_transform(f(x), x);
```

This is used in the same way as the `laplace_transform` command.

K-transform

The K-transform

$$f(\omega) = \int_0^{\infty} F(t) K_{\nu}(2\sqrt{\omega t}) dt$$

can be calculated by using the `K_transform` command e.g.

```
K_transform(f(x), x);
```

This is used in the same way as the `laplace_transform` command.

StruveH transform

The StruveH transform

$$f(\omega) = \int_0^{\infty} F(t) \operatorname{StruveH}(\nu, 2\sqrt{\omega t}) dt$$

can be calculated by using the `struveh_transform` command e.g.

```
struveh_transform(f(x), x);
```

This is used in the same way as the `laplace_transform` command.

Fourier sine transform

The Fourier sine transform

$$f(s) = \int_0^{\infty} F(t) \sin(st) dt$$

can be calculated by using the `fourier_sin` command e.g.

```
fourier_sin(f(x), x);
```

This is used in the same way as the `laplace_transform` command.

Fourier cosine transform

The Fourier cosine transform

$$f(s) = \int_0^{\infty} F(t) \cos(st) dt$$

can be calculated by using the `fourier_cos` command e.g.

```
fourier_cos(f(x), x);
```

This is used in the same way as the `laplace_transform` command.

16.16.6 Additional Meijer G-function Definitions

The relevant Meijer G representation for any function is found by a pattern-matching process which is carried out on a list of Meijer G-function definitions. This list, although extensive, can never hope to be complete and therefore the user may wish to add more definitions. Definitions can be added by adding the following lines:


```
defint_choose(f(~x), ~var => f1(n, x);

symbolic putv(mellin!-transforms!*, n, '
              ((() (m n p q) (ai) (bj) (C) (var))));
```

where $f(x)$ is the new function, $i = 1..p$, $j=1..q$, C = a constant, var = variable, n = an indexing number.

For example when considering $\cos(x)$ we have

Meijer G representation -

$$\sqrt{\pi} G_{02}^{10} \left(\frac{x^2}{4} \left| \begin{array}{c} \cdot \cdot \\ 0 \frac{1}{2} \end{array} \right. \right) dx$$

Internal definite integration package representation -

```
defint_choose(cos(~x), ~var) => f1(3, x);
```

where 3 is the indexing number corresponding to the 3 in the following formula

```
symbolic putv(mellin!-transforms!*, 3, '
              ((() (1 0 0 2) () (nil (quotient 1 2))
                 (sqrt pi) (quotient (expt x 2) 4))));
```

or the more interesting example of $J_n(x)$:

Meijer G representation -

$$G_{02}^{10} \left(\frac{x^2}{4} \left| \begin{array}{c} \cdot \cdot \\ \frac{n}{2} \frac{-n}{2} \end{array} \right. \right) dx$$

Internal definite integration package representation -

```
defint_choose(besselj(~n, ~x), ~var) => f1(50, x, n);

symbolic putv(mellin!-transforms!*, 50, '
              ((n) (1 0 0 2) () ((quotient n 2)
                                (minus quotient n 2)) 1
                                (quotient (expt x 2) 4))));
```

16.16.7 The print_conditions function

The required conditions for the validity of the transform integrals can be viewed using the following command:

```
print_conditions() .
```

For example after calculating the following laplace transform

```
laplace_transform(x^k,x);
```

using the `print_conditions` command would produce

```
repart(sum(ai) - sum(bj)) + 1/2 (q + 1 - p) > (q - p) repart(s)

and ( - min(repart(bj)) < repart(s)) < 1 - max(repart(ai))

or mod(arg(eta)) = pi * delta

or ( - min(repart(bj)) < repart(s)) < 1 - max(repart(ai))

or mod(arg(eta)) < pi * delta
```

where

$$\begin{aligned} \delta &= s + t - \frac{u-v}{2} \\ \eta &= 1 - \alpha(v-u) - \mu - \rho \\ \mu &= \sum_{j=1}^q b_j - \sum_{i=1}^p a_i + \frac{p-q}{2} + 1 \\ \rho &= \sum_{j=1}^v d_j - \sum_{i=1}^u c_i + \frac{u-v}{2} + 1 \\ s, t, u, v, p, q, \alpha &\text{ as in (1)} \end{aligned}$$

16.16.8 Acknowledgements

I would like to thank Victor Adamchik whose implementation of the definite integration package for REDUCE is vital to this interface.

Bibliography

- [1] A.P. Prudnikov, Yu.A. Brychkov and O.I. Marichev, *Integrals and Series, Volume 3: More Special Functions* Gordon and Breach Science Publishers (1990)
- [2] V.S. Adamchik and O.I. Marichev, *The Algorithm for Calculating Integrals of Hypergeometric Type Functions and its Realization in Reduce System* from ISSAC 90: Symbolic and Algebraic Computation Addison-Wesley Publishing Company (1990)

- [3] Yudell L. Luke, *The Special Functions and their Approximations, Volume 1* Academic Press (1969).

16.17 DESIR: Differential linear homogeneous equation solutions in the neighborhood of irregular and regular singular points

This package enables the basis of formal solutions to be computed for an ordinary homogeneous differential equation with polynomial coefficients over \mathbb{Q} of any order, in the neighborhood of zero (regular or irregular singular point, or ordinary point).

Authors: C. Dicrescenzo, F. Richard-Jung, E. Tournier.

Differential linear homogenous Equation Solutions in the
neighbourhood of Irregular and Regular singular points

Version 3.1 - Septembre 89

Groupe de Calcul Formel de Grenoble
laboratoire TIM3

(C. Dicrescenzo, F. Richard-Jung, E. Tournier)

E-mail: dicresc@afp.imag.fr

16.17.1 INTRODUCTION

This software enables the basis of formal solutions to be computed for an ordinary homogeneous differential equation with polynomial coefficients over \mathbb{Q} of any order, in the neighbourhood of zero (regular or irregular singular point, or ordinary point).

Tools have been added to deal with equations with a polynomial right-hand side, parameters and a singular point not to be found at zero.

This software can be used in two ways :

- direct (DELIRE procedure)
- interactive (DESIR procedure)

The basic procedure is the DELIRE procedure which enables the solutions of a linear homogeneous differential equation to be computed in the neighbourhood

of zero.

The DESIR procedure is a procedure without argument whereby DELIRE can be called without preliminary treatment to the data, that is to say, in an interactive autonomous way. This procedure also proposes some transformations on the initial equation. This allows one to start comfortably with an equation which has a non zero singular point, a polynomial right-hand side and parameters.

This document is a succinct user manual. For more details on the underlying mathematics and the algorithms used, the reader can refer to :

E. Tournier : Solutions formelles d'équations différentielles - Le logiciel de calcul formel DESIR.

These d'Etat de l'Université Joseph Fourier (Grenoble - avril 87).

He will find more precision on use of parameters in :

F. Richard-Jung : Representation graphique de solutions d'équations différentielles dans le champ complexe.

These de l'Université Louis Pasteur (Strasbourg - septembre 88).

16.17.2 FORMS OF SOLUTIONS

We have tried to represent solutions in the simplest form possible. For that, we have had to choose different forms according to the complexity of the equation (parameters) and the later use we shall have of these solutions.

"general solution" = {....., { split_sol , cond },....}

cond = list of conditions or empty list (if there is no condition) that parameters have to verify such that split_sol is in the basis of solutions. In fact, if there are parameters, basis of solutions can have different expressions according to the values of parameters. (Note : if cond={}, the list "general solution" has one element only.)

split_sol = { *q*, *ram*, *polysol*, *r* }
(" split solution " enables precise information on the solution to be obtained immediately)

The variable in the differential operator being *x*, solutions are expressed in respect to a new variable *xt*, which is a fractional power of *x*, in the following way :

q : polynomial in $1/xt$ with complex coefficients
 ram : $xt = x^{ram}$ ($1/ram$ is an integer)
 $polysol$: polynomial in $\log(xt)$ with formal series in xt coefficients
 r : root of a complex coefficient polynomial ("indicial equation").

"standard solution" = $e^{qx} x^{r*ram} polysolx$

qx and $polysolx$ are q and $polysol$ expressions in which xt has been replaced by x^{ram}

N.B. : the form of these solutions is simplified according to the nature of the point zero.

- if 0 is a regular singular point : the series appearing in $polysol$ are convergent, $ram = 1$ and $q = 0$.
- if 0 is a regular point, we also have : $polysol$ is constant in $\log(xt)$ (no logarithmic terms).

16.17.3 INTERACTIVE USE

To call the procedure : `desir();`
`solution:=desir();`

The DESIR procedure computes formal solutions of a linear homogeneous differential equation in an interactive way.

In this equation the variable *must be* x .

The procedure requires the order and the coefficients of the equation, the names of parameters if there are any, then if the user wants to transform this equation and how (for example to bring back a singular point to zero see procedures `changehom`, `changevar`, `changeonc` -).

This procedure **DISPLAYS** the solutions and **RETURNS** a list of general term { `lcoeff`, { ..., { `general_solution` }, ..., } }. The number of elements in this list is linked to the number of transformations requested :

- * `lcoeff` : list of coefficients of the differential equation
- * `general_solution` : solution written in the general form

16.17.4 DIRECT USE

procedure delire($x, k, grille, lcoeff, param$);

This procedure computes formal solutions of a linear homogeneous differential equation with polynomial coefficients over \mathbb{Q} and of any order, in the neighborhood of zero, regular or irregular singular point. In fact it initializes the call of the NEWTON procedure that is a recursive procedure (algorithm of NEWTON-RAMIS-MALGRANGE)

x : variable
 k : "number of desired terms".
 For each formal series in xt appearing in $polysol$,
 $a_0 + a_1xt + a_2xt^2 + \dots + a_nxt^n + \dots$, we compute the $k + 1$ first coefficients a_0, a_1, \dots, a_k .
 $grille$: the coefficients of the differential operator are polynomial in x^{grille} (in general $grille = 1$)
 $lcoeff$: list of coefficients of the differential operator (in increasing order of differentiation)
 $param$: list of parameters

This procedure RETURNS the list of general solutions.

16.17.5 USEFUL FUNCTIONS

Reading of equation coefficients

procedure lectabcoef();

This procedure is called by DESIR to read the coefficients of an equation, in *increasing order of differentiation*, but can be used independently.

reading of n : order of the equation.

reading of parameters (only if a variable other than x appears in the coefficients)

this procedure returns the list $\{ lcoeff, param \}$ made up of the list of coefficients and the list of parameters (which can be empty).

Verification of results

procedure solvalide($solutions, solk, k$);

This procedure enables the validity of the solution number *solk* in the list "solutions" to be verified.

solutions = {*lcoeff*, {..., {*general_solution*}, ...}} is any element of the list returned by DESIR or is {*lcoeff*, *sol*} where *sol* is the list returned by DELIRE.

If we carry over the solution $e^{qx} x^{r*ram} polysol x$ in the equation, the result has the form $e^{qx} x^{r*ram} reste$, where *reste* is a polynomial in $\log(xt)$, with polynomial coefficients in xt . This procedure computes the minimal valuation *V* of *reste* as polynomial in xt , using *k* "number of desired terms" asked for at the call of DESIR or DELIRE, and DISPLAYS the "theoretical" size order of the regular part of the result : $x^{ram*(r+v)}$.

On the other hand, this procedure carries over the solution in the equation and DISPLAYS the significative term of the result. This is of the form :

$$e^{qx} x^a polynomial(\log(xt)), \quad \text{with } a \geq ram * (r + v).$$

Finally this procedure RETURNS the complete result of the carry over of the solution in the equation.

This procedure cannot be used if the solution number *solk* is linked to a condition.

Writing of different forms of results

procedure standsol(solutions);

This procedure enables the simplified form of each solution to be obtained from the list "solutions", {*lcoeff*, {..., {*general_solution*}, ...}} which is one of the elements of the list returned by DESIR, or {*lcoeff*, *sol*} where *sol* is the list returned by DELIRE.

This procedure RETURNS a list of 3 elements : { *lcoeff*, *solstand*, *solcond* }

lcoeff = list of differential equation coefficients
solstand = list of solutions written in standard form
solcond = list of conditional solutions that have not been written in standard form. These solutions remain in general form.

This procedure has no meaning for "conditional" solutions. In case, a value has to be given to the parameters, that can be done either by calling the procedure SORPARAM that displays and returns these solutions in the standard form, either by calling the procedure SOLPARAM which returns these solutions in general form.

procedure sorsol(sol);

This procedure is called by DESIR to write the solution *sol*, given in general form, in standard form with enumeration of different conditions (if there are any). It can be used independently.

Writing of solutions after the choice of parameters

procedure sorparam(*solutions, param*);

This is an interactive procedure which displays the solutions evaluated : the value of parameters is requested.

solutions : {*lcoeff*, {..., {*general_solution*}, ...}}
param : list of parameters.

It returns the list formed of 2 elements :

- list of evaluated coefficients of the equation
- list of standard solutions evaluated for the value of parameters.

procedure solparam(*solutions, param, valparam*);

This procedure evaluates the general solutions for the value of parameters given by *valparam* and returns these solutions in general form.

solutions : {*lcoeff*, {..., {*general_solution*}, ...}}
param : list of parameters
valparam : list of parameters values

It returns the list formed of 2 elements :

- list of evaluated coefficients of the equation
- list of solutions in general form, evaluated for the value of parameters.

Transformations

procedure changehom(*lcoeff, x, secmember, id*);

Differentiation of an equation with right-hand side.

lcoeff : list of coefficients of the equation
x : variable
secmember : right-hand side
id : order of the differentiation.

It returns the list of coefficients of the differentiated equation. It enables an equation with polynomial right-hand side to be transformed into a homogeneous equation by differentiating id times, $id = \text{degre}(\text{secmember}) + 1$.

procedure changevar(*lcoeff*, *x*, *v*, *fact*);

Changing of variable in the homogeneous equation defined by the list *lcoeff* of its coefficients : the old variable x and the new variable v are linked by the relation $x = \text{fact}(v)$.

It returns the list of coefficients in respect to the variable v of the new equation.

examples of use :

- translation enabling a rational singularity to be brought back to zero.
- $x = 1/v$ brings the infinity to 0.

procedure changefonc(*lcoeff*, *x*, *q*, *fact*);

Changing of unknown function in the homogeneous equation defined by the list *lcoeff* of its coefficients :

lcoeff : list of coefficients of the initial equation
x : variable
q : new unknown function
fact : y being the unknown function $y = \text{fact}(q)$

It returns the list of coefficients of the new equation.

Example of use :

this procedure enables the computation, in the neighbourhood of an irregular singularity, of the "reduced" equation associated to one of the slopes (the Newton polygon having a null slope of no null length). This equation gives much informations on the associated divergent series.

Optional writing of intermediary results

switch trdesir : when it is ON, at each step of the Newton algorithm, a description of the Newton polygon is displayed (it is possible to follow the break of slopes), and at each call of the FROBENIUS procedure (case of a null slope) the corresponding indicial equation is displayed.

By default, this switch is OFF.

16.17.6 LIMITATIONS

1. This DESIR version is limited to differential equations leading to indicial equations of degree ≤ 3 . To pass beyond this limit, a further version written in the D5 environment of the computation with algebraic numbers has to be used.
2. The computation of a basis of solutions for an equation depending on parameters is assured only when the indicial equations are of degree ≤ 2 .

16.18 DFPART: Derivatives of generic functions

This package supports computations with total and partial derivatives of formal function objects. Such computations can be useful in the context of differential equations or power series expansions.

Author: Herbert Melenk.

The package DFPART supports computations with total and partial derivatives of formal function objects. Such computations can be useful in the context of differential equations or power series expansions.

16.18.1 Generic Functions

A generic function is a symbol which represents a mathematical function. The minimal information about a generic function is the number of its arguments. In order to facilitate the programming and for a better readable output this package assumes that the arguments of a generic function have default names such as $f(x, y), q(\rho, \phi)$. A generic function is declared by prototype form in a statement

```
GENERIC_FUNCTION fname(arg1, arg2 ··· argn);
```

where $fname$ is the (new) name of a function and arg_i are symbols for its formal arguments. In the following $fname$ is referred to as “generic function”, $arg_1, arg_2 \cdots arg_n$ as “generic arguments” and $fname(arg_1, arg_2 \cdots arg_n)$ as “generic form”. Examples:

```
generic_function f(x, y);
generic_function g(z);
```

After this declaration REDUCE knows that

- there are formal partial derivatives $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial g}{\partial z}$ and higher ones, while partial derivatives of f and g with respect to other variables are assumed as zero,
- expressions of the type $f(), g()$ are abbreviations for $f(x, y), g(z)$,
- expressions of the type $f(u, v)$ are abbreviations for $sub(x = u, y = v, f(x, y))$
- a total derivative $\frac{df(u, v)}{dw}$ has to be computed as $\frac{\partial f}{\partial x} \frac{du}{dw} + \frac{\partial f}{\partial y} \frac{dv}{dw}$

16.18.2 Partial Derivatives

The operator `DFP` represents a partial derivative:

`DFP(expr, dfarg1, dfarg2 · · · dfargn);`

where *expr* is a function expression and *dfarg_i* are the differentiation variables. Examples:

`dfp (f () , { x , y }) ;`

means $\frac{\partial^2 f}{\partial x \partial y}$ and

`dfp (f (u , v) , { x , y }) ;`

stands for $\frac{\partial^2 f}{\partial x \partial y}(u, v)$. For compatibility with the *DF* operator the differentiation variables need not be entered in list form; instead the syntax of *DF* can be used, where the function expression is followed by the differentiation variables, eventually with repetition numbers. Such forms are interenally converted to the above form with a list as second parameter.

The expression *expr* can be a generic function with or without arguments, or an arithmetic expression built from generic functions and other algebraic parts. In the second case the standard differentiation rules are applied in order to reduce each derivative expressions to a minimal form.

When the switch *NAT* is on partial derivatives of generic functions are printed in standard index notation, that is f_{xy} for $\frac{\partial^2 f}{\partial x \partial y}$ and $f_{xy}(u, v)$ for $\frac{\partial^2 f}{\partial x \partial y}(u, v)$. Therefore single characters should be used for the arguments whenever possible. Examples:

```
generic_function f(x,y);
generic_function g(y);
dfp(f(),x,2);
```

```
F
  XX
```

```
dfp(f()*g(),x,2);
```

```
F   *G()
  XX
```

```
dfp (f () *g () , x, y) ;
```

$$F_{XY} * G() + F_X * G_Y$$

The difference between partial and total derivatives is illustrated by the following example:

```
generic_function h(x) ;
dfp (f(x, h(x)) *g(h(x)) , x) ;
```

$$F_X(X, H(X)) * G(H(X))$$

```
df (f(x, h(x)) *g(h(x)) , x) ;
```

$$F_X(X, H(X)) * G(H(X)) + F_Y(X, H(X)) * H_X(X) * G(H(X))$$

$$+ G_Y(H(X)) * H_X(X) * F(X, H(X))$$

Cooperation of partial derivatives and Taylor series under a differential side relation

$\frac{dq}{dx} = f(x, q)$:

```
load_package taylor;
operator q;
let df(q(~x), x) => f(x, q(x));
taylor(q(x0+h), h, 0, 3);
```

$$Q(X0) + F_X(X0, Q(X0)) * H + \frac{F_X(X0, Q(X0)) + F_Y(X0, Q(X0)) * F(X0, Q(X0))}{2} * H^2$$

$$+ (F_{XX}(X0, Q(X0)) + F_{XY}(X0, Q(X0)) * F(X0, Q(X0)))$$

$$+ \frac{F_X(X0, Q(X0)) * F_Y(X0, Q(X0)) + F_{YX}(X0, Q(X0)) * F(X0, Q(X0))}{2}$$

2

2

3

$$\begin{aligned}
& + \frac{F_{XX}(X_0, Q(X_0)) * F(X_0, Q(X_0))}{YY} + \frac{F_{XY}(X_0, Q(X_0)) * F(X_0, Q(X_0))}{Y} / 6 * H \\
& + O(H^4)
\end{aligned}$$

Normally partial differentials are assumed as non-commutative

$$\text{dfp}(f(), x, y) - \text{dfp}(f(), y, x);$$

$$\frac{F}{XY} - \frac{F}{YX}$$

However, a generic function can be declared to have globally interchangeable partial derivatives using the declaration `DFP_COMMUTE` which takes the name of a generic function or a generic function form as argument. For such a function differentiation variables are rearranged corresponding to the sequence of the generic variables.

```

generic_function q(x,y);
dfp_commute q(x,y);
dfp(q(), {x,y,y}) + dfp(q(), {y,x,y}) + dfp(q(), {y,y,x});

3*Q
  XYY

```

If only a part of the derivatives commute, this has to be declared using the standard `REDUCE` rule mechanism. Please note that then the derivative variables must be written as list.

16.18.3 Substitutions

When a generic form or a `DFP` expression takes part in a substitution the following steps are performed:

1. The substitutions are performed for the arguments. If the argument list is empty the substitution is applied to the generic arguments of the function; if these change, the resulting forms are used as new actual arguments. If the generic function itself is not affected by the substitution, the process stops here.

2. If the function name or the generic function form occurs as a left hand side in the substitution list, it is replaced by the corresponding right hand side.
3. The new form is partially differentiated according to the list of partial derivative variables.
4. The (eventually modified) actual parameters are substituted into the form for their corresponding generic variables. This substitution is done by name.

Examples:

```
generic_function f(x,y);
sub(y=10,f());

F(X,10)

sub(y=10,dfp(f(),x,2));

F  (X,10)
XX

sub(y=10,dfp(f(y,y),x,2));

F  (10,10)
XX

sub(f=x**3*y**3,dfp(f(),x,2));

      3
6*X*Y

generic_function ff(y,z);
sub(f=ff,f(a,b));

FF(B,Z)
```

The dataset `dfpart.tst` contains more examples, including a complete application for computing the coefficient equations for Runge-Kutta ODE solvers.

16.19 DUMMY: Canonical form of expressions with dummy variables

This package allows a user to find the canonical form of expressions involving dummy variables. In that way, the simplification of polynomial expressions can be fully done. The indeterminates are general operator objects endowed with as few properties as possible. In that way the package may be used in a large spectrum of applications.

Author: Alain Dresse.

16.19.1 Introduction

The possibility to handle dummy variables and to manipulate dummy summations are important features in many applications. In particular, in theoretical physics, the possibility to represent complicated expressions concisely and to realize simplifications efficiently depend on both capabilities. However, when dummy variables are used, there are many more ways to express a given mathematical objects since the names of dummy variables may be chosen almost arbitrarily. Therefore, from the point of view of computer algebra the simplification problem is much more difficult. Given a definite ordering, one is, at least, to find a representation which is independent of the names chosen for the dummy variables otherwise, simplifications are impossible. The package does handle any number of dummy variables and summations present in expressions which are arbitrary multivariate polynomials and which have operator objects eventually dependent on one (or several) dummy variable(s) as some of their indeterminates. These operators have the same generality as the one existing in REDUCE. They can be noncommutative, anticommutative or commutative. They can have any kind of symmetry property. Such polynomials will be called in the following *dummy* polynomials. Any monomial of this kind will be called *dummy* monomial. For any such object, the package allows to find a well defined *normal form* in one-to-one correspondance with it.

In section 2, the convention for writing dummy summations is explained and the available declarations to introduce or suppress dummy variables are given.

In section 3, the commands allowing to give various algebraic properties to the operators are described.

In section 4, the use of the function CANONICAL is explained and illustrated.

In section 5, a fairly complete set of references is given.

The use of DUMMY requires that the package ASSIST version 2.2 be available. This is the case when REDUCE 3.6 is used. When loaded, ASSIST is automatically loaded.

16.19.2 Dummy variables and dummy summations

A dummy variable (let us name it dv) is an identifier which runs from the integer i_1 to another integer i_2 . To the extent that no definite space is defined, i_1 and i_2 are assumed to be some integers which are the *same* for all dummy variables.

If f is any REDUCE operator, then the simplest dummy summation associated to dv is the sum

$$\sum_{dv=i_1}^{i_2} f(dv)$$

and is simply written as

$$f(dv).$$

No other rules govern the implicit summations. dv can appear as many times we want since the operator f may depend on an arbitrary number of variables. So, the package is potentially applicable to many contexts. For instance, it is possible to add rules of the kind one encounters in tensor calculus.

Obviously, there are as many ways we want to express the *same* quantity. If the name of another dummy variable is dum then the previous expression is written as

$$\sum_{dum=i_1}^{i_2} f(dum)$$

and the computer algebra system should be able to find that the expression

$$f(dv) - f(dum);$$

is equal to 0. A very special case which is *allowed* is when f is the identity operator. So, a generic dummy polynomial will be a sum of dummy monomials of the kind

$$\prod_i c_i * f_i(dv_1, \dots, dv_{k_i}, fr_1, \dots, fr_{l_i})$$

where dv_1, \dots , are dummy variables while fr_1, \dots , are ordinary or free variables.

To declare dummy variables, two commands are available:

- i.

```
dummy_base <idp>;
```

where `idp` is the name of any unassigned identifier.

- ii.

```
dummy_names <d>, <dp>, <dpp> . . . .;
```

The first one declares idp_1, \dots, idp_n as dummy variables i.e. all variables of the form idp_{xxx} where xxx is a number will be dummy variables, such as $idp_1, idp_2, \dots, idp_{23}$. The second one gives special names for dummy variables. All other identifiers which may appear are assumed to be *free*. However, there is a restriction: named and base dummy variables cannot be declared *simultaneously*. The above declarations are mutually *exclusive*. Here is an example showing that:

```
dummy_base dv; ==> dv

% dummy indices are dv1, dv2, dv3, ...

dummy_names i, j, k; ==>

***** The created dummy base dv must be cleared
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(i)*op(dv2)$
```

means a sum over dv_1, dv_2 . To clear the dummy base, and to create the dummy names i, j, k one is to do

```
clear_dummy_base; ==> t

dummy_names i, j, k; ==> t

% dummy indices are i, j, k.
```

When this is done, an expression like

```
op(dv1)*sin(dv2)*abs(x)*op(i)^3*op(dv2)$
```

means a sum over i . One should keep in mind that every application of the above commands erases the previous ones. It is also possible to display the declared dummy names using `SHOW_DUMMY_NAMES`:

```
show_dummy_names(); ==> {i, j, k}
```

To suppress *all* dummy variables one can enter

```
clear_dummy_names; clear_dummy_base;
```

16.19.3 The Operators and their Properties

All dummy variables *should appear at first level* as arguments of operators. For instance, if i and j are dummy variables, the expression

```
rr:= op(i,j)-op(j,j)
```

is allowed but the expression

```
op(i,op(j)) - op(j,op(j))
```

is *not* allowed. This is because dummy variables are not detected if they appear at a level larger than 1. Apart from that there is no restrictions. Operators may be commutative, noncommutative or even anticommutative. Therefore they may be elements of an algebra, they may be tensors, spinors, grassman variables, etc. ... By default they are assumed to be *commutative* and without symmetry properties. The REDUCE command NONCOM is taken into account and, in addition, the command

```
anticom at1, at2;
```

makes the operators at_1 and at_2 anticommutative.

One can also give symmetry properties to them. The usual declarations SYMMETRIC and ANTISYMMETRIC are taken into account. Moreover and most important they can be endowed with a *partial* symmetry through the command SYMTREE. Here are three illustrative examples for the r operator:

```
symtree (r,{!+, 1, 2, 3, 4});
symtree (r,{!*, 1, {!-, 2, 3, 4}});
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});
```

The first one makes the operator (fully) symmetric. The second one declares it antisymmetric with respect to the three last indices. The symbols !*, !+ and !- at the beginning of each list mean that the operator has no symmetry, is symmetric or is antisymmetric with respect to the indices inside the list. Notice that the indices are not denoted by their names but merely by their natural order of appearance. 1 means the first written argument of r , 2 its second argument etc. The first command is equivalent to the declaration `symmetric` except that the number of indices of r is *restricted* to 4 i.e. to the number declared in SYMTREE. In the second example r is stated to have no symmetry with respect to the first index and is declared to be antisymmetric with respect to the three last indices. In the third example, r is made symmetric with respect to the interchange of the pairs of indices 1,2 and 3,4 respectively and is made antisymmetric separately within the pairs (1, 2) and (3, 4). It is the symmetry of the Riemann tensor. The anticommutation property and the

various symmetry properties may be suppressed by the commands `REMANTICOM` and `REMSYM`. To eliminate partial symmetry properties one can also use `SYMTREE` itself. For example, assuming that r has the Riemann symmetry, to eliminate it do

```
symtree (r,{!*, 1, 2, 3, 4});
```

However, notice that the number of indices remains fixed and equal to 4 while with `REMSYM` it becomes again arbitrary.

16.19.4 The Function `CANONICAL`

`CANONICAL` is the most important functionality of the package. It can be applied on any polynomial whether it is a dummy polynomial or not. It returns a normal form uniquely determined from the current ordering of the system. If the polynomial does not contain any dummy index, it is rewritten taking into account the various operator properties or symmetries described above. For instance,

```
symtree (r, {!+, {!-, 1, 2}, {!-, 3, 4}});

aa:=r(x3,x4,x2,x1)$

canonical aa; ==> - r(x1,x2,x3,x4).
```

If it contains dummy indices, `CANONICAL` takes also into account the various dummy summations, makes the relevant simplifications, eventually rename the dummy indices and returns the resulting normal form. Here is a simple example:

```
operator at1,at2;
anticom at1,at2;

dummy_names i,j,k; ==> t

show_dummy_names(); ==> {i,j,k}

rr:=at1(i)*at2(k) -at2(k)*at1(i)$

canonical rr; ==> 2*at1(i)*at2(j)
```

It is important to notice, in the above example, that in addition to the summations over indices i and k , and of the anticommutativity property of the operators, `canonical` has replaced the index k by the index j . This substitution is essential to get full simplification. Several other examples are given in the test file and,

there, the output of `CANONICAL` is explained.

As stated in the previous section, the dependence of operators on dummy indices is limited to *first* level. An erroneous result will be generated if it is not the case as the subsequent example illustrates:

```
operator op;

dummy_names i, j;

rr:=op(i, op(j))-op(j, op(j))$

canonical rr; ==> 0
```

Zero is obtained because, in the second term, `CANONICAL` has replaced j by i but has left $op(j)$ unchanged because it *does not see* the index j which is inside. This fact has also the consequence that it is unable to simplify correctly (or at all) expressions which contain some derivatives. For instance (i and j are dummy indices):

```
aa:=df(op(x, i), x) -df(op(x, j), x)$

canonical aa; ==> df(op(x, i), x) - df(op(x, j), x)
```

instead of zero. A second limitation is that `CANONICAL` does not add anything to the problem of simplifications when side relations (like Bianchi identities) are present.

16.19.5 Bibliography

- **Butler, G. and Lam, C. W. H.**, “A general backtrack algorithm for the isomorphism problem of combinatorial objects”, J. Symb. Comput. vol.1, (1985) p.363-381.
- **Butler, G. and Cannon, J. J.**, “Computing in Permutation and Matrix Groups I: Normal Closure, Commutator Subgroups, Series”, Math. Comp. vol.39, number 60, (1982), p. 663-670.
- **Butler, G.**, “Computing in Permutation and Matrix Groups II: Backtrack Algorithm”, Math. Comp. vol.39, number 160, (1982), p.671-680.
- **Leon, J.S.**, “On an Algorithm for Finding a Base and a Strong Generating Set for a Group Given by Generating Permutations”, Math. Comp. vol.35, (1980), p.941-974.
- **Leon, J. S.**, “Computing Automorphism Groups of Combinatorial Objects”, Proc. LMS Symp. on Computational Group Theory, Durham, England, editor: Atkinson, M. D., Academic Press, London, (1984).

- **Leon, J. S.**, “Permutation Group Algorithms Based on Partitions, I: Theory and Algorithms”, *J.Symb. Comput.* vol.12, (1991) p. 533-583.
- **Linton, Stephen Q.**, “Double Coset Enumeration”, *J. Symb. Comput.*, vol.12, (1991) p. 415-426.
- **McKay, B. D.**, “Computing Automorphism Groups and Canonical Labellings of Graphs”, *Proc. Internat. Conf. on Combinatorial Theory, Lecture Notes in Mathematics*“ vol. 686, (1977), p.223-232, Springer-Verlag, Berlin.
- **Rodionov, A. Ya. and Taranov, A. Yu.**, “Combinatorial Aspects of Simplification of Algebraic Expression”, *Proceedings of Eurocal 87, Lecture Notes in Comp. Sci.*, vol. 378, (1989), p. 192.
- **Sims, C. C.**, “Determining the Conjugacy Classes of a Permutation Group”, *Computers in Algebra and Number Theory, SIAM-AMS Proceedings*, vol. 4, (1971), p. 191-195, editor G. Birkhoff and M. Hall Jr., Amer. Math. Soc..
- **Sims, C. C.**, “Computation with Permutation Groups”, *Proc. of the Second Symposium on Symbolic and Algebraic Manipulation*, (1971), p. 23-28, editor S. R. Petrick, Assoc. Comput. Mach., New York.
- **Burnel A., Caprasse H., Dresse A.**, “ Computing the BRST operator used in Quantization of Gauge Theories” *IJMPC* vol. 3, (1993) p.321-35.
- **Caprasse H.**, “BRST charge and Poisson Algebras”, *Discrete Mathematics and Theoretical Computer Science, Special Issue: Lie Computations papers*, <http://dmtcs.thomsonscience.com>, (1997).

16.20 EXCALC: A differential geometry package

EXCALC is designed for easy use by all who are familiar with the calculus of Modern Differential Geometry. The program is currently able to handle scalar-valued exterior forms, vectors and operations between them, as well as non-scalar valued forms (indexed forms). It is thus an ideal tool for studying differential equations, doing calculations in general relativity and field theories, or doing simple things such as calculating the Laplacian of a tensor field for an arbitrary given frame.

Author: Eberhard Schrüfer.

Acknowledgments

This program was developed over several years. I would like to express my deep gratitude to Dr. Anthony Hearn for his continuous interest in this work, and especially for his hospitality and support during a visit in 1984/85 at the RAND Corporation, where substantial progress on this package could be achieved. The Heinrich Hertz-Stiftung supported this visit. Many thanks are also due to Drs. F.W. Hehl, University of Cologne, and J.D. McCrea, University College Dublin, for their suggestions and work on testing this program.

16.20.1 Introduction

EXCALC is designed for easy use by all who are familiar with the calculus of Modern Differential Geometry. Its syntax is kept as close as possible to standard textbook notations. Therefore, no great experience in writing computer algebra programs is required. It is almost possible to input to the computer the same as what would have been written down for a hand-calculation. For example, the statement

$$f * x^y + u \lrcorner (y^z x)$$

would be recognized by the program as a formula involving exterior products and an inner product. The program is currently able to handle scalar-valued exterior forms, vectors and operations between them, as well as non-scalar valued forms (indexed forms). With this, it should be an ideal tool for studying differential equations, doing calculations in general relativity and field theories, or doing such simple things as calculating the Laplacian of a tensor field for an arbitrary given frame. With the increasing popularity of this calculus, this program should have an application in almost any field of physics and mathematics.

Since the program is completely embedded in REDUCE, all features and facilities of REDUCE are available in a calculation. Even for those who are not quite comfortable in this calculus, there is a good chance of learning it by just playing with

the program.

This is the last release of version 2. A much extended differential geometry package (which includes complete symbolic index simplification, tensors, mappings, bundles and others) is under development.

Complaints and comments are appreciated and should be send to the author. If the use of this program leads to a publication, this document should be cited, and a copy of the article to the above address would be welcome.

16.20.2 Declarations

Geometrical objects like exterior forms or vectors are introduced to the system by declaration commands. The declarations can appear anywhere in a program, but must, of course, be made prior to the use of the object. Everything that has no declaration is treated as a constant; therefore zero-forms must also be declared.

An exterior form is introduced by

PFORM $\langle \text{declaration}_1 \rangle, \langle \text{declaration}_2 \rangle, \dots;$

where

$\langle \text{declaration} \rangle ::= \langle \text{name} \rangle \mid \langle \text{list of names} \rangle = \langle \text{number} \rangle \mid \langle \text{identifier} \rangle \mid$
 $\langle \text{expression} \rangle$
 $\langle \text{name} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{identifier} \rangle (\langle \text{arguments} \rangle)$

For example

```
pform u=k,v=4,f=0,w=dim-1;
```

declares U to be an exterior form of degree K, V to be a form of degree 4, F to be a form of degree 0 (a function), and W to be a form of degree DIM-1.

If the exterior form should have indices, the declaration would be

```
pform curv(a,b)=2, chris(a,b)=1;
```

The names of the indices are arbitrary.

Exterior forms of the same degree can be grouped into lists to save typing.

```
pform {x,y,z}=0, {rho(k,l),u,v(k)}=1;
```

The declaration of vectors is similar. The command TVECTOR takes a list of names.

TVECTOR $\langle name_1 \rangle, \langle name_2 \rangle, \dots;$

For example, to declare X as a vector and COMM as a vector with two indices, one would say

```
tvector x, comm(a,b);
```

If a declaration of an already existing name is made, the old declaration is removed, and the new one is taken.

The exterior degree of a symbol or a general expression can be obtained with the function

EXDEGREE $\langle expression \rangle;$

Example:

```
exdegree(u + 3*chris(k,-k));
```

```
1
```

16.20.3 Exterior Multiplication

Exterior multiplication between exterior forms is carried out with the nary infix operator \wedge (wedge). Factors are ordered according to the usual ordering in REDUCE using the commutation rule for exterior products.

Example 10

```
pform u=1, v=1, w=k;
```

```
u^v;
```

```
U^V
```

```
v^u;
```

```
- U^V
```

```
u^u;
```

```
0
```

```
w^u^v;
```

```
K
```

```
( - 1) *U^V^W

(3*u-a*w)^(w+5*v)^u;

A*(5*U^V^W - U^W^W)
```

It is possible to declare the dimension of the underlying space by

SPACEDIM *< number >* | *< identifier >*;

If an exterior product has a degree higher than the dimension of the space, it is replaced by 0:

```
spacedim 4;

pform u=2,v=3;

u^v;

0
```

16.20.4 Partial Differentiation

Partial differentiation is denoted by the operator @. Its capability is the same as the REDUCE DF operator.

Example 11

```
@(sin x,x);

COS(X)

@(f,x);

0
```

An identifier can be declared to be a function of certain variables. This is done with the command FDOMAIN. The following would tell the partial differentiation operator that F is a function of the variables X and Y and that H is a function of X.

```
fdomain f=f(x,y),h=h(x);
```

Applying @ to F and H would result in

```
@(x*f, x);
```

```
F + X*@ F
      X
```

```
@(h, y);
```

```
0
```

The partial derivative symbol can also be an operator with a single argument. It then represents a natural base element of a tangent vector.

Example 12

```
a*@ x + b*@ y;
```

```
A*@ + B*@
  X   Y
```

16.20.5 Exterior Differentiation

Exterior differentiation of exterior forms is carried out by the operator `d`. Products are normally differentiated out, *i.e.*

```
pform x=0, y=k, z=m;
```

```
d(x * y);
```

```
X*d Y + d X^Y
```

```
d(r*y);
```

```
R*d Y
```

```
d(x*y^z);
```

```

      K
( - 1) *X*Y^d Z + X*d Y^Z + d X^Y^Z
```

This expansion can be suppressed by the command `NOXPND D`.

```
noxpnd d;
```

$$d(y^z);$$

$$d(Y^Z)$$

To obtain a canonical form for an exterior product when the expansion is switched off, the operator D is shifted to the right if it appears in the leftmost place.

$$d\ y\ ^\ z;$$

$$- \binom{K}{-1} * Y^d Z + d(Y^Z)$$

Expansion is performed again when the command `XPND D` is executed.

Functions which are implicitly defined by the `FDOMAIN` command are expanded into partial derivatives:

$$\text{pform } x=0, y=0, z=0, f=0;$$

$$\text{fdomain } f=f(x, y);$$

$$d\ f;$$

$$\frac{\partial}{\partial X} F * d X + \frac{\partial}{\partial Y} F * d Y$$

If an argument of an implicitly defined function has further dependencies the chain rule will be applied *e.g.*

$$\text{fdomain } y=y(z);$$

$$d\ f;$$

$$\frac{\partial}{\partial X} F * d X + \frac{\partial}{\partial Y} F * \frac{\partial}{\partial Z} Y * d Z$$

Expansion into partial derivatives can be inhibited by `NOXPND @` and enabled again by `XPND @`.

The operator is of course aware of the rules that a repeated application always leads to zero and that there is no exterior form of higher degree than the dimension of the space.

$$d\ d\ x;$$

```

0

pform u=k;
spacedim k;

d u;

0

```

16.20.6 Inner Product

The inner product between a vector and an exterior form is represented by the diphthong $_ |$ (underscore or-bar), which is the notation of many textbooks. If the exterior form is an exterior product, the inner product is carried through any factor.

Example 13

```

pform x=0, y=k, z=m;

tvector u, v;

u \_ | (x*y^z);

K
X*(( - 1) *Y^U \_ | Z + U \_ | Y^Z)

```

In repeated applications of the inner product to the same exterior form the vector arguments are ordered *e.g.*

```

(u+x*v) \_ | (u \_ | (3*z));

- 3*U \_ | V \_ | Z

```

The duality of natural base elements is also known by the system, *i.e.*

```

pform {x,y}=0;

(a*@ x+b*@ (y)) \_ | (3*d x-d y);

3*A - B

```

16.20.7 Lie Derivative

The Lie derivative can be taken between a vector and an exterior form or between two vectors. It is represented by the infix operator $|_{_}$. In the case of Lie differentiating, an exterior form by a vector, the Lie derivative is expressed through inner products and exterior differentiations, *i.e.*

```
pform z=k;

tvector u;

u |_{\_} z;

U |_{\_} d Z + d(U |_{\_} Z)
```

If the arguments of the Lie derivative are vectors, the vectors are ordered using the anticommutivity property, and functions (zero forms) are differentiated out.

Example 14

```
tvector u,v;

v |_{\_} u;

- U |_{\_} V

pform x=0,y=0;

(x*u) |_{\_} (y*v);

- U*Y*V |_{\_} d X + V*X*U |_{\_} d Y + X*Y*U |_{\_} V
```

16.20.8 Hodge-* Duality Operator

The Hodge-* duality operator maps an exterior form of degree K to an exterior form of degree $N-K$, where N is the dimension of the space. The double application of the operator must lead back to the original exterior form up to a factor. The following example shows how the factor is chosen here

```
spacedim n;
pform x=k;

# # x;
```

$$(K^2 + K*N) * X * SGN$$

The indeterminate SGN in the above example denotes the sign of the determinant of the metric. It can be assigned a value or will be automatically set if more of the metric structure is specified (via COFRAME), *i.e.* it is then set to $g/|g|$, where g is the determinant of the metric. If the Hodge-* operator appears in an exterior product of maximal degree as the leftmost factor, the Hodge-* is shifted to the right according to

$$\begin{aligned} & \text{pform } \{x, y\} = k; \\ & \# x \wedge y; \\ & \quad \quad \quad 2 \\ & \quad \quad (K \quad + K \star N) \\ & (\quad - 1) \quad \quad \quad \star X \wedge \# Y \end{aligned}$$

More simplifications are performed if a coframe is defined.

16.20.9 Variational Derivative

The function VARDF returns as its value the variation of a given Lagrangian n-form with respect to a specified exterior form (a field of the Lagrangian). In the shared variable BNDEQ!*, the expression is stored that has to yield zero if integrated over the boundary.

Syntax:

VARDF(*< Lagrangian n-form >*, *< exterior form >*)

Example 15

```
spacedim 4;

pform l=4, a=1, j=3;

l:=-1/2*d a ^ # d a - a^# j$ %Lagrangian of the e.m. field

vardf(l,a);

- (# J + d # d A) %Maxwell's equations

bndeq!*;

- 'A^# d A %Equation at the boundary
```

Restrictions:

In the current implementation, the Lagrangian must be built up by the fields and the operations d , $\#$, and $@$. Variation with respect to indexed quantities is currently not allowed.

For the calculation of the conserved currents induced by symmetry operators (vector fields), the function `NOETHER` is provided. It has the syntax:

NOETHER(\langle Lagrangian n -form \rangle , \langle field \rangle , \langle symmetry generator \rangle)

Example 16

```
pform l=4, a=1, f=2;

spacedim 4;

l:= -1/2*d a^#d a;      %Free Maxwell field;

tvector x(k);           %An unspecified generator;

noether(l, a, x(-k));

( - 2*d(X _|A)^# d A - (X _|d A)^# d A + d A^(X _|# d A))/2
      K                K                K
```

The above expression would be the canonical energy momentum 3-forms of the Maxwell field, if X is interpreted as a translation;

16.20.10 Handling of Indices

Exterior forms and vectors may have indices. On input, the indices are given as arguments of the object. A positive argument denotes a superscript and a negative argument a subscript. On output, the indexed quantity is displayed two dimensionally if `NAT` is on. Indices may be identifiers or numbers.

Example 17

```
pform om(k, l)=m, e(k)=1;

e(k)^e(-l);

      K
E ^E
      L
```

```
om(4,-2);
```

```
4
OM
2
```

In the current release, full simplification is performed only if an index range is specified. It is hoped that this restriction can be removed soon. If the index range (the values that the indices can obtain) is specified, the given expression is evaluated for all possible index values, and the summation convention is understood.

Example 18

```
indexrange t,r,ph,z;
```

```
pform e(k)=1,s(k,l)=2;
```

```
w := e(k)*e(-k);
```

```
W := E *E + E *E + E *E + E *E
      T      R      PH      Z
      T      R      PH      Z
```

```
s(k,l):=e(k)^e(l);
```

```
T T
S := 0
```

```
R T      T R
S := - E ^E
```

```
PH T      T PH
S := - E ^E
```

```
.
.
.
```

If the expression to be evaluated is not an assignment, the values of the expression are displayed as an assignment to an indexed variable with name NS. This is done only on output, *i.e.* no actual binding to the variable NS occurs.

```

e(k) ^ e(l) ;

      T  T
NS      :=  0

      R  T          T  R
NS      :=  - E  ^ E
.
.
.
```

It should be noted, however, that the index positions on the variable NS can sometimes not be uniquely determined by the system (because of possible reorderings in the expression). Generally it is advisable to use assignments to display complicated expressions.

A range can also be assigned to individual index-names. For example, the declaration

```

indexrange {k,l}={x,y,z}, {u,v,w}={1,2};
```

would assign to the index identifiers k,l the range values x,y,z and to the index identifiers u,v,w the range values 1,2. The use of an index identifier not listed in previous indexrange statements has the range of the union of all given index ranges.

With the above example of an indexrange statement, the following index evaluations would take place

```

pform w n=0;

w(k) * w(-k) ;

      X          Y          Z
W * W  + W * W  + W * W
      X          Y          Z

w(u) * w(-u) ;

      1          2
W * W  + W * W
      1          2

w(r) * w(-r) ;
```

$$\begin{array}{ccccccccc}
 & 1 & & 2 & & X & & Y & & Z \\
 W * W & + & W * W & + & W * W & + & W * W & + & W * W \\
 1 & & 2 & & X & & Y & & Z
 \end{array}$$

In certain cases, one would like to inhibit the summation over specified index names, or at all. For this the command

NOSUM < *indexname*₁ >, ...;

and the switch NOSUM are available. The command NOSUM has the effect that summation is not performed over those indices which had been listed. The command RENOSUM enables summation again. The switch NOSUM, if on, inhibits any summation.

It is possible to declare symmetry properties for an indexed quantity by the command INDEX_SYMMETRIES. A prototypical example is as follows

```

index_symmetries u(k,l,m,n): symmetric      in {k,l},{m,n}
                                antisymmetric in {{k,l},{m,n}},
                                g(k,l),h(k,l): symmetric;

```

It declares the object u symmetric in the first two and last two indices and antisymmetric with respect to commutation of the given index pairs. If an object is completely symmetric or antisymmetric, the indices need not to be given after the corresponding keyword as shown above for g and h.

If applicable, this command should be issued, since great savings in memory and execution time result. Only strict components are printed.

The commands symmetric and antisymmetric of earlier releases have no effect.

16.20.11 Metric Structures

A metric structure is defined in **EXCALC** by specifying a set of basis one-forms (the coframe) together with the metric.

Syntax:

```

COFRAME < identifier > < (index1) > =< expression1 >,
          < identifier > < (index2) > =< expression2 >,
          .
          .
          .
          < identifier > < (indexn) > =< expressionn >

```

WITH METRIC *< name >=< expression >;*

This statement automatically sets the dimension of the space and the index range. The clause **WITH METRIC** can be omitted if the metric is Euclidean and the shorthand **WITH SIGNATURE** *< diagonal elements >* can be used in the case of a pseudo-Euclidean metric. The splitting of a metric structure in its metric tensor coefficients and basis one-forms is completely arbitrary including the extremes of an orthonormal frame and a coordinate frame.

Example 19

```

coframe e r=d r, e(ph)=r*d ph
    with metric g=e(r)*e(r)+e(ph)*e(ph);      %Polar coframe

coframe e(r)=d r, e(ph)=r*d(ph);              %Same as before

coframe o(t)=d t, o x=d x
    with signature -1,1;                       %A Lorentz coframe

coframe b(xi)=d xi, b(eta)=d eta              %A lightcone coframe
    with metric w=-1/2*(b(xi)*b(eta)+b(eta)*b(xi));

coframe e r=d r, e ph=d ph                    %Polar coordinate
    with metric g=e r*e r+r**2*e ph*e ph;     %basis

```

Individual elements of the metric can be accessed just by calling them with the desired indices. The value of the determinant of the covariant metric is stored in the variable **DETM!***. The metric is not needed for lowering or raising of indices as the system performs this automatically, *i.e.* no matter in what index position values were assigned to an indexed quantity, the values can be retrieved for any index position just by writing the indexed quantity with the desired indices.

Example 20

```

coframe e t=d t, e x=d x, e y=d y
    with signature -1,1,1;

pform f(k,l)=0;

antisymmetric f;

f(-t,-x):=ex$ f(-x,-y):=b$ f(-t,-y):=0$

```

```

on nero;

f(k,-1):=f(k,-1);

X
F      := - EX
  T

T
F      := - EX
  X

Y
F      := - B
  X

X
F      := B
  Y

```

Any expression containing differentials of the coordinate functions will be transformed into an expression of the basis one-forms. The system also knows how to take the exterior derivative of the basis one-forms.

Example 21(Spherical coordinates)

```

coframe e(r)=d(r), e(th)=r*d(th), e(ph)=r*sin(th)*d(ph);

d r^d th;

R TH
(E ^E )/R

d(e(th));

R TH
(E ^E )/R

pform f=0;

fdomain f=f(r,th,ph);

factor e;

```

```

on rat;

d f;          %The "gradient" of F in spherical coordinates;

      R          TH          PH
E *@  F + (E *@  F)/R + (E *@  F)/(R*SIN(TH))
      R          TH          PH

```

The frame dual to the frame defined by the `COFRAME` command can be introduced by **FRAME** command.

FRAME < *identifier* >;

This command causes the dual property to be recognized, and the tangent vectors of the coordinate functions are replaced by the frame basis vectors.

Example 22

```

coframe b r=d r,b ph=r*d ph,e z=d z; %Cylindrical coframe;

frame x;

on nero;

x(-k) _| b(1);

      R
NS      := 1
      R

      PH
NS      := 1
      PH

      Z
NS      := 1
      Z

x(-k) _| x(-1);          %The commutator of the dual frame;

      NS      := X /R
      PH R      PH

```



```

NS      := ( - X ) / R %i.e. it is not a coordinate base;
R PH      PH

```

As a convenience, the frames can be displayed at any point in a program by the command `DISPLAYFRAME;`.

The Hodge-* duality operator returns the explicitly constructed dual element if applied to coframe base elements. The metric is properly taken into account.

The total antisymmetric Levi-Cevita tensor `EPS` is also available. The value of `EPS` with an even permutation of the indices in a covariant position is taken to be +1.

16.20.12 Riemannian Connections

The command `RIEMANNCONX` is provided for calculating the connection 1 forms. The values are stored on the name given to `RIEMANNCONX`. This command is far more efficient than calculating the connection from the differential of the basis one-forms and using inner products.

Example 23(Calculate the connection 1-form and curvature 2-form on $S(2)$)

```

coframe e th=r*d th,e ph=r*sin(th)*d ph;

riemannconx om;

om(k,-1); %Display the connection forms;

TH
NS      := 0
TH

PH      PH
NS      := (E *COS (TH)) / (SIN (TH) *R)
TH

TH      PH
NS      := ( - E *COS (TH)) / (SIN (TH) *R)
PH

PH
NS      := 0

```

```

      PH

pform curv(k,l)=2;

curv(k,-l):=d om(k,-l) + om(k,-m)^om(m-l);
           %The curvature forms

      TH
CURV      := 0
      TH

      PH      TH PH 2
CURV      := ( - E ^E )/R
      TH      %Of course it was a sphere with
               %radius R.

      TH      TH PH 2
CURV      := (E ^E )/R
      PH

      PH
CURV      := 0
      PH

```

16.20.13 Ordering and Structuring

The ordering of an exterior form or vector can be changed by the command `FORDER`. In an expression, the first identifier or kernel in the arguments of `FORDER` is ordered ahead of the second, and so on, and ordered ahead of all not appearing as arguments. This ordering is done on the internal level and not only on output. The execution of this statement can therefore have tremendous effects on computation time and memory requirements. `REMFORDER` brings back standard ordering for those elements that are listed as arguments.

An expression can be put in a more structured form by renaming a subexpression. This is done with the command `KEEP` which has the syntax

KEEP < name₁ >=< expression₁ >,< name₂ >=< expression₂ >,...

The effect is that rules are set up for simplifying < name > without introducing its definition in an expression. In an expression the system also tries by reordering to generate as many instances of < name > as possible.

Example 24

```

pform x=0,y=0,z=0,f=0,j=3;

keep j=d x^d y^d z;

j;

J

d j;

0

j^d x;

0

fdomain f=f(x);

d f^d y^d z;

@ F*J
X

```

The capabilities of `KEEP` are currently very limited. Only exterior products should occur as righthand sides in `KEEP`.

16.20.14 Summary of Operators and Commands

Table 16.1 summarizes EXCALC commands and the page number they are defined on.

\wedge	Exterior Multiplication	288
@	Partial Differentiation	289
@	Tangent Vector	290
#	Hodge-* Operator	293
-	Inner Product	292
-	Lie Derivative	293
COFRAME	Declaration of a coframe	299
d	Exterior differentiation	290
DISPLAYFRAME	Displays the frame	303
EPS	Levi-Civita tensor	303
EXDEGREE	Calculates the exterior degree of an expression	288
FDOMAIN	Declaration of implicit dependencies	289
FORDER	Ordering command	304
FRAME	Declares the frame dual to the coframe	302
INDEXRANGE	Declaration of indices	297
INDEX_SYMMETRIES	Declares arbitrary index symmetry properties	299
KEEP	Structuring command	304
METRIC	Clause of COFRAME to specify a metric	299
NOETHER	Calculates the Noether current	296
NOSUM	Inhibits summation convention	299
NOXPND d	Inhibits the use of product rule for d	290
NOXPND @	Inhibits expansion into partial derivatives	291
PFORM	Declaration of exterior forms	287
REMFORDER	Clears ordering	304
RENOSUM	Enables summation convention	299
RIEMANNCONX	Calculation of a Riemannian Connection	303
SIGNATURE	Clause of COFRAME to specify a pseudo-Euclidean metric	300
SPACEDIM	Command to set the dimension of a space	289
TVECTOR	Declaration of vectors	287
VARDF	Variational derivative	295
XPND d	Enables the use of product rule for d (default)	291
XPND @	Enables expansion into partial derivatives (default)	291

Table 16.1: EXCALC Command Summary

16.20.15 Examples

The following examples should illustrate the use of **EXCALC**. It is not intended to show the most efficient or most elegant way of stating the problems; rather the variety of syntactic constructs are exemplified. The examples are on a test file distributed with **EXCALC**.

```
% Problem: Calculate the PDE's for the isovector of the heat equation.
% -----
%           (c.f. B.K. Harrison, f.B. Estabrook, "Geometric Approach...",
%           J. Math. Phys. 12, 653, 1971)

% The heat equation  $\frac{\partial}{\partial x^2} \psi = \frac{\partial}{\partial t} \psi$  is equivalent to the set of exterior
%
% equations (with  $u = \frac{\partial}{\partial t} \psi$ ,  $y = \frac{\partial}{\partial x} \psi$ ):
%
pform {psi,u,x,y,t}=0,a=1,{da,b}=2;

a := d psi - u*d t - y*d x;

da := - d u^d t - d y^d x;

b := u*d x^d t - d y^d t;

% Now calculate the PDE's for the isovector.

tvector v;

pform {vpsi,vt,vu,vx,vy}=0;
fdomain vpsi=vpsi(psi,t,u,x,y), vt=vt(psi,t,u,x,y), vu=vu(psi,t,u,x,y),
vx=vx(psi,t,u,x,y), vy=vy(psi,t,u,x,y);

v := vpsi*@ psi + vt*@ t + vu*@ u + vx*@ x + vy*@ y;

factor d;
on rat;

il := v |_ a - l*a;

pform o=1;

o := ot*d t + ox*d x + ou*d u + oy*d y;
```

```

fdomain f=f(psi,t,u,x,y);

i11 := v _| d a - l*a + d f;

let vx=-@ (f,y),vt=-@ (f,u),vu=@ (f,t)+u*@ (f,psi),vy=@ (f,x)+y*@ (f,psi),
    vpsi=f-u*@ (f,u)-y*@ (f,y);

factor ^;

i2 := v |_ b - xi*b - o^a + zeta*da;

let ou=0,oy=@ (f,u,psi),ox=-u*@ (f,u,psi),
    ot=@ (f,x,psi)+u*@ (f,y,psi)+y*@ (f,psi,psi);

i2;

let zeta=-@ (f,u,x)-@ (f,u,y)*u-@ (f,u,psi)*y;

i2;

let xi=-@ (f,t,u)-u*@ (f,u,psi)+@ (f,x,y)+u*@ (f,y,y)+y*@ (f,y,psi)+@ (f,psi);

i2;

let @ (f,u,u)=0;

i2;      % These PDE's have to be solved.


clear a,da,b,v,i1,i11,o,i2,xi,t;
remfdomain f,vpsi,vt,vu,vx,vy;
clear @ (f,u,u);


% Problem:
% -----
% Calculate the integrability conditions for the system of PDE's:
% (c.f. B.F. Schutz, "Geometrical Methods of Mathematical Physics"
% Cambridge University Press, 1984, p. 156)


% @ z /@ x + a1*z  + b1*z  = c1
%   1             1      2

% @ z /@ y + a2*z  + b2*z  = c2
%   1             1      2

% @ z /@ x + f1*z  + g1*z  = h1
%   2             1      2

```

```

% @ z /@ y + f2*z  + g2*z  = h2
%      2          1          2          ;

pform w(k)=1,integ(k)=4,{z(k),x,y}=0,{a,b,c,f,g,h}=1,
      {a1,a2,b1,b2,c1,c2,f1,f2,g1,g2,h1,h2}=0;

fdomain a1=a1(x,y),a2=a2(x,y),b1=b1(x,y),b2=b2(x,y),
        c1=c1(x,y),c2=c2(x,y),f1=f1(x,y),f2=f2(x,y),
        g1=g1(x,y),g2=g2(x,y),h1=h1(x,y),h2=h2(x,y);

a:=a1*d x+a2*d y$
b:=b1*d x+b2*d y$
c:=c1*d x+c2*d y$
f:=f1*d x+f2*d y$
g:=g1*d x+g2*d y$
h:=h1*d x+h2*d y$

% The equivalent exterior system:
factor d;
w(1) := d z(-1) + z(-1)*a + z(-2)*b - c;
w(2) := d z(-2) + z(-1)*f + z(-2)*g - h;
indexrange 1,2;
factor z;
% The integrability conditions:

integ(k) := d w(k) ^ w(1) ^ w(2);

clear a,b,c,f,g,h,x,y,w(k),integ(k),z(k);
remfdomain a1,a2,b1,c1,c2,f1,f2,g1,g2,h1,h2;

% Problem:
% -----
% Calculate the PDE's for the generators of the d-theta symmetries of
% the Lagrangian system of the planar Kepler problem.
% c.f. W.Sarlet, F.Cantrijn, Siam Review 23, 467, 1981
% Verify that time translation is a d-theta symmetry and calculate the
% corresponding integral.

pform {t,q(k),v(k),lam(k),tau,xi(k),eta(k)}=0,theta=1,f=0,
      {l,glq(k),glv(k),glt}=0;

tvector gam,y;

indexrange 1,2;

fdomain tau=tau(t,q(k),v(k)),xi=xi(t,q(k),v(k)),f=f(t,q(k),v(k));

```

```

l := 1/2*(v(1)**2 + v(2)**2) + m/r$           % The Lagrangian.

pform r=0;
fdomain r=r(q(k));
let @ (r,q 1)=q(1)/r,@ (r,q 2)=q(2)/r,q(1)**2+q(2)**2=r**2;

lam(k) := -m*q(k)/r;                          % The force.

gam := @ t + v(k)*@(q(k)) + lam(k)*@(v(k))$

eta(k) := gam _| d xi(k) - v(k)*gam _| d tau$

y := tau*@ t + xi(k)*@(q(k)) + eta(k)*@(v(k))$ % Symmetry generator.

theta := l*d t + @(l,v(k))*(d q(k) - v(k)*d t)$

factor @;

s := y _| theta - d f$

glq(k) := @(q k) _| s;
glv(k) := @(v k) _| s;
glt := @(t) _| s;

% Translation in time must generate a symmetry.
xi(k) := 0;
tau := 1;

glq k := glq k;
glv k := glv k;
glt;

% The corresponding integral is of course the energy.
integ := - y _| theta;

clear l,lam k,gam,eta k,y,theta,s,glq k,glv k,glt,t,q k,v k,tau,xi k;
remfdomain r,f,tau,xi;

% Problem:
% -----
% Calculate the "gradient" and "Laplacian" of a function and the "curl"
% and "divergence" of a one-form in elliptic coordinates.

coframe e u = sqrt(cosh(v)**2 - sin(u)**2)*d u,
           e v = sqrt(cosh(v)**2 - sin(u)**2)*d v,
           e phi = cos u*sinh v*d phi;

```



```

pform f=0;

fdomain f=f(u,v,phi);

factor e,^;
on rat,gcd;
order cosh v, sin u;
% The gradient:
d f;

factor @;
% The Laplacian:
# d # d f;

% Another way of calculating the Laplacian:
-#vardf(1/2*d f^#d f,f);

remfac @;

% Now calculate the "curl" and the "divergence" of a one-form.

pform w=1,a(k)=0;

fdomain a=a(u,v,phi);

w := a(-k)*e k;
% The curl:
x := # d w;

factor @;
% The divergence:
y := # d # w;

remfac @;
clear x,y,w,u,v,phi,e k,a k;
remfdomain a,f;

% Problem:
% -----
% Calculate in a spherical coordinate system the Navier Stokes equations.

coframe e r=d r, e theta =r*d theta, e phi = r*sin theta *d phi;
frame x;

fdomain v=v(t,r,theta,phi),p=p(r,theta,phi);

```

```

pform v(k)=0,p=0,w=1;

% We first calculate the convective derivative.

w := v(-k)*e(k)$

factor e; on rat;

cdv := @ (w,t) + (v(k)*x(-k)) | _ w - 1/2*d(v(k)*v(-k));

%next we calculate the viscous terms;

visc := nu*(d#d# w - #d#d w) + mu*d#d# w;

% Finally we add the pressure term and print the components of the
% whole equation.

pform nasteq=1,nast(k)=0;

nasteq := cdv - visc + 1/rho*d p$

factor @;

nast(-k) := x(-k) _| nasteq;

remfac @,e;

clear v k,x k,nast k,cdv,visc,p,w,nasteq,e k;
remfdomain p,v;

% Problem:
% -----
% Calculate from the Lagrangian of a vibrating rod the equation of
% motion and show that the invariance under time translation leads
% to a conserved current.

pform {y,x,t,q,j}=0,lagr=2;

fdomain y=y(x,t),q=q(x),j=j(x);

factor ^;

lagr := 1/2*(rho*q*@ (y,t)**2 - e*j*@ (y,x,x)**2)*d x^d t;

vardf(lagr,y);

% The Lagrangian does not explicitly depend on time; therefore the
% vector field @ t generates a symmetry. The conserved current is

```

```

pform c=1;
factor d;

c := noether(lagr,y,@ t);

% The exterior derivative of this must be zero or a multiple of the
% equation of motion (weak conservation law) to be a conserved current.

remfac d;

d c;

% i.e. it is a multiple of the equation of motion.

clear lagr,c,j,y,q;
remfdomain y,q,j;

% Problem:
% -----
% Show that the metric structure given by Eguchi and Hanson induces a
% self-dual curvature.
% c.f. T. Eguchi, P.B. Gilkey, A.J. Hanson, "Gravitation, Gauge Theories
% and Differential Geometry", Physics Reports 66, 213, 1980

for all x let cos(x)**2=1-sin(x)**2;

pform f=0,g=0;
fdomain f=f(r), g=g(r);

coframe   o(r) = f*d r,
          o(theta) = (r/2)*(sin(psi)*d theta - sin(theta)*cos(psi)*d phi),
          o(phi) = (r/2)*(-cos(psi)*d theta - sin(theta)*sin(psi)*d phi),
          o(psi) = (r/2)*g*(d psi + cos(theta)*d phi);

frame e;

pform gamma(a,b)=1,curv2(a,b)=2;
index_symmetries gamma(a,b),curv2(a,b): antisymmetric;

factor o;

gamma(-a,-b) := -(1/2)*( e(-a) _| (e(-c) _| (d o(-b)))
                      -e(-b) _| (e(-a) _| (d o(-c)))
                      +e(-c) _| (e(-b) _| (d o(-a))) )*o(c)$

curv2(-a,b) := d gamma(-a,b) + gamma(-c,b)^gamma(-a,c)$

```

```

let f=1/g,g=sqrt(1-(a/r)**4);

pform chck(k,l)=2;
index_symmetries chck(k,l): antisymmetric;

% The following has to be zero for a self-dual curvature.

chck(k,l) := 1/2*eps(k,l,m,n)*curv2(-m,-n) + curv2(k,l);

clear gamma(a,b),curv2(a,b),f,g,chck(a,b),o(k),e(k),r,phi,psi;
remfdomain f,g;

% Example: 6-dimensional FRW model with quadratic curvature terms in
% -----
% the Lagrangian (Lanczos and Gauss-Bonnet terms).
% cf. Henriques, Nuclear Physics, B277, 621 (1986)

for all x let cos(x)**2+sin(x)**2=1;

pform {r,s}=0;
fdomain r=r(t),s=s(t);

coframe o(t) = d t,
              o(1) = r*d u/(1 + k*(u**2)/4),
              o(2) = r*u*d theta/(1 + k*(u**2)/4),
              o(3) = r*u*sin(theta)*d phi/(1 + k*(u**2)/4),
              o(4) = s*d v1,
              o(5) = s*sin(v1)*d v2
with metric g =-o(t)*o(t)+o(1)*o(1)+o(2)*o(2)+o(3)*o(3)
              +o(4)*o(4)+o(5)*o(5);

frame e;

on nero; factor o,^;

riemannconx om;

pform curv(k,l)=2,{riemann(a,b,c,d),ricci(a,b),riccisc}=0;

index_symmetries curv(k,l): antisymmetric,
                             riemann(k,l,m,n): antisymmetric in {k,l},{m,n}
                                                symmetric in {{k,l},{m,n}},
                             ricci(k,l): symmetric;

curv(k,l) := d om(k,l) + om(k,-m)^om(m,l);

riemann(a,b,c,d) := e(d) _| (e (c) _| curv(a,b));

```

```

% The rest is done in the Ricci calculus language,

ricci(-a,-b) := riemann(c,-a,-d,-b)*g(-c,d);

riccisc := ricci(-a,-b)*g(a,b);

pform {laglanc,inv1,inv2} = 0;

index_symmetries riemc3(k,l),riemri(k,l),
                  hlang(k,l),einst(k,l): symmetric;

pform {riemc3(i,j),riemri(i,j)}=0;

riemc3(-i,-j) := riemann(-i,-k,-l,-m)*riemann(-j,k,l,m)$
inv1 := riemc3(-i,-j)*g(i,j);
riemri(-i,-j) := 2*riemann(-i,-k,-j,-l)*ricci(k,l)$
inv2 := ricci(-a,-b)*ricci(a,b);
laglanc := (1/2)*(inv1 - 4*inv2 + riccisc**2);

pform {einst(a,b),hlang(a,b)}=0;

hlang(-i,-j) := 2*(riemc3(-i,-j) - riemri(-i,-j) -
  2*ricci(-i,-k)*ricci(-j,k) +
  riccisc*ricci(-i,-j) - (1/2)*laglanc*g(-i,-j));

% The complete Einstein tensor:

einst(-i,-j) := (ricci(-i,-j) - (1/2)*riccisc*g(-i,-j))*alp1 +
hlang(-i,-j)*alp2$

alp1 := 1$
factor alp2;

einst(-i,-j) := einst(-i,-j);

clear o(k),e(k),riemc3(i,j),riemri(i,j),curv(k,l),riemann(a,b,c,d),
  ricci(a,b),riccisc,t,u,v1,v2,theta,phi,r,om(k,l),einst(a,b),
  hlang(a,b);

remfdomain r,s;

% Problem:
% -----
% Calculate for a given coframe and given torsion the Riemannian part and
% the torsion induced part of the connection. Calculate the curvature.

% For a more elaborate example see E.Schruefer, F.W. Hehl, J.D. McCrea,
% "Application of the REDUCE package EXCALC to the Poincare gauge field

```

```

% theory of gravity", GRG Journal, vol. 19, (1988) 197--218

pform {ff, gg}=0;

fdomain ff=ff(r), gg=gg(r);

coframe o(4) = d u + 2*b0*cos(theta)*d phi,
      o(1) = ff*(d u + 2*b0*cos(theta)*d phi) + d r,
      o(2) = gg*d theta,
      o(3) = gg*sin(theta)*d phi
with metric g = -o(4)*o(1)-o(4)*o(1)+o(2)*o(2)+o(3)*o(3);

frame e;

pform {tor(a), gwt(a)}=2, gamma(a,b)=1,
      {u1,u3,u5}=0;

index_symmetries gamma(a,b): antisymmetric;

fdomain u1=u1(r), u3=u3(r), u5=u5(r);

tor(4) := 0$

tor(1) := -u5*o(4)^o(1) - 2*u3*o(2)^o(3)$

tor(2) := u1*o(4)^o(2) + u3*o(4)^o(3)$

tor(3) := u1*o(4)^o(3) - u3*o(4)^o(2)$

gwt(-a) := d o(-a) - tor(-a)$

% The following is the combined connection.
% The Riemannian part could have equally well been calculated by the
% RIEMANNCONX statement.

gamma(-a,-b) := (1/2)*( e(-b) _| (e(-c) _| gwt(-a))
      +e(-c) _| (e(-a) _| gwt(-b))
      -e(-a) _| (e(-b) _| gwt(-c)) )*o(c);

pform curv(a,b)=2;
index_symmetries curv(a,b): antisymmetric;
factor ^;

curv(-a,b) := d gamma(-a,b) + gamma(-c,b)^gamma(-a,c);

clear o(k), e(k), curv(a,b), gamma(a,b), theta, phi, x, y, z, r, s, t, u, v, p, q, c, cs;
remfdomain u1, u3, u5, ff, gg;

showtime;

```

end;

16.21 FIDE: Finite difference method for partial differential equations

This package performs automation of the process of numerically solving partial differential equations systems (PDES) by means of computer algebra. For PDES solving, the finite difference method is applied. The computer algebra system REDUCE and the numerical programming language FORTRAN are used in the presented methodology. The main aim of this methodology is to speed up the process of preparing numerical programs for solving PDES. This process is quite often, especially for complicated systems, a tedious and time consuming task.

Documentation for this package is in plain text.

Author: Richard Liska.

16.22 FPS: Automatic calculation of formal power series

This package can expand a specific class of functions into their corresponding Laurent-Puiseux series.

Authors: Wolfram Koepf and Winfried Neun.

16.22.1 Introduction

This package can expand functions of certain type into their corresponding Laurent-Puiseux series as a sum of terms of the form

$$\sum_{k=0}^{\infty} a_k (x - x_0)^{mk/n+s}$$

where m is the ‘symmetry number’, s is the ‘shift number’, n is the ‘Puiseux number’, and x_0 is the ‘point of development’. The following types are supported:

- **functions of ‘rational type’**, which are either rational or have a rational derivative of some order;
- **functions of ‘hypergeometric type’** where $a(k+m)/a(k)$ is a rational function for some integer m ;
- **functions of ‘explike type’** which satisfy a linear homogeneous differential equation with constant coefficients.

The FPS package is an implementation of the method presented in [2]. The implementations of this package for MAPLE (by D. Gruntz) and MATHEMATICA (by W. Koepf) served as guidelines for this one.

Numerous examples can be found in [3]–[4], most of which are contained in the test file `fps.tst`. Many more examples can be found in the extensive bibliography of Hansen [1].

16.22.2 REDUCE operator FPS

The FPS Package must be loaded first by:

```
load FPS;
```

`FPS(f, x, x0)` tries to find a formal power series expansion for f with respect to the variable x at the point of development x_0 . It also works for formal Laurent (negative exponents) and Puiseux series (fractional exponents). If the third argument is omitted, then $x_0 := 0$ is assumed.

Examples: `FPS (asin (x) ^2, x)` results in

$$\text{infsum}\left(\frac{x^{2k} \cdot 2^{2k} \cdot \text{factorial}(k) \cdot x^2}{\text{factorial}(2k+1) \cdot (k+1)}, k, 0, \text{infinity}\right)$$

`FPS (sin x, x, pi)` gives

$$\text{infsum}\left(\frac{(-\pi + x)^{2k} \cdot (-1)^k \cdot (-\pi + x)^k}{\text{factorial}(2k+1)}, k, 0, \text{infinity}\right)$$

and `FPS (sqrt (2-x^2), x)` yields

$$\text{infsum}\left(\frac{-x^{2k} \cdot \sqrt{2} \cdot \text{factorial}(2k)}{8 \cdot \text{factorial}(k) \cdot (2k-1)}, k, 0, \text{infinity}\right)$$

Note: The result contains one or more `infsum` terms such that it does not interfere with the `REDUCE` operator `sum`. In graphical oriented `REDUCE` interfaces this operator results in the usual \sum notation.

If possible, the output is given using factorials. In some cases, the use of the Pochhammer symbol `pochhammer (a, k) := a(a+1) ··· (a+k-1)` is necessary.

The operator `FPS` uses the operator `SimpleDE` of the next section.

If an error message of type

Could not find the limit of:

occurs, you can set the corresponding limit yourself and try a recalculation. In the computation of `FPS (atan (cot (x)), x, 0)`, `REDUCE` is not able to find the value for the limit `limit (atan (cot (x)), x, 0)` since the `atan` function is multi-valued. One can choose the branch of `atan` such that this limit equals $\pi/2$ so that we may set

```
let limit (atan (cot (~x)), x, 0) => pi/2;
```

and a recalculation of `FPS (atan (cot (x)), x, 0)` yields the output $\pi - 2x$ which is the correct local series representation.

16.22.3 REDUCE operator `SimpleDE`

`SimpleDE(f, x)` tries to find a homogeneous linear differential equation with polynomial coefficients for f with respect to x . Make sure that y is not a used variable. The setting `factor df;` is recommended to receive a nicer output form.

Examples: `SimpleDE(asin(x)^2, x)` then results in

$$df(y, x, 3) * (x^2 - 1) + 3 * df(y, x, 2) * x + df(y, x)$$

`SimpleDE(exp(x^(1/3)), x)` gives

$$27 * df(y, x, 3) * x^2 + 54 * df(y, x, 2) * x + 6 * df(y, x) - y$$

and `SimpleDE(sqrt(2-x^2), x)` yields

$$df(y, x) * (x^2 - 2) - x * y$$

The depth for the search of a differential equation for f is controlled by the variable `fps_search_depth`; higher values for `fps_search_depth` will increase the chance to find the solution, but increases the complexity as well. The default value for `fps_search_depth` is 5. For `FPS(sin(x^(1/3)), x)`, or `SimpleDE(sin(x^(1/3)), x)` e. g., a setting `fps_search_depth:=6` is necessary.

The output of the FPS package can be influenced by the switch `tracefps`. Setting on `tracefps` causes various prints of intermediate results.

16.22.4 Problems in the current version

The handling of logarithmic singularities is not yet implemented.

The rational type implementation is not yet complete.

The support of special functions [5] will be part of the next version.

Bibliography

- [1] E. R. Hansen, *A table of series and products*. Prentice-Hall, Englewood Cliffs, NJ, 1975.

- [2] Wolfram Koepf, *Power Series in Computer Algebra*, J. Symbolic Computation 13 (1992)
- [3] Wolfram Koepf, *Examples for the Algorithmic Calculation of Formal Puiseux, Laurent and Power series*, SIGSAM Bulletin 27, 1993, 20-32.
- [4] Wolfram Koepf, *Algorithmic development of power series*. In: Artificial intelligence and symbolic mathematical computing, ed. by J. Calmet and J. A. Campbell, International Conference AISMC-1, Karlsruhe, Germany, August 1992, Proceedings, Lecture Notes in Computer Science **737**, Springer-Verlag, Berlin–Heidelberg, 1993, 195–213.
- [5] Wolfram Koepf, *Algorithmic work with orthogonal polynomials and special functions*. Konrad-Zuse-Zentrum Berlin (ZIB), Preprint SC 94-5, 1994.

16.23 GCREF: A Graph Cross Referencer

This package reuses the code of the RCREF package to create a graph displaying the interdependency of procedures in a Reduce source code file.

Authors: A. Dolzmann, T. Sturm.

16.23.1 Basic Usage

Similarly to the Reduce cross referencer, it is used via switches as follows:

```
load_package gcref;
on gcref;
in "<filename>.red";
off gcref;
```

At `off gcref;` the graph is printed to the screen in TGF format. To redirect this output to a file, use the following:

```
load_package gcref;
on gcref;
in "<filename>.red";
out "<filename>.tgf";
off gcref;
shut "<filename>.tgf";
```

16.23.2 Shell Script "gcref"

There is a shell script "gcref" in this directory automizing this like

```
./gcref filename.red
```

"gcref" is configured to use CSL Reduce. To use PSL Reduce instead, set \$REDUCE in the environment. To use PSL by default, define

```
REDUCE=redpsl
```

in line 3 of "gcref".

16.23.3 Redering with yED

The obtained TGF file can be viewed with a graph editor. I recommend using the free software yED, which is written in Java and available for many platforms.

http://www.yworks.com/en/products_yed_about.html

Note that TGF is not suitable for storing rendering information. After opening the TGF file with yED, the graph has to be rendered explicitly as follows:

- * From menu "Layout" choose "Hierarchical Layout".

To resize the nodes to the procedure names

- * from menu "Tools" choose "Fit Node to Label".

Feel free to experiment with yED and use other layout and layout options, which might be suitable for your particular software.

For saving your particular layout at the end, use the GRAPHML format instead of TGF.

16.24 GENTRAN: A code generation package

GENTRAN is an automatic code GENERator and TRANslator. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR, PASCAL or C code can be generated through a series of interactive commands or under the control of a template processing routine. Large expressions can be automatically segmented into subexpressions of manageable size, and a special file-handling mechanism maintains stacks of open I/O channels to allow output to be sent to any number of files simultaneously and to facilitate recursive invocation of the whole code generation process.

Author: Barbara L. Gates.

February 1991

GENTRAN is an automatic code GENERator and TRANslator which runs under REDUCE and VAXIMA. It constructs complete numerical programs based on sets of algorithmic specifications and symbolic expressions. Formatted FORTRAN, RATFOR or C code can be generated through a series of interactive commands or under the control of a template processing routine. Large expressions can be automatically segmented into subexpressions of manageable size, and a special file-handling mechanism maintains stacks of open I/O channels to allow output to be sent to any number of files simultaneously and to facilitate recursive invocation of the whole code generation process. GENTRAN provides the flexibility necessary to handle most code generation applications. This manual describes usage of the GENTRAN package for REDUCE.

Acknowledgements

The GENTRAN package was created at Kent State University to generate numerical code for computations in finite element analysis. I would like to thank Prof. Paul Wang for his guidance and many suggestions used in designing the original package for VAXIMA.

The second version of GENTRAN was implemented at Twente University of Technology to run under REDUCE. It was designed to be interfaced with a code optimization facility created by Dr. J. A. van Hulzen. I would like to thank Dr. van Hulzen for all of his help in the implementation of GENTRAN in RLISP during a stay at his university in The Netherlands.

Finally, I would like to thank Dr. Anthony Hearn of the RAND Corporation for his help in better integrating GENTRAN into the REDUCE environment.

16.24.1 INTRODUCTION

Solving a problem in science or engineering is often a two-step process. First the problem is modeled mathematically and derived symbolically to provide a set of formulas which describe how to solve the problem numerically. Next numerical programs are written based on this set of formulas to efficiently compute specific values for given sets of input. Computer algebra systems such as REDUCE provide powerful tools for use in the formula-derivation phase but only provide primitive program-coding tools. The GENTRAN package [?, ?, ?, ?] has been constructed to automate the tedious, time consuming and error-prone task of writing numerical programs based on a set of symbolic expressions.

The GENTRAN Code Generator and Translator

The GENTRAN code GENeration and TRANslation package, originally implemented in Franz LISP to run under VAXIMA [?], is now also implemented in RLISP to run under REDUCE. Although GENTRAN was originally created specifically to generate numerical code for use with an existing FORTRAN-based finite element analysis package [?, ?], it was designed to provide the flexibility required to handle most code generation applications. GENTRAN contains code generation commands, file-handling commands, mode switches, and global variables, all of which are accessible from both the algebraic and symbolic modes of REDUCE to give the user maximal control over the code generation process. Formatted FORTRAN [?], RATFOR [?], C [?], or PASCAL code can be generated from algorithmic specifications, i.e., descriptions of the behaviour of the target numerical program expressed in the REDUCE programming language, and from symbolically derived expressions and formulas.

In addition to arithmetic expressions and assignment statements, GENTRAN can also generate type declarations and control-flow structures. Code generation can be guided by user-supplied template file(s) to insert generated code into pre-existing program skeletons, or it can be accomplished interactively through a series of translation commands without the use of template files. Special mode switches enable the user to turn on or off specific features such as automatic segmentation of large expressions, and global variables allow the user to modify the code formatting process. Generated code can be sent to one or more files and, optionally, to the user's terminal. Stacks of open I/O channels facilitate temporary output redirection and recursive invocation of the code generation process.

Code Optimization

A code optimizer [?], which runs under REDUCE, has been constructed to reduce the arithmetic complexity of a set of symbolic expressions (see the SCOPE

package on page ??). It optimizes them by extracting common subexpressions and assigning them to temporary variables which are inserted in their places. The optimization technique is based on mapping the expressions onto a matrix of coefficients and exponents which are searched for patterns corresponding to the common subexpressions. Due to this process the size of the expressions is often considerably reduced.

GENTRAN and the Code Optimizer have been interfaced to make it possible to generate optimized numerical programs directly from REDUCE. Setting the switch GENTRANOPT **ON** specifies that all sequences of assignment statements are to be optimized before being converted to numerical code.

Organization of the Manual

The remainder of this manual is divided into five sections. Sections 16.24.2 and 16.24.3 describe code generation. Section 16.24.2 explains interactive code generation, the expression segmentation facility, and how temporary variables can be generated; then section 16.24.3 explains how code generation can be guided by a user-supplied template file. Section 16.24.4 describes methods of output redirection, and section 16.24.5 describes user-accessible global variables and mode switches which alter the code generation process. Finally section 16.24.6 presents three complete examples.

Typographic Conventions The following conventions are used in the syntactic definitions of commands in this manual:

- Command parts which must be typed exactly as shown are given in **BOLD PRINT**.
- User-supplied arguments are *emphasized*.
- [...] indicate optional command parts.

The syntax of each GENTRAN command is shown terminated with a ;. However, either ; or \$ can be used to terminate any command with the usual REDUCE meaning: ; indicates that the returned value is to be printed, while \$ indicates that printing of the returned value is to be suppressed.

Throughout this manual it is stated that file name arguments must be atoms. The exact type of atom (e.g., identifier or string) is system and/or site dependent. The instructions for the implementation being used should therefore be consulted.

16.24.2 Interactive Code Generation

GENTRAN generates numerical programs based on algorithmic specifications in the REDUCE programming language and derived symbolic expressions produced by REDUCE evaluations. FORTRAN, RATFOR, PASCAL or C code can be produced. Type declarations can be generated, and comments and other literal strings can be inserted into the generated code. In addition, large arithmetic expressions can be automatically segmented into a sequence of subexpressions of manageable size.

This section explains how to select the target language, generate code, control expression segmentation, and how to generate temporary variable names.

Target Language Selection

Before generating code, the target numerical language must be selected. GENTRAN is currently able to generate FORTRAN, RATFOR, PASCAL and C code. The global variable **GENTRANLANG!*** determines which type of code is produced. **GENTRANLANG!*** can be set in algebraic or symbolic mode. It can be set to any value, but only four atoms have special meaning: **FORTTRAN**, **RATFOR**, **PASCAL** and **C**. Any other value is assumed to mean **FORTTRAN**. **GENTRANLANG!*** is always initialized to **FORTTRAN**.

Translation

The **GENTRAN** (GENerate/TRANslate) command is used to generate numerical code and also to translate code from algorithmic specifications in the REDUCE programming language to code in the target numerical language. Section 16.24.2 explains code *generation*. This section explains code *translation*.

A substantial subset of all expressions and statements in the REDUCE programming language can be translated directly into numerical code. The **GENTRAN** command takes a REDUCE expression, statement, or procedure definition, and translates it into code in the target language.

Syntax:

GENTRAN *stmt* [**OUT** *f1,f2,...,fn*];

Arguments:

stmt is any REDUCE expression, statement (simple, compound, or group), or procedure definition that can be translated by GENTRAN into the target language² *stmt* may contain any number of calls to

²See 16.24.8 for a complete listing of REDUCE expressions and statements that can be translated.

the special functions **EVAL**, **DECLARE**, and **LITERAL** (see sections [16.24.2](#) – [16.24.2](#)).

$f1, f2, \dots, fn$ is an optional argument list containing one or more f 's, where each f is one of:

<i>an atom</i>	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN (see section 16.24.4)

Side Effects:

GENTRAN translates *stmt* into formatted code in the target language.

If the optional part of the command is not given, generated code is simply written to the current output file. However, if it is given, then the current output file is temporarily overridden. Generated code is written to each file represented by $f1, f2, \dots, fn$ for this command only. Files which were open prior to the call to **GENTRAN** will remain open after the call, and files which did not exist prior to the call will be created, opened, written to, and closed. The output stack will be exactly the same both before and after the call.

Returned Value:

GENTRAN returns the name(s) of the file(s) to which code was written.

Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS
    OVERWRITE FILE? (Y/N)
```

```
***** WRONG TYPE OF ARG
```

```
exp
```

```
***** CANNOT BE TRANSLATED
```

Example 25³

³When the **PERIOD** flag (default setting: ON) is turned on, all integers are automatically printed as real numbers except exponents, subscripts in subscripted variables, and index values in DO-type loops.

```

1: GENTRANLANG!* := 'FORTRAN$

2: GENTRAN
2:   FOR I:=1:N DO
2:     V(I) := 0$

      DO 25001 I=1,N
      V(I)=0.0
25001 CONTINUE

3: GENTRANLANG!* := 'RATFOR$

4: GENTRAN
4:   FOR I:=1:N DO
4:     FOR J:=I+1:N DO
4:       <<
4:         X(J,I) := X(I,J);
4:         Y(J,I) := Y(I,J)
4:       >>$

DO I=1,N
  DO J=I+1,N
  {
    X(J,I)=X(I,J)
    Y(J,I)=Y(I,J)
  }

5: GENTRANLANG!* := 'C$

6: GENTRAN
6:   P := FOR I:=1:N PRODUCT I$

{
  P=1;
  for (I=1;I<=N;++I)
    P*=I;
}

7: GENTRANLANG!* := 'PASCAL$

8: GENTRAN
8:   S := FOR I := 1:10 SUM V(I)$
BEGIN
  S:=0;

```

```

      FOR I:=1 TO 10 DO
        S:=S+V(I)
      END;

```

Translation is a convenient method of producing numerical code when the exact behaviour of the resultant code is known. It gives the REDUCE user who is familiar with the syntax of statements in the REDUCE programming language the ability to write code in a numerical programming language without knowing the exact syntactical requirements of the language. However the *real* power of the **GENTRAN** command lies in its ability to generate code: it can produce numerical code from symbolic expressions derived in REDUCE in addition to translating statements directly. This aspect is described in section 16.24.2.

Precision

By default **GENTRAN** generates constants and type declarations in single precision form. If the user requires double precision output then the switch **DOUBLE** must be set **ON**. This does the following:

- Declarations of appropriate type are converted to their double precision counterparts. In FORTRAN and RATFOR this means that objects of type *REAL* are converted to objects of type *DOUBLE PRECISION* and objects of type *COMPLEX* are converted to *COMPLEX*16*⁴. In C the counterpart of *float* is *double*, and of *int* is *long*. There is no complex data type and trying to translate complex objects causes an error.
- Similarly subprograms are given their correct type where appropriate.
- In FORTRAN and RATFOR *REAL* and *COMPLEX* numbers are printed with the correct double precision format.
- Intrinsic functions are converted to their double precision counterparts (e.g. in FORTRAN *SIN* → *DSIN* etc.).

Intrinsic FORTRAN and RATFOR functions. An attempt is made to convert the arguments of intrinsic functions to the correct type. For example:

```

5: GENTRAN f:=sin(1) $
      F=SIN(1.0)

6: GENTRAN f:=sin(x) $

```

⁴This is not part of the ANSI FORTRAN standard. Some compilers accept *DOUBLE COMPLEX* as well as, or instead of, *COMPLEX*16*, and some accept neither.

```
F=SIN (REAL (X) )
```

```
7: GENTRAN DECLARE <<x:real>>$
```

```
8: GENTRAN f:=sin(x) $
    F=SIN (X)
```

Which function is used to coerce the argument may, of course, depend on the setting of the switch **DOUBLE**.

Number of printed floating point digits. To ensure the correct number of floating point digits are generated it may be necessary to use either the **PRECISION** or **PRINT!-PRECISION** commands. The former alters the number of digits REDUCE calculates, the latter only the number of digits REDUCE prints. Each takes an integer argument. It is not possible to set the printed precision higher than the actual precision. Calling **PRINT!-PRECISION** with a negative argument causes the printed precision to revert to the actual precision.

```
1: on rounded$

2: precision 16$

3: 1/3;

0.333 33333 33333 333

4: print!-precision 6$

5: 1/3;

0.333333

6: print!-precision(-1)$

7: 1/3;

0.333 33333 33333 333
```

Code Generation: Evaluation Prior to Translation

Section [16.24.2](#) showed how REDUCE statements and expressions can be translated directly into the target language. This section shows how to indicate that parts

of those statements and expressions are to be handed to REDUCE to be evaluated before being translated. In other words, this section explains how to generate numerical code from algorithmic specifications (in the REDUCE programming language) and symbolic expressions. Each of the following four subsections describes a special function or operator that can be used to request partial or full evaluation of expressions prior to translation. Note that these functions and operators have the described effects *only* when applied to arguments to the **GENTRAN** function and that evaluation is done in algebraic or symbolic mode, depending on the value of the REDUCE variable **!*MODE**.

The EVAL Function Syntax:

EVAL *exp*

Argument:

exp is any REDUCE expression or statement which, after evaluation by REDUCE, results in an expression that can be translated by GENTRAN into the target language.

Side Effect:

When **EVAL** is called on an expression which is to be translated, it tells **GENTRAN** to give the expression to REDUCE for evaluation first, and then to translate the result of that evaluation.

Example 26

The following formula, F, has been derived symbolically:

$$2 * X^2 - 5 * X + 6$$

We wish to generate an assignment statement for the quotient of F and its derivative.

```
1: GENTRAN
1:      Q := EVAL (F) / EVAL (DF (F, X) ) $
```

$$Q = (2.0 * X^2 - (5.0 * X) + 6.0) / (4.0 * X - 5.0)$$

The := Operator In many applications, assignments must be generated in which the left-hand side is some known variable name, but the right-hand side is an expression that must be evaluated. For this reason, a special operator is provided to indicate that the expression on the right-hand side is to be evaluated prior to translation. This special operator is **:=** (i.e., the usual REDUCE assignment operator with an extra “:” on the right).

Example 27

```
1: GENTRAN
1:  DERIV :=: DF (X^4-X^3+2*x^2+1, X) $

      DERIV=4.0*X**3-(3.0*X**2)+4.0*X
```

Each built-in operator in REDUCE has an alternative alphanumeric identifier associated with it. Similarly, the GENTRAN **:=** operator has a special identifier associated with it: **RSETQ** may be used interchangeably with **:=** on input.

The ::= Operator When assignments to matrix or array elements must be generated, many times the indices of the element must be evaluated first. The special operator **::=** can be used within a call to **GENTRAN** to indicate that the indices of the matrix or array element on the left-hand side of the assignment are to be evaluated prior to translation. (This is the usual REDUCE assignment operator with an extra “:” on the left.)

Example 28

We wish to generate assignments which assign zeros to all elements on the main diagonal of M, an n x n matrix.

```
10: FOR j := 1 : 8 DO
10:     GENTRAN
10:         M(j, j) ::= 0$

      M(1,1)=0.0
      M(2,2)=0.0
      :
      :
      M(8,8)=0.0
```

LSETQ may be used interchangeably with **::=** on input.

The ::= Operator In applications in which evaluated expressions are to be assigned to array elements with evaluated subscripts, the **::=** operator can be used.

It is a combination of the `::=` and `:=` operators described in sections [16.24.2](#) and [16.24.2](#).

Example 29

The following matrix, M, has been derived symbolically:

```
(  A    0  -1   1)
(                )
(  0    B    0   0)
(                )
( -1    0    C  -1)
(                )
(  1    0  -1   D)
```

We wish to generate assignment statements for those elements on the main diagonal of the matrix.

```
10: FOR j := 1 : 4 DO
10:     GENTRAN
10:         M(j, j) ::= M(j, j) $
```

```
M(1, 1) = A
M(2, 2) = B
M(3, 3) = C
M(4, 4) = D
```

The alternative alphanumeric identifier associated with **::=** is **LRSETQ**.

Explicit Type Declarations

Type declarations are automatically generated each time a subprogram heading is generated. Type declarations are constructed from information stored in the GENTRAN symbol table. The user can place entries into the symbol table explicitly through calls to the special GENTRAN function **DECLARE**.

Syntax:

```
DECLARE v1,v2,... ,vn : type;
or
DECLARE
<<
    v11,v12,... ,v1n : type1;
    v21,v22,... ,v2n : type2;
    :
    :
    vn1,vnn,... ,vnn : typen;
>>;
```

Arguments:

Each v_1, v_2, \dots, v_n is a list of one or more variables (optionally subscripted to indicate array dimensions), or variable ranges (two letters separated by a “-”). v ’s are not evaluated unless given as arguments to **EVAL**.

Each *type* is a variable type in the target language. Each must be an atom, optionally preceded by the atom **IMPLICIT**. *type*’s are not evaluated unless given as arguments to **EVAL**.

Side Effect:

Entries are placed in the symbol table for each variable or variable range declared in the call to this function. The function call itself is removed from the statement group being translated. Then after translation, type declarations are generated from these symbol table entries before the resulting executable statements are printed.

Diagnostic Message:

```
***** INVALID SYNTAX
```

Example 30

```
1: GENTRAN
1: <<
1:     DECLARE
1:     <<
1:         A-H, O-Z : IMPLICIT REAL;
1:         M(4,4)    : INTEGER
1:     >>;
1:     FOR I:=1:4 DO
1:         FOR J:=1:4 DO
1:             IF I=J
1:                 THEN M(I,J):=1
1:                 ELSE M(I,J):=0;
1:     DECLARE I, J : INTEGER;
1: >>$

      IMPLICIT REAL (A-H,O-Z)
      INTEGER M(4,4), I, J
      DO 25001 I=1,4
          DO 25002 J=1,4
```

```

                IF (I.EQ.J) THEN
                    M(I,J)=1.0
                ELSE
                    M(I,J)=0.0
                ENDIF
25002          CONTINUE
25001 CONTINUE

```

The **DECLARE** statement can also be used to declare subprogram types (i.e. **SUBROUTINE** or **FUNCTION**) for FORTRAN and RATFOR code, and function types for all four languages.

Example 31

```

1: GENTRANLANG!* := 'RATFOR$

2: GENTRAN
2:      PROCEDURE FAC N;
2:      BEGIN
2:      DECLARE
2:      <<
2:          FAC : FUNCTION;
2:          FAC, N : INTEGER
2:      >>;
2:      F := FOR I:=1:N PRODUCT I;
2:      DECLARE F, I : INTEGER;
2:      RETURN F
2:      END$

INTEGER FUNCTION FAC(N)
INTEGER N,F,I
{
    F=1
    DO I=1,N
        F=F*I
    }
RETURN (F)
END

3: GENTRANLANG!* := 'C$
4: GENTRAN
4:      PROCEDURE FAC N;
4:      BEGIN
4:      DECLARE FAC, N, I, F : INTEGER;

```

```

4:      F := FOR I:=1:N PRODUCT I;
4:      RETURN F
4:      END$

int FAC(N)
int N;
{
    int I,F;
    {
        F=1;
        for (I=1; I<=N; ++I)
            F*=I;
    }
    return(F);
}

```

When generating code for subscripted variables (i.e., matrix and array elements), it is important to keep several things in mind. First of all, when a REDUCE array is declared with a declaration such as

ARRAY A(*n*)\$

where *n* is a positive integer, **A** is actually being declared to be of size **n+1**. Each of the elements **A(0)**, **A(1)**, ... , **A(n)** can be used. However, a FORTRAN or RATFOR declaration such as

DIMENSION A(*n*)

declares **A** only to be of size **n**. Only the elements **A(1)**, **A(2)**, ... , **A(n)** can be used. Furthermore, a C declaration such as

float A[*n*];

declares **A** to be of size **n** with elements referred to as **A[0]**, **A[1]**, ... , **A[n-1]**.

To resolve these array size and subscripting conflicts, the user should remember the following:

- *All REDUCE array subscripts are translated literally.* Therefore it is the user's responsibility to be sure that array elements with subscript 0 are not translated into FORTRAN or RATFOR.
- Since C and PASCAL arrays allow elements with a subscript of 0, when an array is declared to be of size *n* by the user, *the actual generated type declaration will be of size n+1* so that the user can translate elements with

subscripts from 0, and up to and including n .

If the user is generating C code, it is possible to produce declarations for arrays with unknown bounds:

```
5: gentran declare <<x(*,*) :real;y(*) :integer>>$

6: gendecs nil;
float X[ ][ ];
int Y[ ];
```

Implicit Type Declarations

Some type declarations can be made automatically if the switch **GETDECS** is **ON**. In this case:

1. The indices of loops are automatically declared to be integers.
2. There is a global variable **DEFTYPE!***, which is the default type given to objects. Subprograms, their parameters, and local scalar objects are automatically assigned this type. Note that types such as **REAL*8** or **DOUBLE PRECISION** should not be used as, if **DOUBLE** is on, then a default type of **REAL** will in fact be **DOUBLE PRECISION** anyway.
3. If **GENTRAN** is used to translate a **REDUCE** procedure, then it assigns objects declared **SCALAR** the type given by **DEFTYPE!***. Note that it is legal to use the commands **INTEGER** and **REAL** in the place of **SCALAR**, which allows the user to specify an actual type. The procedure may also be given a return type, in which case that is used as the default. For example:

```
2: on getdecs,gendecs$

3: GENTRAN
3: real procedure f(x);
3: begin integer n;real y;
3:     n := 4;
3:     y := n/(1+x)^2;
3:     return y;
3: end;
    REAL FUNCTION F(X)
    INTEGER N
    REAL X,Y
    N=4
```

```

Y=N/ (1.0+X) **2
F=Y
RETURN
END

```

More about Type Declarations

A check is made on output to ensure that all types generated are legal ones. This is necessary since **DEFTYPE!*** can be set to anything. Note that **DEFTYPE!*** ought normally to be given a simple type as its value, such as **REAL**, **INTEGER**, or **COMPLEX**, since this will always be translated into the corresponding type in the target language on output.

An entry is removed from the symbol table once a declaration has been generated for it. The **KEEPDECS** switch (by default **OFF**) disables this, allowing a user to check the types of objects which GENTRAN has generated (useful if they are being generated automatically)

Comments and Literal Strings

Comments and other strings of characters can be inserted directly into the stream of generated code through a call to the special function **LITERAL**.

Syntax:

LITERAL *arg1,arg2,...,argn*;

Arguments:

arg1,arg2,...,argn is an argument list containing one or more *arg*'s, where each *arg* either is, or evaluates to, an atom. The atoms **TAB!*** and **CR!*** have special meanings. *arg*'s are not evaluated unless given as arguments to **EVAL**.

Side Effect:

This statement is replaced by the character sequence resulting from concatenation of the given atoms. Double quotes are stripped from all string type *arg*'s, and the reserved atoms **TAB!*** and **CR!*** are replaced by a tab to the current level of indentation, and an end-of-line character, respectively.

Example 32

Suppose N has value 10.

```

1: GENTRANLANG!* := 'FORTRAN$

2: GENTRAN
2: <<
2:     LITERAL
2:     "C",TAB!*, "--THIS IS A FORTRAN COMMENT--",CR!*,
2:     "C",CR!*;
2:     LITERAL
2:     TAB!*, "DATA N/", EVAL(N), "/", CR!*
2: >>$

C     --THIS IS A FORTRAN COMMENT--
C
      DATA N/10/

3: GENTRANLANG!* := 'RATFOR$
4: GENTRAN
4:     FOR I:=1:N DO
4:     <<
4:         LITERAL
4:         TAB!*, "# THIS IS A RATFOR COMMENT",CR!*;
4:         LITERAL
4:         TAB!*, "WRITE(6,10) (M(I,J),J=1,N)",CR!*,
4:         10,TAB!*, "FORMAT(1X,10(I5,3X))",CR!*
4:     >>$

DO I=1,N
    {
        # THIS IS A RATFOR COMMENT
        WRITE(6,10) (M(I,J),J=1,N)
10    FORMAT(1X,10(I5,3X))
    }

5: GENTRANLANG!* := 'C$
6: GENTRAN
6: <<
6:     X:=0;
6:     LITERAL "/* THIS IS A",CR!*,
6:     "    C COMMENT */",CR!*
6: >>$

{

```



```

      X=0.0;
/* THIS IS A
   C COMMENT */
}

7:  GENTRANLANG!* := 'PASCAL$

8:  GENTRAN
8:  <<
8:      X := SIN(Y);
8:      LITERAL "{ THIS IS A PASCAL COMMENT }", CR!*
8:  >>$
BEGIN
      X:=SIN(Y)
{ THIS IS A PASCAL COMMENT }
END;

```

Expression Segmentation

Symbolic derivations can easily produce formulas that can be anywhere from a few lines to several pages in length. Such formulas can be translated into numerical assignment statements, but unless they are broken into smaller pieces they may be too long for a compiler to handle. (The maximum number of continuation lines for one statement allowed by most FORTRAN compilers is only 19.) Therefore GENTRAN contains a segmentation facility which automatically *segments*, or breaks down unreasonably large expressions.

The segmentation facility generates a sequence of assignment statements, each of which assigns a subexpression to an automatically generated temporary variable. This sequence is generated in such a way that temporary variables are re-used as soon as possible, thereby keeping the number of automatically generated variables to a minimum. The facility can be turned on or off by setting the mode switch **GENTRANSEG** accordingly (i.e., by calling the REDUCE function **ON** or **OFF** on it). The user can control the maximum allowable expression size by setting the variable **MAXEXPPRINTLEN!*** to the maximum number of characters allowed in an expression printed in the target language (excluding spaces automatically printed by the formatter). The **GENTRANSEG** switch is on initially, and **MAXEXPPRINTLEN!*** is initialized to 800.

Example 33

```
1: ON EXP$
```

```

2: JUNK1 := (A+B+C+D) ^ 2$

3: MAXEXPPRINTLEN!* := 24$

4: GENTRAN VAL :=: JUNK1$

    T0=A**2+2.0*A*B
    T0=T0+2.0*A*C+2.0*A*D
    T0=T0+B**2+2.0*B*C
    T0=T0+2.0*B*D+C**2
    VAL=T0+2.0*C*D+D**2

5: JUNK2 := JUNK1 / (E+F+G) $

6: MAXEXPPRINTLEN!* := 23$

7: GENTRANLANG!* := 'C$

8: GENTRAN VAL :=: JUNK2$

{
    T0=power(A,2)+2.0*A*B;
    T0+=2.0*A*C;
    T0=T0+2.0*A*D+power(B,2);
    T0+=2.0*B*C;
    T0=T0+2.0*B*D+power(C,2);
    T0=T0+2.0*C*D+power(D,2);
    VAL=T0/(exp(1.0)+F+G);
}

```

Implicit Type Declarations When the segmentation routine generates temporary variables, it places type declarations in the symbol table for those variables if possible. It uses the following rules to determine their type:

- (1) If the type of the variable to which the large expression is being assigned is already known (i.e., has been declared by the user), then the temporary variables will be declared to be of that same type.
- (2) If the global variable **TEMPVARTYPE!*** has a non-NIL value, then the temporary variables are declared to be of that type.
- (3) Otherwise, the variables are not declared.

Example 34

```

1: MAXEXPPRINTLEN!* := 20$

2: TEMPVARTYPE!* := 'REAL$

3: GENTRAN
3: <<
3:     DECLARE ISUM : INTEGER;
3:     ISUM := II+JJ+2*KK+LL+10*MM+NN;
3:     PROD := V(X,Y)*SIN(X)*COS(Y^2)*(X+Y+Z^2)
3: >>$

      INTEGER ISUM,T0
      REAL T1
      T0=II+JJ+2.0*KK+LL
      ISUM=T0+10.0*MM+NN
      T1=V(X,Y)*SIN(X)*COS(Y**2)
      PROD=T1*(X+Y+Z**2)

```

Generation of Temporary Variable Names

As we have just seen, GENTRAN's segmentation module generates temporary variables and places type declarations in the symbol table for them whenever possible. Various other modules also generate variables and corresponding declarations. All of these modules call one special GENTRAN function each time they need a temporary variable name. This function is **TEMPVAR**. There are situations in which it may be convenient for the user to be able to generate temporary variable names directly.⁵ Therefore **TEMPVAR** is a user-accessible function which may be called from both the algebraic and symbolic modes of REDUCE.

Syntax:

TEMPVAR *type*

Argument:

type is an atom which either indicates the variable type in the target language (INTEGER, REAL, etc.), or is **NIL** if the variable type is unknown.

Side Effects:

⁵One such example is suppression of the simplification process to generate numerical code which is more efficient. See the example in section 16.24.6 on page 386.

TEMPVAR creates temporary variable names by repeatedly concatenating the values of the global variables **TEMPVARNAME!*** (which has a default value of **T**) and **TEMPVARNUM!*** (which is initially set to 0) and incrementing **TEMPVARNUM!*** until a variable name is created which satisfies one of the following conditions:

- (1) It was not generated previously, and it has not been declared by the user.
- (2) It was previously generated to hold the same type of value that it must hold this time (e.g. **INTEGER**, **REAL**, etc.), and the value assigned to it previously is no longer needed.

If *type* is a non-NIL argument, or if *type* is **NIL** and the global variable **TEMPVARTYPE!*** (initially **NIL**) has been set to a non-NIL value, then a type entry for the generated variable name is placed in the symbol table.

Returned Value:

TEMPVAR returns an atom which can be used as a variable.

Note: It is the user's responsibility to set **TEMPVARNAME!*** and **TEMPVARNUM!*** to values such that generated variable names will not clash with variables used elsewhere in the program unless those variables have been declared.

Marking Temporary Variables In section 16.24.2 we saw that a temporary variable name (of a certain type) can be regenerated when the value previously assigned to it is no longer needed. This section describes a function which *marks* a variable to indicate that it currently holds a significant value, and the next section describes functions which *unmark* variables to indicate that the values they hold are no longer significant.

Syntax:

MARKVAR *var*

Argument:

var is an atom.

Side Effects:

MARKVAR sets a flag on *var*'s property list to indicate that *var* currently holds a significant value.

Returned Value:

MARKVAR returns *var*.

Example 35

The following matrix, M has been derived symbolically:

$$\begin{pmatrix} X*(Y+Z) & 0 & X*Z \\ (& &) \\ (& -X & X+Y & 0) \\ (& &) \\ (& X*Z & 0 & Z**2) \end{pmatrix}$$

We wish to replace each non-zero element by a generated variable name to prevent these expressions from being resubstituted into further calculations. (We will also record these substitutions in the numerical program we are constructing by generating assignment statements.)⁶

```

9:  SHARE var$

10:  FOR j := 1 : 3 DO
10:      FOR k := 1 : 3 DO
10:          IF M(j,k) NEQ 0 THEN
10:              <<
10:                  var := TEMPVAR(NIL);
10:                  MARKVAR var;
10:                  GENTRAN
10:                      EVAL(var) :=: M(j,k);
10:                  M(j,k) := var
10:              >>$

```

```

T0=X*(Y+Z)
T1=X*Z
T2=-X
T3=X+Y
T4=X*Z
T5=Z**2

```

Now matrix M contains the following entries:

⁶ Note: **MARKVAR** is a symbolic mode procedure. Therefore, the name of each variable whose value is to be passed to it from algebraic mode must appear in a **SHARE** declaration. This tells REDUCE to share the variable's value between algebraic and symbolic modes.

```

(T0      0   T1)
(
(T2   T3      0)
(
(T4      0   T5)

```

Unmarking Temporary Variables After the value assigned to a temporary variable has been used in the numerical program and is no longer needed, the variable name can be *unmarked* with the **UNMARKVAR** function.

Syntax:

UNMARKVAR *var*;

Argument:

var is an atom (variable name) or an expression containing one or more variable names.

Side Effect:

UNMARKVAR resets flags on the property lists of all variable names in *var* to indicate that they do not hold significant values any longer.

Enabling and Disabling Generation of Type Declarations

GENTRAN maintains a symbol table of variable type and dimension information. It adds information to the symbol table by processing user-supplied calls to the **DECLARE** function (see Section 16.24.2) and as a side effect of generating temporary variable names (see Sections 16.24.2 and 16.24.2). All information is stored in the symbol table until GENTRAN is ready to print formatted numerical code. Since programming languages such as FORTRAN require that type declarations appear before executable statements, GENTRAN automatically extracts all relevant type information and prints it in the form of type declarations before printing executable statements. This feature is useful when the entire body of a (sub)program is generated at one time: in this case, type declarations are printed before any executable code. However, if the user chooses to generate code in pieces, the resulting code may have type declarations interleaved with executable code. For this reason, the user may turn the **GENDECS** mode switch on or off, depending on whether or not s/he chooses to use this feature.

In the following we re-examine the example of Section 16.24.2.

Example 36

```

1: MAXEXPPRINTLEN!* := 20$

2: TEMPVARTYPE!* := 'REAL!*8$

3: GENTRAN
3: <<
3:     DECLARE ISUM : INTEGER;
3:     ISUM := II+JJ+2*KK+LL+10*MM+NN
3: >>$

        INTEGER ISUM,T0
        T0=II+JJ+2*KK+LL
        ISUM=T0+10*MM+NN

4: GENTRAN PROD := V(X,Y)*SIN(X)*COS(Y^2)*(X+Y+Z^2)$

        REAL*8 T2
        T2=V(X,Y)*SIN(REAL(X))*COS(REAL(Y**2))
        PROD=T2*(X+Y+Z**2)

5: OFF GENDECS$

6: GENTRAN
6: <<
6:     DECLARE ISUM : INTEGER;
6:     ISUM := II+JJ+2*KK+LL+10*MM+NN
6: >>$

        T0=II+JJ+2*KK+LL
        ISUM=T0+10*MM+NN

7: GENTRAN PROD := V(X,Y)*SIN(X)*COS(Y^2)*(X+Y+Z^2)$

        T2=V(X,Y)*SIN(REAL(X))*COS(REAL(Y**2))
        PROD=T2*(X+Y+Z**2)

```

In Section [16.24.3](#) we will explain how to further control the generation of type declarations.

Complex Numbers

With the switch **COMPLEX** set **ON**, GENTRAN will generate the correct representation for a complex number in the given precision provided that:

1. The current language supports a complex data type (if it doesn't then an error results);
2. The complex quantity is evaluated by REDUCE to give an object of the correct domain; i.e.

```
GENTRAN x:=: 1+i;

GENTRAN x:= eval 1+i;

z := 1+i;
GENTRAN x:=: z;
```

will all generate the correct result, as will their Symbolic mode equivalents, while:

```
GENTRAN x := 1+i;
```

will not.

Intrinsic Functions

A warning is issued if a standard REDUCE function is encountered which does not have an intrinsic counterpart in the target language (e.g. *cot*, *sec* etc.). Output is not halted in case this is a user-supplied function, either via a REDUCE definition or within a GENTRAN template.

The types of intrinsic FORTRAN functions are coerced to reals (in the correct precision) as the following examples demonstrate:

```
19: GENTRAN x:=sin(0) $
    X=SIN(0.0)

20: GENTRAN x:=cos(A) $
    X=COS(REAL(A) )

21: ON DOUBLE$

22: GENTRAN x := log(1) $
    X=DLOG(1.0D0)

23: GENTRAN x := exp(B) $
    X=DEXP(DBLE(B) )
```



```
24: GENTRAN DECLARE <<b:real>>$
```

```
25: GENTRAN x := exp(B) $
    X=DEXP(B)
```

Miscellaneous

MAKECALLS A statement like:

```
GENTRAN x^2+1$
```

will yield the result:

```
X**2+1
```

but, under normal circumstances, a statement like:

```
GENTRAN sin(x) $
```

will yield the result:

```
CALL SIN(X)
```

The switch **MAKECALLS** (OFF by default) will make GENTRAN yield

```
SIN(X)
```

This is useful if you don't know in advance what the form of the expression which you are translating is going to be.

E When GENTRAN encounters e it translates it into EXP(1), and when GENTRAN encounters e^x it is translated to EXP(X). This is then translated into the correct statement in the given language and precision. Note that it is still possible to do something like:

```
GENTRAN e:=:e;
```

and get the correct result.

Booleans

Some languages, like Fortran-77, have a boolean data type. Others, like C, do not. When translating Reduce code into a language with a boolean data type, GENTRAN will recognise the special identifiers *true* and *false*. For example:

```
3: gentran <<declare t:logical ;t:=true>>;
    LOGICAL T
    T=.TRUE.
```

16.24.3 Template Processing

In some code generation applications pieces of the target numerical program are known in advance. A *template* file containing a program outline is supplied by the user, and formulas are derived in REDUCE, converted to numerical code, and inserted in the corresponding places in the program outline to form a complete numerical program. A template processor is provided by GENTRAN for use in these applications.

The Template Processing Command

Syntax:

GENTRANIN *f1,f2,... fm* [**OUT** *f1,f2,... fn*];

Arguments:

f1,f2,... fm is an argument list containing one or more *f*'s, where each *f* is one of:

<i>an atom</i>	=	a template (input) file
T	=	the terminal

f1,f2,... fn is an optional argument list containing one or more *f*'s, where each *f* is one of:

<i>an atom</i>	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN (see section 16.24.4)

Side Effects:

GENTRANIN processes each template file $f1, f2, \dots, fn$ sequentially.

A template file may contain any number of parts, each of which is either an active or an inactive part. All active parts start with the character sequence **;BEGIN;** and end with **;END;**. The end of the template file is indicated by an extra **;END;** character sequence.

Inactive parts of template files are assumed to contain code in the target language (FORTRAN, RATFOR, PASCAL or C, depending on the value of the global variable **GENTRANLANG!***). All inactive parts are copied to the output. Comments delimited by the appropriate characters,

C ... <cr>	FORTRAN (beginning in column 1)
# ... <cr>	RATFOR
/* ... */	C
{ ... } or *(...)*	PASCAL

are also copied in their entirety to the output. Thus the character sequences **;BEGIN;** and **;END;** have no special meanings within comments.

Active parts may contain any number of REDUCE expressions, statements, and commands. They are not copied directly to the output. Instead, they are given to REDUCE for evaluation in algebraic mode⁷. All output generated by each evaluation is sent to the output file(s). Returned values are only printed on the terminal.

Active parts will most likely contain calls to **GENTRAN** to generate code. This means that the result of processing a template file will be the original template file with all active parts replaced by generated code.

If **OUT** $f1, f2, \dots, fn$ is not given, generated code is simply written to the current-output file.

However, if **OUT** $f1, f2, \dots, fn$ is given, then the current-output file is temporarily overridden. Generated code is written to each file represented by $f1, f2, \dots, fn$ for this command only. Files which were open prior to the call to **GENTRANIN** will remain open after the call, and files which did not exist prior to the call will be created, opened, written to, and closed. The output-stack will be exactly the same both before and after the call.

Returned Value:

⁷ Active parts are evaluated in algebraic mode unless the mode is explicitly changed to symbolic from within the active part itself. This is true no matter which mode the system was in when the template processor was called.

GENTRANIN returns the names of all files written to by this command.

Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS
    OVERWRITE FILE? (Y/N)

***** NONEXISTENT INPUT FILE

***** TEMPLATE FILE ALREADY OPEN FOR INPUT

***** WRONG TYPE OF ARG
```

Example 37

Suppose we wish to generate a FORTRAN subprogram to compute the determinant of a 3 x 3 matrix. We can construct a template file with an outline of the FORTRAN subprogram and REDUCE and GENTRAN commands to fill it in:

Contents of file `det.tem`:

```
      REAL FUNCTION DET(M)
      REAL M(3,3)
;BEGIN;
      OPERATOR M$
      MATRIX MM(3,3)$
      MM := MAT( (M(1,1),M(1,2),M(1,3)),
                  (M(2,1),M(2,2),M(2,3)),
                  (M(3,1),M(3,2),M(3,3)) )$
      GENTRAN DET :=: DET(MM)$
;END;
      RETURN
      END
;END;
```

Now we can generate a FORTRAN subprogram with the following REDUCE session:

```
1: GENTRANLANG!* := 'FORTRAN$

2: GENTRANIN
2:      "det.tem"
2: OUT "det.f"$
```

Contents of file det.f:

```

REAL FUNCTION DET(M)
REAL M(3,3)
DET=M(3,3)*M(2,2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)
. *M(2,3)*M(1,1))+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1
. ,2)-(M(3,1)*M(2,2)*M(1,3))
RETURN
END

```

Copying Files into Template Files

Template files can be copied into other template files with recursive calls to **GEN-TRANIN**; i.e., by calling **GENTRANIN** from the active part of a template file.

For example, suppose we wish to copy the contents of a file containing a subprogram into a file containing a main program. We will call **GENTRANIN** to do the copying, so the subprogram file must have **;END;** on its last line:

Contents of file det.tem:

```

REAL FUNCTION DET(M)
REAL M(3,3)
DET=M(3,3)*M(2,2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)
. *M(2,3)*M(1,1))+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1
. ,2)-(M(3,1)*M(2,2)*M(1,3))
RETURN
END
;END;

```

Now the template file for the main program can be constructed with an active part which will include file det.tem:

Contents of file main.tem:

```

C
C  MAIN PROGRAM
C
      REAL M(3,3),DET
      WRITE(6,*) 'ENTER 3 x 3 MATRIX'
      DO 100 I=1,3
         READ(5,*) (M(I,J),J=1,3)
100    CONTINUE
      WRITE(6,*) ' DET = ', DET(M)
      STOP

```

```

        END
C
C  DETERMINANT CALCULATION
C
;BEGIN;
        GENTRANIN "det.tem"$
;END;
;END;

```

The following REDUCE session will create the file `main.f`:

```

1: GENTRANIN
1:      "main.tem"
1: OUT "main.f"$

```

Contents of file `main.f`:

```

C
C  MAIN PROGRAM
C
        REAL M(3,3),DET
        WRITE(6,*) 'ENTER 3 x 3 MATRIX'
        DO 100 I=1,3
            READ(5,*) (M(I,J),J=1,3)
100    CONTINUE
        WRITE(6,*) ' DET = ', DET(M)
        STOP
        END
C
C  DETERMINANT CALCULATION
C
        REAL FUNCTION DET(M)
        REAL M(3,3)
        DET=M(3,3)*M(2,2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)
        . *M(2,3)*M(1,1))+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*
        . M(1,2)-(M(3,1)*M(2,2)*M(1,3))
        RETURN
        END

```

The Template File Stack

The REDUCE **IN** command takes one or more file names as arguments. REDUCE reads each of the given files and executes all statements and commands, any of

which may be another **IN** command. A stack of input file names is maintained by **REDUCE** to allow recursive invocation of the **IN** command. Similarly, a stack of template file names is maintained by **GENTRAN** to facilitate recursive invocation of the template processor. Section 16.24.3 showed that the **GENTRANIN** command can be called recursively to copy files into other files. This section shows that template files which are copied into other template files can also contain active parts, and thus the whole code generation process can be invoked recursively.

We can generalize the example of section 16.24.3 by generating code recursively. We can extend it to generate code which will compute entries of the inverse matrix, also. Suppose we have created the file `init.red`, which contains **REDUCE** commands to create an nxn matrix **MM** and initialize its entries to $M(1,1)$, $M(1,2)$, \dots , $M(n,n)$, for some user-entered value of n :

Contents of file `init.red`:

```
OPERATOR M$
MATRIX MM(n,n) $
FOR J := 1 : n DO
    FOR K := 1 : n DO
        MM(J,K) := M(J,K) $
END$
```

We have also created template files `det.tem` and `inv.tem` which contain outlines of **FORTRAN** subprograms to compute the determinant and inverse of an nxn matrix, respectively:

Contents of file `det.tem`:

```
REAL FUNCTION DET(M)
;BEGIN;
    GENTRAN
    <<
        DECLARE M(EVAL(n),EVAL(n)) : REAL;
        DET :=: DET(MM)
    >>$
;END;
RETURN
END
;END;
```

Contents of file `inv.tem`:

```
SUBROUTINE INV(M,MINV)
;BEGIN;
```

```

GENTRAN
<<
    DECLARE M(EVAL(n),EVAL(n)),
    MINV(EVAL(n),EVAL(n)) : REAL;
    MINV :=: MM^(-1)
>>$
;END;
    RETURN
    END
;END;

```

Now we can construct a template file with a generalized version of the main program given in section 16.24.3 and can place **GENTRANIN** commands in this file to generate code recursively from the template files det.tem and inv.tem:

Contents of file main.tem:

```

C
C  MAIN PROGRAM
C
;BEGIN;
    GENTRAN
    <<
        DECLARE
        <<
            M(EVAL(n),EVAL(n)),
            DET,
            MINV(EVAL(n),EVAL(n)) : REAL;
            N : INTEGER
        >>;
        LITERAL TAB!*,"DATA N/", EVAL(n), "/", CR!*
    >>$
;END;

    WRITE(6,*) 'ENTER ', N, 'x', N, ' MATRIX'
    DO 100 I=1,N
        READ(5,*) (M(I,J),J=1,N)
100  CONTINUE
    WRITE(6,*) ' DET = ', DET(M)
    WRITE(6,*) ' INVERSE MATRIX:'
    CALL INV(M,MINV)
    DO 200 I=1,N
        WRITE(6,*) (MINV(I,J),J=1,N)
200  CONTINUE
    STOP

```



```

      END
C
C  DETERMINANT CALCULATION
C
;BEGIN;
      GENTRANIN "det.tem"$
;END;
C
C  INVERSE CALCULATION
C
;BEGIN;
      GENTRANIN "inv.tem"$
;END;
;END;

```

The following REDUCE session will create the file `main.f`:

```

1: n := 3$
2: IN "init.red"$
3: GENTRANLANG!* := 'FORTRAN$
4: GENTRANIN
4:      "main.tem"
4: OUT "main.f"$

```

Contents of file `main.f`:

```

C
C  MAIN PROGRAM
C
      REAL M(3,3),DET,MINV(3,3)
      INTEGER N
      DATA N/3/
      WRITE(6,*) 'ENTER ', N, 'x', N, ' MATRIX'
      DO 100 I=1,N
          READ(5,*) (M(I,J),J=1,N)
100  CONTINUE
      WRITE(6,*) ' DET = ', DET(M)
      WRITE(6,*) ' INVERSE MATRIX:'
      CALL INV(M,MINV)
      DO 200 I=1,N
          WRITE(6,*) (MINV(I,J),J=1,N)
200  CONTINUE

```

```

200    CONTINUE
      STOP
      END

C
C  DETERMINANT CALCULATION
C
      REAL FUNCTION DET(M)
      REAL M(3,3)
      DET=M(3,3)*M(2,2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)
      . *M(2,3)*M(1,1))+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)
      . *M(1,2)-(M(3,1)*M(2,2)*M(1,3))
      RETURN
      END

C
C  INVERSE CALCULATION
C
      SUBROUTINE INV(M,MINV)
      REAL M(3,3),MINV(3,3)
      MINV(1,1)=(M(3,3)*M(2,2)-(M(3,2)*M(2,3)))/(M(3,3)*M(2,2)
      . *M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))
      . +M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(2
      . ,2)*M(1,3)))
      MINV(1,2)=(-(M(3,3)*M(1,2))+M(3,2)*M(1,3))/(M(3,3)*M(2,
      . 2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))
      . )+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(
      . 2,2)*M(1,3)))
      MINV(1,3)=(M(2,3)*M(1,2)-(M(2,2)*M(1,3)))/(M(3,3)*M(2,2)
      . )*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))
      . +M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(2
      . ,2)*M(1,3)))
      MINV(2,1)=(-(M(3,3)*M(2,1))+M(3,1)*M(2,3))/(M(3,3)*M(2,
      . 2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))
      . )+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(
      . 2,2)*M(1,3)))
      MINV(2,2)=(M(3,3)*M(1,1)-(M(3,1)*M(1,3)))/(M(3,3)*M(2,2)
      . )*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))
      . +M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(2
      . ,2)*M(1,3)))
      MINV(2,3)=(-(M(2,3)*M(1,1))+M(2,1)*M(1,3))/(M(3,3)*M(2,
      . 2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))
      . )+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(
      . 2,2)*M(1,3)))
      MINV(3,1)=(M(3,2)*M(2,1)-(M(3,1)*M(2,2)))/(M(3,3)*M(2,2)
      . )*M(1,1)-(M(3,3)*M(2,1)*M(1,2))-(M(3,2)*M(2,3)*M(1,1))

```

```

. +M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(2,
. ,2)*M(1,3))
MINV(3,2)=(-(M(3,2)*M(1,1))+M(3,1)*M(1,2))/(M(3,3)*M(2,
. 2)*M(1,1)-(M(3,3)*M(2,1)*M(1,2)-(M(3,2)*M(2,3)*M(1,1)
. )+M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(
. 2,2)*M(1,3)))
MINV(3,3)=(M(2,2)*M(1,1)-(M(2,1)*M(1,2)))/(M(3,3)*M(2,2
. )*M(1,1)-(M(3,3)*M(2,1)*M(1,2)-(M(3,2)*M(2,3)*M(1,1)
. +M(3,2)*M(2,1)*M(1,3)+M(3,1)*M(2,3)*M(1,2)-(M(3,1)*M(2
. ,2)*M(1,3)))
RETURN
END

```

This is an example of a modular approach to code generation; separate subprogram templates are given in separate files. Furthermore, the template files are general; they can be used for matrices of any predetermined size. Therefore, we can easily generate different subprograms to handle matrices of different sizes from the same template files simply by assigning different values to n , and reloading the file `init.red`.

Template Processing and Generation of Type Declarations

In Section 16.24.2 we described the **GENDECS** flag. We explained that type declarations are not generated when this flag is turned off. Now that the concept of template processing has been explained, it is appropriate to continue our discussion of generation of type declarations.

When the **GENDECS** flag is off, type declaration information is not simply discarded — it is still maintained in the symbol table. Only the automatic extraction of this information in the form of declarations is disabled. When the **GENDECS** flag is turned off, all type information associated with a specific subprogram can be retrieved in the form of generated declarations by calling the **GENDECS** function with the subprogram name as argument. The template processor recognizes function and subroutine headings. It always keeps track of the name of the subprogram it is processing. Therefore, the declarations associated with a particular subprogram *subprogrname* can be generated with a call to **GENDECS** as follows:

GENDECS *subprogrname*\$

By using the **GENDECS** flag and function together with the template processing facility, it is possible to have type information inserted into the symbol table during a first pass over a template file, and then to have it extracted during a second pass. Consider the following example in which the original template file is transformed into an intermediate template during the first pass, and then into the final file of

FORTTRAN code during the second pass:

Contents of file `junk.tem`:

```

;BEGIN;
MAXEXPPRINTLEN!* := 50$
OFF GENDECS$
;END;
      SUBROUTINE CALC (X,Y,Z,A,B,RES)
;BEGIN;
GENTRAN LITERAL ";BEGIN;", CR!*,
          "GENDECS CALC$", CR!*,
          ";END;", CR!*$
;END;
      X=3.75
      Y=-10.2
      Z=16.473
;BEGIN;
GENTRAN
<<
      DECLARE X,Y,Z,A,B,RES : REAL;
      RES :=: (X + Y + Z)^3*(A + B)^2
>>$
;END;
      RETURN
      END
;BEGIN;
GENTRAN LITERAL ";END;", CR!*$
;END;
;END;

```

Invocation of the template processor on this file produces an intermediate template file:

```

1: GENTRANIN
1:      "junk.tem"
1: OUT "#junk.tem"$

```

Contents of file `#junk.tem`:

```

      SUBROUTINE CALC (X,Y,Z,A,B,RES)
;BEGIN;
GENDECS CALC$
;END;

```

```

X=3.75
Y=-10.2
Z=16.473
T0=A**2*X**3+3.0*A**2*X**2*Y
T0=T0+3.0*A**2*X**2*Z+3.0*A**2*X*Y**2
T0=T0+6.0*A**2*X*Y*Z+3.0*A**2*X*Z**2
T0=T0+A**2*Y**3+3.0*A**2*Y**2*Z
T0=T0+3.0*A**2*Y*Z**2+A**2*Z**3
T0=T0+2.0*A*B*X**3+6.0*A*B*X**2*Y
T0=T0+6.0*A*B*X**2*Z+6.0*A*B*X*Y**2
T0=T0+12.0*A*B*X*Y*Z+6.0*A*B*X*Z**2
T0=T0+2.0*A*B*Y**3+6.0*A*B*Y**2*Z
T0=T0+6.0*A*B*Y*Z**2+2.0*A*B*Z**3
T0=T0+B**2*X**3+3.0*B**2*X**2*Y
T0=T0+3.0*B**2*X**2*Z+3.0*B**2*X*Y**2
T0=T0+6.0*B**2*X*Y*Z+3.0*B**2*X*Z**2
T0=T0+B**2*Y**3+3.0*B**2*Y**2*Z
RES=T0+3.0*B**2*Y*Z**2+B**2*Z**3
RETURN
END
;END;

```

Another pass of the template processor produced the final file of FORTRAN code:

```

2: GENTRANIN
2:      "#junk.tem"
2: OUT  "junk.f"$

```

Contents of file junk.f:

```

SUBROUTINE CALC(X,Y,Z,A,B,RES)
REAL X,Y,Z,A,B,RES,T0
X=3.75
Y=-10.2
Z=16.473
T0=A**2*X**3+3.0*A**2*X**2*Y
T0=T0+3.0*A**2*X**2*Z+3.0*A**2*X*Y**2
T0=T0+6.0*A**2*X*Y*Z+3.0*A**2*X*Z**2
T0=T0+A**2*Y**3+3.0*A**2*Y**2*Z
T0=T0+3.0*A**2*Y*Z**2+A**2*Z**3
T0=T0+2.0*A*B*X**3+6.0*A*B*X**2*Y
T0=T0+6.0*A*B*X**2*Z+6.0*A*B*X*Y**2
T0=T0+12.0*A*B*X*Y*Z+6.0*A*B*X*Z**2
T0=T0+2.0*A*B*Y**3+6.0*A*B*Y**2*Z

```

```

T0=T0+6.0*A*B*Y*Z**2+2.0*A*B*Z**3
T0=T0+B**2*X**3+3.0*B**2*X**2*Y
T0=T0+3.0*B**2*X**2*Z+3.0*B**2*X*Y**2
T0=T0+6.0*B**2*X*Y*Z+3.0*B**2*X*Z**2
T0=T0+B**2*Y**3+3.0*B**2*Y**2*Z
RES=T0+3.0*B**2*Y*Z**2+B**2*Z**3
RETURN
END

```

Referencing Subprogram and Parameter Names

In some code generation applications in which template processing is used, it is useful to be able to reference the names of the parameters given in the subprogram header. For this reason, the special symbols **!\$1**, **!\$2**, ..., **!\$*n***, where *n* is the number of parameters, can be used in computations and code generation commands in active parts of template files. Each of these symbols will be replaced by the corresponding parameter name when code is generated. In addition, the special symbol **!\$0** will be replaced by the subprogram name. This is useful when FORTRAN or RATFOR functions are being generated. Finally, the special global variable **!\$!** is bound to the number of parameters in the subprogram header.

16.24.4 Output Redirection

Many examples given thus far in this manual have sent all generated code to the terminal screen. In actual code generation applications, however, code must be sent to a file which will be compiled at a later time. This section explains methods of redirecting code to a file as it is generated. Any number of output files can be open simultaneously, and generated code can be sent to any combination of these open files.

File Selection Commands

REDUCE provides the user with two file handling commands for output redirection: **OUT** and **SHUT**. The **OUT** command takes a single file name as argument and directs all REDUCE output to that file from then on, until another **OUT** changes the output file, or **SHUT** closes it. Output can go to only one file at a time, although many can be open. If the file has previously been used for output during the current job and not **SHUT**, then the new output is appended onto the end of the file. Any existing file is erased before its first use for output in a job. To output on the terminal without closing the output file, the reserved file name **T** (for terminal) may be used.

The REDUCE **SHUT** command takes a list of names of files which have been previously opened via an **OUT** command and closes them. Most systems require this action by the user before he ends the REDUCE job; otherwise the output may be lost. If a file is **SHUT** and a further **OUT** command is issued for the same file, the file is erased before the new output is written. If it is the current output file that is **SHUT**, output will switch to the terminal.

These commands are suitable for most applications in which REDUCE output must be saved. However, they have two deficiencies when considered for use in code generation applications. First, they are inconvenient. **OUT** tells REDUCE to direct *all* output to a specified file. Thus in addition to output written as side effects of functions, returned values are also written to the file (unless the user is careful to terminate all statements and commands with a \$, in which case only output produced by side effects is written). If code generation is to be accomplished interactively; i.e., if algebraic computations and code generation commands are interleaved, then **OUT filename\$** must be issued before every group of code generation requests, and **OUT T\$** must be issued after every group. Secondly, the **OUT** command does not allow output to be sent to two or more files without reissuing the **OUT** with another file name. In an effort to remove these deficiencies and make the code generation commands flexible and easy to use, separate file handling commands are provided by GENTRAN which redirect generated code *only*.

The **GENTRANOUT** and **GENTRANSHUT** commands are identical to the REDUCE **OUT** and **SHUT** commands with the following exceptions:

- **GENTRANOUT** and **GENTRANSHUT** redirect *only* code which is printed as a side effect of GENTRAN commands.
- **GENTRANOUT** allows more than one file name to be given to indicate that generated code is to be sent to two or more files. (It is particularly convenient to be able to have generated code sent to the terminal screen and one or more file simultaneously.)
- **GENTRANOUT** does not automatically erase existing files; it prints a warning message on the terminal and asks the user whether the existing file should be erased or the whole command be aborted.

The next two subsections describe these commands in detail.

GENTRANOUT Syntax:

GENTRANOUT *f1,f2,...,fn;*

Arguments:

$f1, f2, \dots, fn$ is a list of one or more f 's, where each f is one of:

<i>an atom</i>	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN

Side Effects:

GENTRAN maintains a list of files currently open for output by GENTRAN *only*. **GENTRANOUT** inserts each file name represented by $f1, f2, \dots, fn$ into that list and opens each one for output. It also resets the current output file(s) to be all files in $f1, f2, \dots, fn$.

Returned Value:

GENTRANOUT returns the list of files represented by $f1, f2, \dots, fn$; i.e., the current output file(s) after the command has been executed.

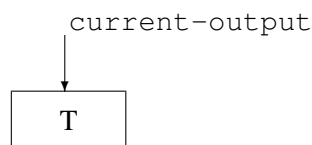
Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS
    OVERWRITE FILE? (Y/N)
```

```
***** WRONG TYPE OF ARG
```

Example 38

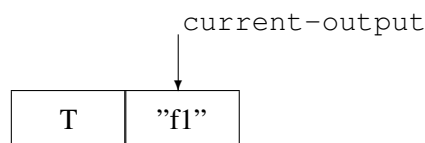
Output file list:



```
1: GENTRANOUT "f1";
```

```
"f1"
```

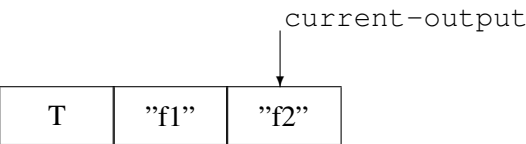
Output file list:



2: GENTRANOUT "f2";

"f2"

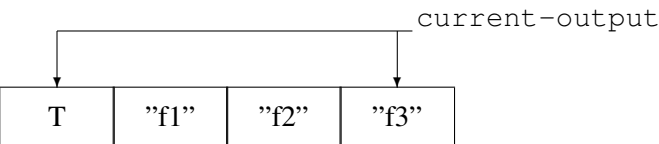
Output file list:



3: GENTRANOUT T, "f3";

{T, "f3"}

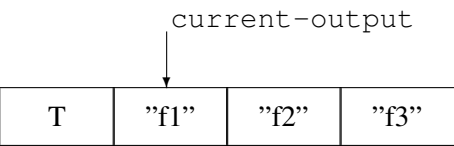
Output file list:



4: GENTRANOUT "f1";

"f1"

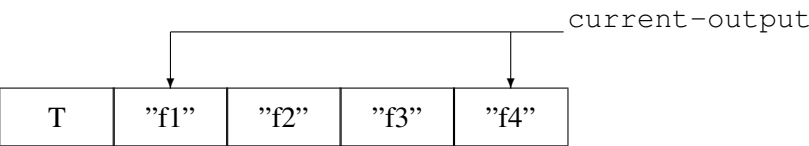
Output file list:



5: GENTRANOUT NIL, "f4";

{"f1", "f4"}

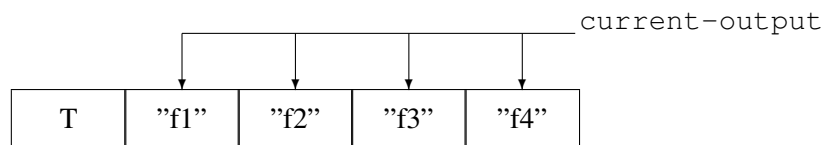
Output file list:



```
6: GENTRANOUT ALL!*;

{"f1", "f2", "f3", "f4"}
```

Output file list:



GENTRANSHUT Syntax:

```
GENTRANSHUT f1,f2,... ,fn;
```

Arguments:

f1,*f2*,... ,*fn* is a list of one or more *f*'s, where each *f* is one of:

<i>an atom</i>	=	an output file
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN

Side Effects:

GENTRANSHUT creates a list of file names from *f1*,*f2*,... ,*fn*, deletes each from the output file list, and closes the corresponding files. If (all of) the current output file(s) are closed, then the current output file is reset to the terminal.

Returned Value:

GENTRANSHUT returns the current output file(s) after the command has been executed.

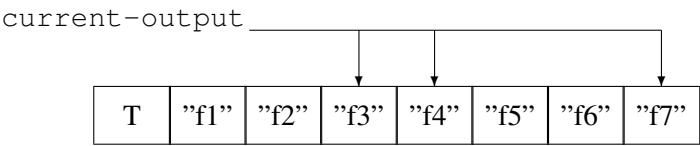
Diagnostic Messages:

```
*** FILE NOT OPEN FOR OUTPUT

***** WRONG TYPE OF ARG
```

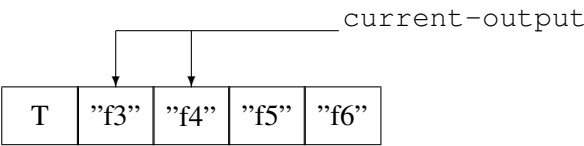
Example 39

Output file list:



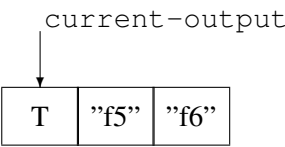
```
1: GENTRANSHUT "f1", "f2", "f7";  
  
{"f3", "f4"}
```

Output file list:



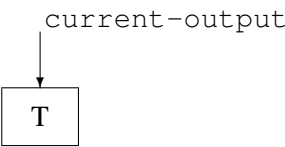
```
2: GENTRANSHUT NIL;  
  
T
```

Output file list:



```
3: GENTRANSHUT ALL!*;  
  
T
```

Output file list:



The Output File Stack

Section 16.24.4 explained the **GENTRANOUT** and **GENTRANSHUT** commands which are very similar to the **REDUCE OUT** and **SHUT** commands but redirect *only code generated as side effects of GENTRAN commands* to files. This section describes another pair of file handling commands provided by GENTRAN.

In some code generation applications it may be convenient to be able to send generated code to one (set of) file(s), then temporarily send code to another (set of) file(s), and later resume sending generated code to the first (set of) file(s). In other words, it is convenient to think of the output files as being arranged in a stack which can be pushed whenever new files are to be written to temporarily, and popped whenever previously written-to files are to be appended onto. **GENTRANPUSH** and **GENTRANPOP** enable the user to manipulate a stack of open output files in these ways.

GENTRANPUSH simply pushes a (set of) file(s) onto the stack and opens each one that is not already open for output. **GENTRANPOP** deletes the top-most occurrence of the given file(s) from the stack and closes each one that is no longer in the stack. The stack is initialized to one element: the terminal. This element is always on the bottom of the stack, and thus, is the default output file. The current output file is always the file(s) on top of the stack.

GENTRANPUSH Syntax:

GENTRANPUSH *f1,f2,...,fn*;

Arguments:

f1,f2,...,fn is a list of one or more *f*'s, where each *f* is one of:

<i>an atom</i>	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN

Side Effects:

GENTRANPUSH creates a list of file name(s) represented by *f1,f2,...,fn* and pushes that list onto the output stack. Each file in the list that is not already open for output is opened at this time. The current output file is reset to this new element on the top of the stack.

Returned Value:

GENTRANPUSH returns the list of files represented by $f1, f2, \dots, fn$;
i.e., the current output file(s) after the command has been executed.

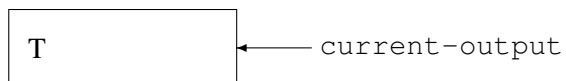
Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS
    OVERWRITE FILE? (Y/N)
```

```
***** WRONG TYPE OF ARG
```

Example 40

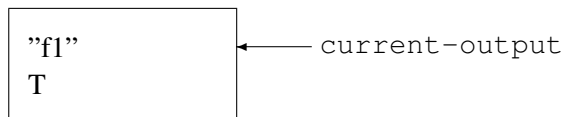
Output stack:



```
1: GENTRANPUSH "f1";
```

```
"f1"
```

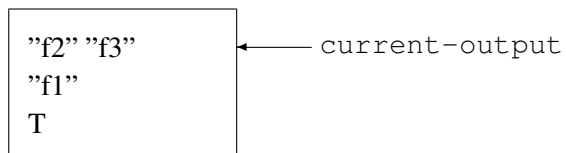
Output stack:



```
2: GENTRANPUSH "f2", "f3";
```

```
{"f2", "f3"}
```

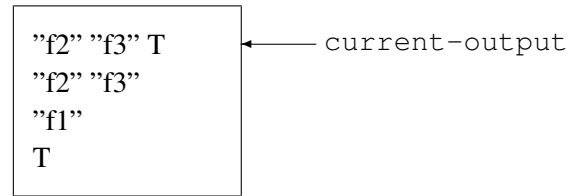
Output stack:



```
3: GENTRANPUSH NIL, T;
```

```
{"f2", "f3", T}
```

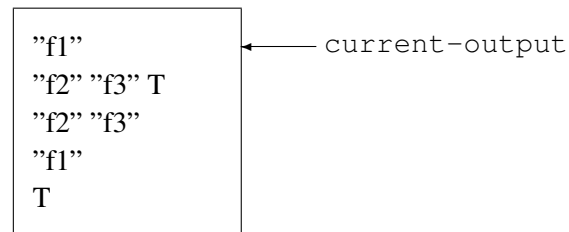
Output stack:



```
4: GENTRANPUSH "f1";
```

```
"f1"
```

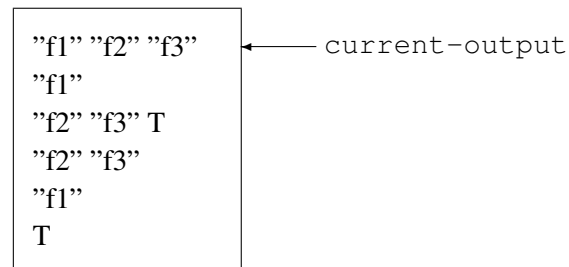
Output stack:



```
5: GENTRANPUSH ALL!*;
```

```
{"f1", "f2", "f3"}
```

Output stack:



GENTRANPOP Syntax:

```
GENTRANPOP f1,f2,... ,fn;
```

Arguments:

f1,*f2*,... ,*fn* is a list of one or more *f*'s, where each *f* is one of:

an atom = an output file
T = the terminal
NIL = the current output file(s)
ALL!* = all files currently open for output
 by GENTRAN

Side Effects:

GENTRANPOP deletes the top-most occurrence of the single element containing the file name(s) represented by f_1, f_2, \dots, f_n from the output stack. Files whose names have been completely removed from the output stack are closed. The current output file is reset to the (new) element on the top of the output stack.

Returned Value:

GENTRANPOP returns the current output file(s) after this command has been executed.

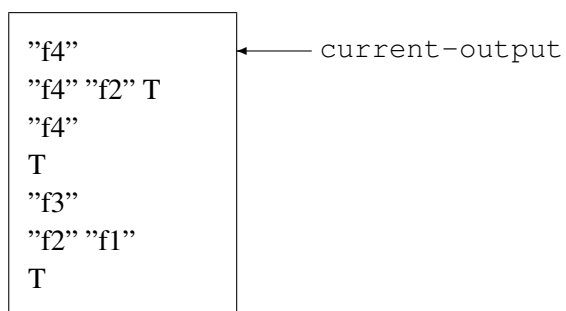
Diagnostic Messages:

*** FILE NOT OPEN FOR OUTPUT

***** WRONG TYPE OF ARG

Example 41

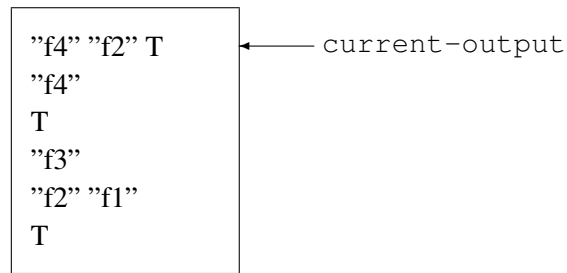
Output stack:



```
1: GENTRANPOP NIL;
```

```
{ "f4", "f2", T }
```

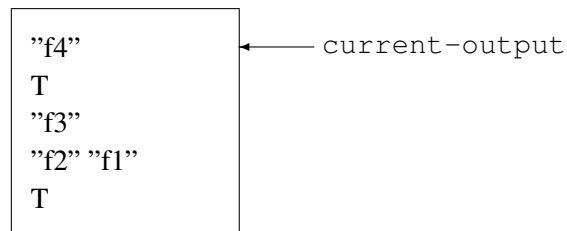
Output stack:



```
2: GENTRANPOP NIL;
```

```
"f4"
```

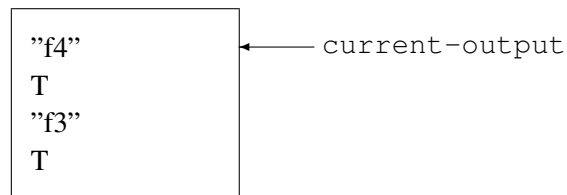
Output stack:



```
3: GENTRANPOP "f2", "f1";
```

```
"f4"
```

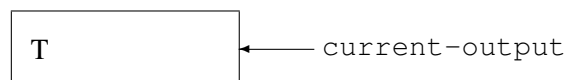
Output stack:



```
4: GENTRANPOP ALL!*;
```

```
T
```

Output stack:



Temporary Output Redirection

Sections 16.24.2 and 16.24.3 explain how to use the code generation and template processing commands. The syntax for these two commands is:

```
GENTRAN stmt [OUT f1,f2,...,fn];
and
GENTRANIN f1,f2,...,fm [OUT f1,f2,...,fn];
```

The optional parts of these two commands can be used for *temporary* output redirection; they can be used when the current output file is to be temporarily reset, for this command only.

Thus the following two sequences of commands are equivalent:

```
10: GENTRANPUSH "f1",T$
11: GENTRAN ... $
12: GENTRANPOP NIL$
```

and

```
10: GENTRAN
10: ...
10: OUT "f1",T$
```

16.24.5 Modification of the Code Generation Process

GENTRAN is designed to be flexible enough to be used in a variety of code generation applications. For this reason, several mode switches and variables are provided to enable the user to tailor the code generation process to meet his or her particular needs.

Mode Switches

The following GENTRAN mode switches can be turned on and off with the REDUCE **ON** and **OFF** commands.

DOUBLE

- When turned on, causes (where appropriate):
 - floating point numbers to be printed in double precision format;

- intrinsic functions to be replaced by their double precision counterparts;
- generated type declarations to be of double precision form.

See also section [16.24.2](#) on page [331](#).

- default setting: off

GENDECS

- when turned on, allows type declarations to be generated automatically; otherwise, type information is stored in but not automatically retrieved from the symbol table. See also sections [16.24.2](#) on page [336](#), [16.24.2](#) on page [341](#), and [16.24.3](#) on page [361](#).
- default setting: on

GENTRANOPT

- when turned on, replaces each block of straightline code by an optimized sequence of assignments. The Code Optimizer takes a sequence of assignments and replaces common subexpressions with temporary variables. It returns the resulting assignment statements with common-subexpression-to-temporary-variable assignment statements preceding them
- default setting: off

GENTRANSEG

- when turned on, checks the print length of expressions and breaks those expressions that are longer than **MAXEXPPRINTLEN!*** down into subexpressions which are assigned to temporary variables. See also section [16.24.2](#) on page [343](#).
- default setting: on

GETDECS

- when on, causes:
 - the indices of loops to be declared integer;
 - objects without an explicit type declaration to be declared of the type given by the variable **DEFTYPE!***.

See also section [16.24.2](#) on page [340](#).

- default setting: off

KEEPDECS

- when on, prevents declarations being removed from the symbol table when type declarations are generated.
- default: off

MAKECALLS

- when turned on, causes GENTRAN to translate functional expressions as subprogram calls.
- default setting: on

PERIOD

- when turned on, causes all integers to be printed out as floating point numbers except:
 - exponents;
 - variable subscripts;
 - index values in DO-type loops;
 - those which have been declared to be integers.
- default setting: on

Variables

Several global variables are provided in GENTRAN to enable the user to

- select the target language
- control expression segmentation
- change automatically generated variable names and statement numbers
- modify the code formatter

The following four subsections describe these variables⁸.

Target Language Selection GENTRANLANG!*

- target language (FORTRAN, RATFOR, PASCAL or C) See also section [16.24.2](#) on page [328](#).
- value type: atom
- default value: FORTRAN

⁸ Note that when an atomic value (other than an integer) is assigned to a variable, that value must be quoted. For example, **GENTRANLANG!* := 'FORTRAN\$** assigns the atom **FORTRAN** to the variable **GENTRANLANG!***.

Expression Segmentation Control MAXEXPPRINTLEN!*

- value used to determine whether or not an expression should be segmented; maximum number of characters permitted in an expression in the target language (excluding spaces printed for formatting). See also section [16.24.2](#) on page [343](#).
- value type: integer
- default value: 800

Variable Names & Statement Numbers TEMPVARNAME!*

- name used as prefix in generating temporary variable names. See also section [16.24.2](#) on page [345](#).
- value type: atom
- default value: T

TEMPVARNUM!*

- number appended to **TEMPVARNAME!*** to create a temporary variable name. If the temporary variable name resulting from appending **TEMPVARNUM!*** onto **TEMPVARNAME!*** has already been generated and still holds a useful value, then **TEMPVARNUM!*** is incremented and temporary variable names are compressed until one is found which was not previously generated or does not still hold a significant value. See also section [16.24.2](#) on page [345](#).
- value type: integer
- default value: 0

TEMPVARTYPE!*

- target language variable type (e.g., INTEGER, REAL!*8, FLOAT, etc) used as a default for automatically generated variables whose type cannot be determined otherwise. If **TEMPVARTYPE!*** is NIL, then generated temporary variables whose type cannot be determined are not automatically declared. See also section [16.24.2](#) on page [345](#).
- value type: atom
- default value: NIL

GENSTMTNUM!*

- number used when a statement number must be generated
- value type: integer
- default value: 25000

GENSTMTINCR!*

- number by which **GENSTMTNUM!*** is increased each time a new statement number is generated.
- value type: integer
- default value: 1

DEFTYPE!*

- default type for objects when the switch **GETDECS** is on. See also section [16.24.2](#) on page [340](#).
- value type: atom
- default value: real

Code Formatting FORTCURRIND!*

- number of blank spaces printed at the beginning of each line of generated FORTRAN code beyond column 6
- value type: integer
- default value: 0

RATCURRIND!*

- number of blank spaces printed at the beginning of each line of generated RATFOR code.
- value type: integer
- default value: 0

CCURRIND!*

- number of blank spaces printed at the beginning of each line of generated C code.
- value type: integer
- default value: 0

PASCCURRIND!*

- number of blank spaces printed at the beginning of each line of generated PASCAL code.
- value type: integer
- default value: 0

TABLEN!*

- number of blank spaces printed for each new level of indentation.
- value type: integer
- default value: 4

FORTLINELEN!*

- maximum number of characters printed on each line of generated FORTRAN code.
- value type: integer
- default value: 72

RATLINELEN!*

- maximum number of characters printed on each line of generated RATFOR code.
- value type: integer
- default value: 80

CLINELEN!*

- maximum number of characters printed on each line of generated C code.
- value type: integer
- default value: 80

PASCLINELEN!*

- maximum number of characters printed on each line of generated PASCAL code.
- value type: integer
- default value: 70

MINFORTLINELEN!*

- minimum number of characters printed on each line of generated FORTRAN code after indentation.
- value type: integer
- default value: 40

MINRATLINELEN!*

- minimum number of characters printed on each line of generated RATFOR code after indentation.
- value type: integer
- default value: 40

MINCLINELEN!*

- minimum number of characters printed on each line of generated C code after indentation.
- value type: integer
- default value: 40

MINPASCLINELEN!*

- minimum number of characters printed on each line of generated PASCAL code after indentation.
- value type: integer
- default value: 40

16.24.6 Examples

Short examples have been given throughout this manual to illustrate usage of the GENTRAN commands. This section gives complete code generation examples.

Interactive Code Generation

Suppose we wish to generate a FORTRAN subprogram which can be used for computing the roots of a polynomial by Graeffe's Root-Squaring Method⁹. This method states that the roots x_i of a polynomial

$$P_n(x) = \sum_{i=0}^n a_i x^{n-i}$$

⁹ This is for instance convenient for ill-conditioned polynomials. More details are given in *Introduction to Numerical Analysis* by C. E. Froberg, Addison-Wesley Publishing Company, 1966.

can be found by constructing the polynomial

$$P_n^*(x^2) = (a_0x^n + a_2x^{n-2} + \dots)^2 - (a_1x^{n-1} + a_3x^{n-3} + \dots)^2$$

with roots x_i^2 . When read into REDUCE, the following file of REDUCE statements will place the coefficients of P_n^* into the list B for some user-entered value of n greater than zero.

Contents of file `graeffe.red`:¹⁰

```

OPERATOR A$
Q := FOR I := 0 STEP 2 UNTIL n    SUM (A(I+1)*X^(n-I))$
R := FOR I := 1 STEP 2 UNTIL n-1  SUM (A(I+1)*X^(n-I))$
P := Q^2 - R^2$
LET X^2 = Y$
B := COEFF(P,Y)$
END$

```

Now a numerical subprogram can be generated with assignment statements for the coefficients of P_n^* (now stored in list B in REDUCE). Since these coefficients are given in terms of the coefficients of P_n (i.e., operator A in REDUCE), the subprogram will need two parameters: A and B, each of which must be arrays of size n+1.

The following REDUCE session will create subroutine GRAEFF for a polynomial of degree n=10 and write it to file `graeffe.f`:

```

1: n := 10$

2: IN "graeffe.red"$

3: GENTRANLANG!* := 'FORTRAN$

4: ON DOUBLE$

5: GENTRAN
5: (
5:     PROCEDURE GRAEFF(A,B);
5:     BEGIN
5:     DECLARE
5:     <<
5:         GRAEFF : SUBROUTINE;
5:         A(11),B(11) : REAL
5:     >>;
5:     LITERAL
5:         "C",CR!*,
5:         "C",TAB!*, "GRAEFFE ROOT-SQUARING METHOD TO FIND",CR!*,

```

¹⁰ In accordance with section 16.24.2, the subscripts of A are I+1 instead of I.


```

5:      "C",TAB!*, "ROOTS OF A POLYNOMIAL",CR!*,
5:      "C",CR!*;
5:      B(1) :=: PART (B,1);
5:      B(2) :=: PART (B,2);
5:      B(3) :=: PART (B,3);
5:      B(4) :=: PART (B,4);
5:      B(5) :=: PART (B,5);
5:      B(6) :=: PART (B,6);
5:      B(7) :=: PART (B,7);
5:      B(8) :=: PART (B,8);
5:      B(9) :=: PART (B,9);
5:      B(10) :=: PART (B,10);
5:      B(11) :=: PART (B,11)
5:      END
5: )
5: OUT "graeffe.f"$

```

Contents of file graeffe.f:

```

      SUBROUTINE GRAEFF(A,B)
      DOUBLE PRECISION A(11),B(11)
C
C      GRAEFFE ROOT-SQUARING METHOD TO FIND
C      ROOTS OF A POLYNOMIAL
C
      B(1)=A(11)**2
      B(2)=2.0D0*A(11)*A(9)-A(10)**2
      B(3)=2.0D0*A(11)*A(7)-(2.0D0*A(10)*A(8))+A(9)**2
      B(4)=2.0D0*A(11)*A(5)-(2.0D0*A(10)*A(6))+2.0D0*A(9)*A(7)
      . )-A(8)**2
      B(5)=2.0D0*A(11)*A(3)-(2.0D0*A(10)*A(4))+2.0D0*A(9)*A(5)
      . )-(2.0D0*A(8)*A(6))+A(7)**2
      B(6)=2.0D0*A(11)*A(1)-(2.0D0*A(10)*A(2))+2.0D0*A(9)*A(3)
      . )-(2.0D0*A(8)*A(4))+2.0D0*A(7)*A(5)-A(6)**2
      B(7)=2.0D0*A(9)*A(1)-(2.0D0*A(8)*A(2))+2.0D0*A(7)*A(3)-
      . (2.0D0*A(6)*A(4))+A(5)**2
      B(8)=2.0D0*A(7)*A(1)-(2.0D0*A(6)*A(2))+2.0D0*A(5)*A(3)-
      . A(4)**2
      B(9)=2.0D0*A(5)*A(1)-(2.0D0*A(4)*A(2))+A(3)**2
      B(10)=2.0D0*A(3)*A(1)-A(2)**2
      B(11)=A(1)**2
      RETURN
      END

```

Code Generation, Segmentation & Temporary Variables

The following 3 x 3 inertia matrix M was derived in the course of some research ¹¹:

$$\begin{aligned}
 M(1,1) &= 18 * \cos(q_3) * \cos(q_2) * m_{30} * p^2 - \sin^2(q_3) * j_{30}y + \sin^2(q_3) \\
 &\quad * j_{30}z - 9 * \sin^2(q_3) * m_{30} * p^2 + j_{10}y + j_{30}y + m_{10} * p^2 + \\
 &\quad 18 * m_{30} * p^2 \\
 M(1,2) &= 9 * \cos(q_3) * \cos(q_2) * m_{30} * p^2 - \sin^2(q_3) * j_{30}y + \sin^2(q_3) \\
 &\quad * j_{30}z - 9 * \sin^2(q_3) * m_{30} * p^2 + j_{30}y + 9 * m_{30} * p^2 \\
 M(2,1) &= M(1,2) \\
 M(1,3) &= -9 * \sin(q_3) * \sin(q_2) * m_{30} * p^2 \\
 M(3,1) &= M(1,3) \\
 M(2,2) &= -\sin^2(q_3) * j_{30}y + \sin^2(q_3) * j_{30}z - 9 * \sin^2(q_3) * m_{30} * p^2 \\
 &\quad + j_{30}y + 9 * m_{30} * p^2 \\
 M(2,3) &= 0 \\
 M(3,2) &= M(2,3) \\
 M(3,3) &= 9 * m_{30} * p^2 + j_{30}x
 \end{aligned}$$

We know M is symmetric. We wish to generate numerical code to compute values for M and its inverse matrix.

Code Generation Generating code for matrix M and its inverse matrix is straightforward. We can simply generate an assignment statement for each element of M, compute the inverse matrix MIV, and generate an assignment statement for each element of MIV. Since we know M is symmetric, we know that MIV will also be symmetric. To avoid duplicate computations, we will not generate assignments for elements below the main diagonals of these matrices. Instead, we will copy elements across the main diagonal by generating nested loops. The following REDUCE session will write to the file m1.f:

```

1: IN "m.red"$ % Initialize M

2: GENTRANOUT "m1.f"$

3: GENTRANLANG!* := 'FORTRAN$

```

¹¹For details see: Bos, A. M. and M. J. L. Tiernego. "Formula Manipulation in the Bond Graph Modelling and Simulation of Large Mechanical Systems", *Journal of the Franklin Institute*, Pergamon Press Ltd., Vol. 319, No. 1/2, pp. 51-65, January/February 1985.

```

4: ON DOUBLE$

5: FOR J := 1 : 3 DO
5:     FOR K := J : 3 DO
5:         GENTRAN M(J,K) ::=: M(J,K) $

6: MIV := M^(-1) $

7: FOR J := 1 : 3 DO
7:     FOR K := J : 3 DO
7:         GENTRAN MIV(J,K) ::=: MIV(J,K) $

8: GENTRAN
8:     FOR J := 1 : 3 DO
8:         FOR K := J+1 : 3 DO
8:             <<
8:                 M(K,J) := M(J,K) ;
8:                 MIV(K,J) := MIV(J,K)
8:             >>$

9: GENTRANSHUT "m1.f"$

```

The contents of `m1.f` are reproduced in [16.24.10](#) on page 426.

This code was generated with the segmentation facility turned off. However, most FORTRAN compilers cannot handle statements more than 20 lines long. The next section shows how to generate segmented assignments.

Segmentation Large arithmetic expressions can be broken into pieces of manageable size with the expression segmentation facility. The following REDUCE session will write segmented assignment statements to the file `m2.f`. Large arithmetic expressions will be broken into subexpressions of approximately 300 characters in length.

```

1: IN "m.red"$ % Initialize M

2: GENTRANOUT "m2.f"$

3: ON DOUBLE$

4: ON GENTRANSEG$

5: MAXEXPPRINTLEN!* := 300$

```

```

6: FOR J := 1 : 3 DO
6:     FOR K := J : 3 DO
6:         GENTRAN M(J,K) ::=: M(J,K) $

7: MIV := M^(-1) $

8: FOR J := 1 : 3 DO
8:     FOR K := J : 3 DO
8:         GENTRAN MIV(J,K) ::=: MIV(J,K) $

9: GENTRAN
9:     FOR J := 1 : 3 DO
9:         FOR K := J+1 : 3 DO
9:             <<
9:                 M(K,J) := M(J,K) ;
9:                 MIV(K,J) := MIV(J,K)
9:             >>$

10: GENTRANSHUT "m2.f"$

```

The contents of file `m2.f` are reproduced in [16.24.10](#) on page [426](#).

Generation of Temporary Variables to Suppress Simplification We can dramatically improve the efficiency of the code generated in sections [16.24.6](#) on page [384](#) and [16.24.6](#) on page [385](#) by replacing expressions by temporary variables before computing the inverse matrix. This effectively suppresses simplification; these expressions will not be substituted into later computations. We will replace each non-zero element of the REDUCE matrix `M` by a generated variable name, and generate a numerical assignment statement to reflect that substitution in the numerical program being generated.

The following REDUCE session will write to the file `m3.f`:

```

1: in "m.red"$ % Initialize M

2: GENTRANOUT "m3.f"$

3: GENTRANLANG!* := 'FORTRAN$

4: ON DOUBLE$

5: FOR J := 1 : 3 DO

```

```

5:      FOR K := J : 3 DO
5:          GENTRAN M(J,K) ::= M(J,K) $

6: SHARE VAR$

7: FOR J := 1 : 3 DO
7:     FOR K := J : 3 DO
7:         IF M(J,K) NEQ 0 THEN
7:             <<
7:                 VAR := TEMPVAR(NIL) $
7:                 MARKVAR VAR$
7:                 M(J,K) := VAR$
7:                 M(K,J) := VAR$
7:                 GENTRAN
7:                     EVAL(VAR) := M(EVAL(J), EVAL(K))
7:             >>$

8: COMMENT ** Contents of matrix M: **$

9: M;

[T0  T1  T2]
[      ]
[T1  T3  0 ]
[      ]
[T2  0   T4]

10: MIV := M^(-1) $

11: FOR J := 1 : 3 DO
11:     FOR K := J : 3 DO
11:         GENTRAN MIV(J,K) ::= MIV(J,K) $

12: GENTRAN
12:     FOR J := 1 : 3 DO
12:         FOR K := J+1 : 3 DO
12:             <<
12:                 M(K,J) := M(J,K) ;
12:                 MIV(K,J) := MIV(J,K)
12:             >>$

13: GENTRANSHUT "m3.f"$

```

Contents of file m3.f:

```

      M(1,1)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(
      . Q3))**2*Y*J30)+DSIN(DBLE(Q3))**2*J30Z+18.0D0*DCOS(DBLE
      . (Q3))*DCOS(DBLE(Q2))*P**2*M30+18.0D0*P**2*M30+P**2*M10
      . +J30Y+J10Y
      M(1,2)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(
      . Q3))**2*J30Y)+DSIN(DBLE(Q3))**2*J30Z+9.0D0*DCOS(DBLE(
      . Q3))*DCOS(DBLE(Q2))*P**2*M30+9.0D0*P**2*M30+J30Y
      M(1,3)=- (9.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**2*M30)
      M(2,2)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(
      . Q3))**2*J30Y)+DSIN(DBLE(Q3))**2*J30Z+9.0D0*P**2*M30+
      . J30Y
      M(2,3)=0.0D0
      M(3,3)=9.0D0*P**2*M30+J30X
      T0=M(1,1)
      T1=M(1,2)
      T2=M(1,3)
      T3=M(2,2)
      T4=M(3,3)
      MIV(1,1)=- (T4*T3) / (T4*T1**2- (T4*T3*T0)+T2**2*T3)
      MIV(1,2)= (T4*T1) / (T4*T1**2- (T4*T3*T0)+T2**2*T3)
      MIV(1,3)= (T2*T3) / (T4*T1**2- (T4*T3*T0)+T2**2*T3)
      MIV(2,2)= (- (T4*T0)+T2**2) / (T4*T1**2- (T4*T3*T0)+T2**2*
      . T3)
      MIV(2,3)= - (T1*T2) / (T4*T1**2- (T4*T3*T0)+T2**2*T3)
      MIV(3,3)= (T1**2- (T3*T0)) / (T4*T1**2- (T4*T3*T0)+T2**2*T3)
      DO 25009 J=1,3
        DO 25010 K=J+1,3
          M(K,J)=M(J,K)
          MIV(K,J)=MIV(J,K)
25010      CONTINUE
25009 CONTINUE

```

Template Processing

Circuit simulation plays a vital role in computer hardware development. A recent paper¹² describes the design of an Automatic Circuitry Code Generator (ACCG), which generates circuit simulation programs based on user-supplied circuit specifications. The actual code generator consists of a series of REDUCE **WRITE** statements, each of which writes one line of FORTRAN code.

¹²Loe, K. F., N. Ohsawa, and E. Goto. "Design of an Automatic Circuitry Code Generator (ACCG)", *RSYMSAC Proceedings*, Wako-shi, Saitama, Japan. 1984.

This section presents an alternative implementation for the ACCG which uses GENTRAN's template processor to generate code. Template processing is a much more natural method of code generation than the REDUCE **WRITE** statement method.

First we will put all REDUCE calculations into two files: `rk.red` and `ham.red`.

Contents of file `rk.red`:¹³

```
COMMENT  -- RUNGE-KUTTA METHOD --$
PROCEDURE RUNGEKUTTA(P1, P2, P, Q, TT);
BEGIN
  SCALAR K11,K12,K21,K22,K31,K32,K41,K42;
  K11 := HH*P1;
  K12 := HH*P2;
  K21 := HH*SUB(TT=TT+HH/2, P=P+K11/2, Q=Q+K12/2, P1);
  K22 := HH*SUB(TT=TT+HH/2, P=P+K11/2, Q=Q+K12/2, P2);
  K31 := HH*SUB(TT=TT+HH/2, P=P+K21/2, Q=Q+K22/2, P1);
  K32 := HH*SUB(TT=TT+HH/2, P=P+K21/2, Q=Q+K22/2, P2);
  K41 := HH*SUB(TT=TT+HH, P=P+K31, Q=Q+K32, P1);
  K42 := HH*SUB(TT=TT+HH, P=P+K31, Q=Q+K32, P2);
  PN := P + (K11 + 2*K21 + 2*K31 + K41)/6;
  QN := Q + (K12 + 2*K22 + 2*K32 + K42)/6
END$
END$
```

Contents of file `ham.red`:

```
COMMENT  -- HAMILTONIAN CALCULATION --$
DIFQ := DF(H,P)$
DIFP := -DF(H,Q) - SUB(QDOT=P/M, DF(D,QDOT))$
RUNGEKUTTA(DIFP, DIFQ, P, Q, TT)$
END$
```

Next we will create a template file with an outline of the target FORTRAN program and GENTRAN commands.

Contents of file `runge.tem`:

¹³ Line 11 of procedure RUNGEKUTTA was changed from

```
K41 := HH*SUB(TT=TT+HH, P=P+K31, Q=Q+K32, P2);
```

as given in (Loe84), to

```
K42 := HH*SUB(TT=TT+HH, P=P+K31, Q=Q+K32, P2);
```

```

      PROGRAM RUNGE
      IMPLICIT DOUBLE PRECISION (K,M)
C
C  INPUT
C
      WRITE(6,*) ' INITIAL VALUE OF P '
      READ(5,*) P
      WRITE(6,*) ' P = ', P
      WRITE(6,*) ' INITIAL VALUE OF Q '
      READ(5,*) Q
      WRITE(6,*) ' Q = ', Q
      WRITE(6,*) ' VALUE OF M '
      READ(5,*) M
      WRITE(6,*) ' M = ', M
      WRITE(6,*) ' VALUE OF K0 '
      READ(5,*) K0
      WRITE(6,*) ' K0 = ', K0
      WRITE(6,*) ' VALUE OF B '
      READ(5,*) B
      WRITE(6,*) ' B = ', B
      WRITE(6,*) ' STEP SIZE OF T '
      READ(5,*) HH
      WRITE(6,*) ' STEP SIZE OF T = ', HH
      WRITE(6,*) ' FINAL VALUE OF T '
      READ(5,*) TP
      WRITE(6,*) ' FINAL VALUE OF T = ', TP
C
C  INITIALIZATION
C
      TT=0.0D0
;BEGIN;
      GENTRAN
      LITERAL
      TAB!*, "WRITE(9,*) ' H = ", EVAL(H), "' ", CR!*,
      TAB!*, "WRITE(9,*) ' D = ", EVAL(D), "' ", CR!*$
;END;
      WRITE(9,901) C
901  FORMAT(' C= ',D20.10)
      WRITE(9,910) TT, Q, P
910  FORMAT(' '3D20.10)
C
C  LOOP
C
;BEGIN;

```



```

GENTRAN
  REPEAT
    <<
      PN :=: PN;
      Q  :=: QN;
      P  := PN;
      TT := TT + HH;
      LITERAL
        TAB!*, "WRITE(9,910) TT, QQ, P", CR!*
    >>
  UNTIL TT >= TF$
;END;
  STOP
END
;END;

```

Now we can generate a circuit simulation program simply by starting a REDUCE session and following three steps:

1. Enter circuit specifications.
2. Perform calculations.
3. Call the GENTRAN template processor.

For example, the following REDUCE session will write a simulation program to the file `runge.f`:

```

1: COMMENT  -- INPUT --$

2: K := 1/(2*M)*P^2$      % kinetic energy

3: U := K0/2*Q^2$        % potential energy

4: D := B/2*QDOT$        % dissipating function

5: H := K + U$           % hamiltonian

6: COMMENT  -- CALCULATIONS --$

7: IN "rk.red", "ham.red"$

8: COMMENT  -- FORTRAN CODE GENERATION --$

```

```

9: GENTRANLANG!* := 'FORTRAN$
10: ON DOUBLE$
11: GENTRANIN "runge.tem" OUT "runge.f"$

```

Contents of file runge.f:

```

      PROGRAM RUNGE
      IMPLICIT DOUBLE PRECISION (K,M)
C
C  INPUT
C
      WRITE(6,*) 'INITIAL VALUE OF P'
      READ(5,*) P
      WRITE(6,*) ' P = ', P
      WRITE(6,*) 'INITIAL VALUE OF Q'
      READ(5,*) Q
      WRITE(6,*) ' Q = ', Q
      WRITE(6,*) 'VALUE OF M'
      READ(5,*) M
      WRITE(6,*) ' M = ', M
      WRITE(6,*) 'VALUE OF K0'
      READ(5,*) K0
      WRITE(6,*) ' K0 = ', K0
      WRITE(6,*) 'VALUE OF B'
      READ(5,*) B
      WRITE(6,*) ' B = ', B
      WRITE(6,*) 'STEP SIZE OF T'
      READ(5,*) HH
      WRITE(6,*) ' STEP SIZE OF T = ', HH
      WRITE(6,*) 'FINAL VALUE OF T'
      READ(5,*) TP
      WRITE(6,*) ' FINAL VALUE OF T = ', TP
C
C  INITIALIZATION
C
      TT=0.0D0
      WRITE(9,*) ' H = (M*Q**2*K0+P**2)/(2.0D0*M)'
      WRITE(9,*) ' D = (B*QDOT)/2.0D0'
      WRITE(9,901) C
901  FORMAT(' C= ',D20.10)
      WRITE(9,910) TT, Q, P
910  FORMAT(' '3D20.10)

```

```

C
C  LOOP
C
25001 CONTINUE
      PN= (- (12.0D0*B*M**2*HH)+2.0D0*B*M*K0*HH**3+24.0D0*
.      M**2*P- (24.0D0*M**2*Q*K0*HH)- (12.0D0*M*P*K0*HH**2)
.      +4.0D0*M*Q*K0**2*HH**3+P*K0**2*HH**4) / (24.0D0*M**2
.      )
      Q= (- (12.0D0*B*M*HH**2)+B*K0*HH**4+48.0D0*M**2*Q+
.      48.0D0*M*P*HH- (24.0D0*M*Q*K0*HH**2)- (8.0D0*P*K0*HH
.      **3)+2.0D0*Q*K0**2*HH**4) / (48.0D0*M**2)
      P=PN
      TT=TT+HH
      WRITE(9,910) TT, QQ, P
      IF (.NOT.TT.GE.TF) GOTO 25001
      STOP
      END

```

16.24.7 Symbolic Mode Functions

Thus far in this manual, commands have been presented which are meant to be used primarily in the algebraic mode of REDUCE. These commands are designed to be used interactively. However, many code generation applications require code to be generated under program control¹⁴. In these applications, it is generally more convenient to generate code from (computed) prefix forms. Therefore, GENTRAN provides code generation and file handling functions designed specifically to be used in the symbolic mode of REDUCE. This section presents the symbolic functions which are analogous to the code generation, template processing, and output file handling commands presented in sections 16.24.2, 16.24.3, and 16.24.4.

Code Generation and Translation

Sections 16.24.2 through 16.24.2 describe interactive commands and functions which generate and translate code, declare variables to be of specific types, and insert literal strings of characters into the stream of generated code. This section describes analogous symbolic mode code generation functions.

Translation of Prefix Forms In algebraic mode, the **GENTRAN** command translates algorithmic specifications supplied in the form of REDUCE statements into numerical code. Similarly, the symbolic function **SYM!-GENTRAN** trans-

¹⁴ [?] contains one such example.

lates algorithmic specifications supplied in the form of REDUCE prefix forms into numerical code.

Syntax:

SYM!-GENTRAN *form*;

Function Type:

expr

Argument:

form is any LISP prefix form that evaluates to a REDUCE prefix form that can be translated by GENTRAN into the target language¹⁵. *form* may contain any number of occurrences of the special forms **EVAL**, **LSETQ**, **RSETQ**, **LRSETQ**, **DECLARE**, and **LITERAL** (see sections 16.24.7 through 16.24.7 on pages 396–398).

Side Effects:

SYM!-GENTRAN translates *form* into formatted code in the target language and writes it to the file(s) currently selected for output.

Returned Value:

SYM!-GENTRAN returns the name(s) of the file(s) to which code was written. If code was written to one file, the returned value is an atom; otherwise, it is a list.

Diagnostic Messages:

*** OUTPUT FILE ALREADY EXISTS

OVERWRITE FILE? (Y/N)

***** WRONG TYPE OF ARG

exp

***** CANNOT BE TRANSLATED

Example 42

¹⁵ See 16.24.8 on page 409 for a complete listing of REDUCE prefix forms that can be translated.

```

1: SYMBOLIC$

2: GENTRANLANG!* := 'FORTRAN$

3:  SYM!-GENTRAN '(FOR I (1 1 n) DO (SETQ (V I) 0))$

      DO 25001 I=1,N
      V(I)=0.0
25001 CONTINUE

4: GENTRANLANG!* := 'RATFOR$

5: SYM!-GENTRAN '(FOR I (1 1 N) DO
5:               (FOR J ((PLUS I 1) 1 N) DO
5:               (PROGN
5:               (SETQ (X J I) (X I J))
5:               (SETQ (Y J I) (Y I J))))))$

DO I=1,N
  DO J=I+1,N
    {
      X(J,I)=X(I,J)
      Y(J,I)=Y(I,J)
    }

6: GENTRANLANG!* := 'C$

7: SYM!-GENTRAN '(SETQ P (FOR I (1 1 N) PRODUCT I))$

{
  P=1;
  for (I=1; I<=N; ++I)
    P*=I;
}

8:  GENTRANLANG!* := 'PASCAL$

9: SYM!-GENTRAN '(SETQ C
9:   (COND ((LESSP A B) A) (T B)))$
IF A<B THEN
  C:=A;
ELSE
  C:=B;

```

Code Generation Sections 16.24.2 through 16.24.2 on pages 333–334 described the special functions and operators **EVAL**, **::=**, **:=:**, and **::=:** that could be included in arguments to the **GENTRAN** command to indicate that parts of those arguments were to be given to REDUCE FOR Evaluation prior to translation. This section describes the analogous functions that can be supplied in prefix form to the **SYM!-GENTRAN** function.

The following special forms may be interleaved arbitrarily in forms supplied as arguments to **SYM!-GENTRAN** to specify partial evaluation: **EVAL**, **LSETQ**, **RSETQ**, and **LRSETQ**. Sections 16.24.7 through 16.24.7 describe these forms. Then section 16.24.7 through 16.24.7 present examples of the usage of these forms for evaluation of expressions in both symbolic and algebraic modes.

The EVAL Form Syntax:

(**EVAL** *form*)

Argument:

form is any LISP prefix form that evaluates to a REDUCE prefix form that can be translated by GENTRAN into the target language.

The LSETQ Form Syntax:

(**LSETQ** *svar exp*)

Arguments:

svar is a subscripted variable in LISP prefix form. Its subscripts must evaluate to REDUCE prefix forms that can be translated into the target language. *exp* is any REDUCE expression in prefix form that can be translated by GENTRAN.

The RSETQ Form Syntax:

(**RSETQ** *var exp*)

Arguments:

var is a variable in REDUCE prefix form. *exp* is a LISP prefix form which evaluates to a translatable REDUCE prefix form.

The LRSETQ Form Syntax:

(LRSETQ svar exp)

Arguments:

svar is a subscripted variable in LISP prefix form with subscripts that evaluate to REDUCE prefix forms that can be translated by GENTRAN. *exp* is a LISP prefix form that evaluates to a translatable REDUCE prefix form.

Symbolic Mode Evaluation The symbolic mode evaluation forms that have just been described are analogous to their algebraic mode counterparts, except that by default, they evaluate their argument(s) in symbolic mode. The following is an example of evaluation of subscripts in symbolic mode:

Example 43

```
1: SYMBOLIC$

2: FOR i:=1:2 DO
2:     FOR j:=1:2 DO
2:         SYM!-GENTRAN '(LSETQ (M i j) 0)$

      M(1,1)=0.0
      M(1,2)=0.0
      M(2,1)=0.0
      M(2,2)=0.0
```

Algebraic Mode Evaluation As we have just seen, the symbolic mode evaluation forms evaluate their argument(s) in symbolic mode. This default evaluation mode can be overridden by explicitly requesting evaluation in algebraic mode with the REDUCE **AEVAL** function.

Example 44

```
1: ALGEBRAIC$

2: F := 2*x^2 - 5*X + 6$

3: SYMBOLIC$

4: SYM!-GENTRAN '(SETQ Q (QUOTIENT
```

```

4:                                     (EVAL (AEVAL 'F))
4:                                     (EVAL (AEVAL ' (DF F X)))) )$

      Q=(2.0*X**2-5.0*X+6.0)/(4.0*X-5.0)

5: ALGEBRAIC$

6: M := MAT( ( A,    0,   -1,   1),
6:           ( 0, B^2,    0,   1),
6:           (-1,   B, B*C,   0),
6:           ( 1,    0,   -C, -D))$

7: SYMBOLIC$

8: FOR i:=1:4 DO
8:     SYM!-GENTRAN ' (LRSETQ (M i i)
                        (AEVAL (MKQUOTE (LIST 'M i i)))) )$

      M(1,1)=A
      M(2,2)=B**2
      M(3,3)=B*C
      M(4,4)=-D

```

SHARED Variables The REDUCE **SHARE** command enables variables to be shared between algebraic and symbolic modes. Thus, we can derive an expression in algebraic mode, assign it to a shared variable, and then access the value of that variable to generate code from symbolic mode.

Example:

```

1: ALGEBRAIC$

2: SHARE dfx1$

3: dfx1 := DF(X**4 - X**3 + 2*X**2 + 1, X)$

4: SYMBOLIC$

5: SYM!-GENTRAN ' (RSETQ DERIV dfx1)$
      DERIV=4.0*X**3-(3.0*X**2)+4.0*X

```

Special Translatable Forms Sections 16.24.2 through 16.24.2 described special functions that could be used to declare variable types and insert literal strings of

characters into generated code. This section contains explanations of analogous prefix forms for usage in symbolic mode.

Explicit Type Declarations A similar form of the algebraic mode **DECLARE** function is provided in symbolic mode:

Syntax:

```
(DECLARE  (type1 v1 v2 ... vn1)
          (type2 v1 v2 ... vn2)
          :
          (typen v1 v2 ... vnn))
```

Arguments:

Each $v1\ v2\ \dots\ vn$ is a sequence of one or more variables (optionally subscripted to indicate array dimensions – in prefix form), or variable ranges (two letters concatenated together with "-" in between). vs are not evaluated unless given as arguments to **EVAL**.

Each *type* is a variable type in the target language. Each must be an atom, optionally concatenated to the atom **IMPLICIT!** (note the trailing space). *types* are not evaluated unless given as arguments to **EVAL**.

Side Effect:

Entries are placed in the symbol table for each variable or variable range declared in the call to this function. The function call itself is removed from the statement group being translated. Then after translation, type declarations are generated from these symbol table entries before the resulting executable statements are printed.

Example 45

```
1: SYMBOLIC$

2: GENTRANLANG!* := 'FORTRAN$

3: SYM!-GENTRAN
3:      ' (PROGN
3:          (DECLARE (IMPLICIT! REAL!*8 A!-H O!-Z)
3:                  (INTEGER (M 4 4) ) )
3:          (FOR I (1 1 4) DO
```

```

3:          (FOR J (1 1 4) DO
3:              (COND ((EQUAL I J) (SETQ (M I J) 1))
3:                  (T (SETQ (M I J) 0))))
3:          (DECLARE (INTEGER I J))
:
:
3:      )$

```

```

IMPLICIT REAL*8 (A-H,O-Z)
INTEGER M(4,4), I, J
DO 25001 I=1,4
    DO 25002 J=1,4
        IF (I.EQ.J) THEN
            M(I,J)=1
        ELSE
            M(I,J)=0
        ENDIF
25002    CONTINUE
25001 CONTINUE
:
:

```

```

4: GENTRANLANG!* := 'RATFOR$

```

```

5: SYM!-GENTRAN
5:      '(PROCEDURE FAC NIL EXPR (N)
5:          (BLOCK ()
5:              (DECLARE (FUNCTION FAC)
5:                  (INTEGER FAC N))
5:              (SETQ F (FOR I (1 1 N) PRODUCT I))
5:              (DECLARE (INTEGER F I))
5:              (RETURN F)))$

```

```

INTEGER FUNCTION FAC(N)
INTEGER N,F,I
{
    F=1
    DO I=1,N
        F=F*I
    }
RETURN (F)
END

```

```

6: GENTRANLANG!* := 'C$

```

```

7:  SYM!-GENTRAN
7:      ' (PROCEDURE FAC NIL EXPR (N)
7:          (BLOCK ()
7:              (DECLARE (INTEGER FAC N I F))
7:              (SETQ F (FOR I (1 1 N) PRODUCT I))
7:              (RETURN F)))$

int FAC(N)
int N;
{
    int I,F;
    {
        F=1;
        for (I=1;I<=N;++I)
            F*=I;
    }
    return(F);
}

8:  GENTRANLANG!* := 'PASCAL$

9:  SYM!-GENTRAN
9:      ' (PROCEDURE FAC NIL EXPR (N)
9:          (BLOCK ()
9:              (DECLARE (INTEGER FAC N I F))
9:              (SETQ F (FOR I (1 1 N) PRODUCT I))
9:              (RETURN F)))$
FUNCTION FAC(N:INTEGER):INTEGER;
LABEL
    99999;
VAR
    I,F: INTEGER;
BEGIN
    BEGIN
        F:=1;
        FOR I:=1 TO N DO
            F:=F*I
        END;
    BEGIN
        FAC:=F;
        GOTO 99999{RETURN}
    END;
99999:

```

END;

Comments and Literal Strings A form similar to the algebraic mode **LITERAL** function is provided in symbolic mode:

Syntax:

(LITERAL *arg1 arg2 ... argn*)

Arguments:

arg1 arg2 ... argn is an argument sequence containing one or more *args*, where each *arg* either is, or evaluates to, an atom. The atoms **TAB!*** and **CR!*** have special meanings. *args* are not evaluated unless given as arguments to **EVAL**.

Side Effect:

This form is replaced by the character sequence resulting from concatenation of the given atoms. Double quotes are stripped from all string type *args*, and the reserved atoms **TAB!*** and **CR!*** are replaced by a tab to the current level of indentation, and an end-of-line character, respectively.

Example 46

```

1: SYMBOLIC$

2: GENTRANLANG!* := 'FORTRAN$

3: N := 100$

4: SYM!-GENTRAN
4: ' (PROGN
4:   (LITERAL C TAB!* "--THIS IS A FORTRAN COMMENT--"
4:     CR!* C CR!*)
4:   (LITERAL TAB!* "DATA N/" (EVAL N) "/" CR!*))$

C   --THIS IS A FORTRAN COMMENT--
C
    DATA N/100/

```

```

5: GENTRANLANG!* := 'RATFOR$

6: SYM!-GENTRAN
6: ' (FOR I (1 1 N) DO
6:   (PROGN
6:     (LITERAL TAB!* "# THIS IS A RATFOR COMMENT" CR!*)
6:     (LITERAL TAB!* "WRITE(6,10) (M(I,J),J=1,N)" CR!*
6:       10 TAB!* "FORMAT(1X,10(I5,3X))" CR!*))$

DO I=1,N
  {
    # THIS IS A RATFOR COMMENT
    WRITE(6,10) (M(I,J),J=1,N)
10    FORMAT(1X,10(I5,3X))
  }

7: GENTRANLANG!* := 'C$

8: SYM!-GENTRAN
8: ' (PROGN
8:   (SETQ X 0)
8:   (LITERAL "/* THIS IS A" CR!* "
8:     C COMMENT */" CR!*))$

{
  X=0.0;
/* THIS IS A
  C COMMENT */
}

9: GENTRANLANG!* := 'PASCAL$

10: SYM!-GENTRAN
10: ' (PROGN
10:   (SETQ X (SIN Y))
10:   (LITERAL "{ THIS IS A PASCAL COMMENT }" CR!*))$
BEGIN
  X:=SIN(Y)
{ THIS IS A PASCAL COMMENT }
END;

```

Template Processing

The template processor can be invoked from either algebraic or symbolic mode. Section 16.24.3 described the algebraic mode command. This section describes the analogous symbolic mode function.

Syntax:

SYM!-GENTRANIN *list-of-fnames*;

Function Type:

expr

Argument:

list-of-fnames evaluates to a LISP list containing one or more *fnames*, where each *fname* is one of:

<i>an atom</i>	=	a template (input) file
T	=	the terminal

Side Effects:

SYM!-GENTRANIN processes each template file in *list-of-fnames* sequentially.

A template file may contain any number of parts, each of which is either an active or an inactive part. All active parts start with the character sequence **;BEGIN;** and end with **;END;**. The end of the template file is indicated by an extra **;END;** character sequence.

Inactive parts of template files are assumed to contain code in the target language (FORTRAN, RATFOR, PASCAL or C, depending on the value of the global variable **GENTRANLANG!**). All inactive parts are copied to the output. Comments delimited by the appropriate characters are also copied in their entirety to the output. Thus the character sequences **;BEGIN;** and **;END;** have no special meanings within comments.

Active parts may contain any number of REDUCE expressions, statements, and commands. They are not copied directly to the output. Instead, they are given to REDUCE for evaluation in algebraic mode¹⁶.

¹⁶ Active parts are evaluated in algebraic mode unless the mode is explicitly changed to symbolic from within the active part itself. This is true regardless of which mode the system was in when the template processor was called.

All output generated by each evaluation is sent to the file(s) currently selected for output. Returned values are only printed on the terminal.

Active parts will most likely contain calls to GENTRAN to generate code. This means that the result of processing a template file will be the original template file with all active parts replaced by generated code.

Returned Value:

SYM!-GENTRANIN returns the name(s) of the file(s) to which code was written. If code was written to one file, the returned value is an atom; otherwise, it is a list.

Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS

OVERWRITE FILE? (Y/N)

***** NONEXISTENT INPUT FILE

***** TEMPLATE FILE ALREADY OPEN FOR INPUT

***** WRONG TYPE OF ARG
```

Output Redirection

Section [16.24.4](#) describes four algebraic mode commands which select, open, and close output files. The algebraic mode commands **GENTRANOUT**, **GENTRANSHUT**, **GENTRANPUSH**, and **GENTRANPOP** are analogous to the symbolic mode **SYM!-GENTRANOUT**, **SYM!-GENTRANSHUT**, **SYM!-GENTRANPUSH**, and **SYM!-GENTRANPOP** functions, respectively.

SYM!-GENTRANOUT Syntax:

SYM!-GENTRANOUT *list-of-fnames*;

Function Type:

expr

Argument:

list-of-fnames evaluates to a LISP list containing one or more *fnames*, where each *fname* is one of:

<i>an atom</i>	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN

Side Effect:

GENTRAN maintains a list of files currently open for output by GENTRAN *only*. **SYM!-GENTRANOUT** inserts each file name represented in *list-of-fnames* into that list and opens each one for output. It also resets the currently selected output file(s) to be all of the files represented in *list-of-fnames*.

Returned Value:

SYM!-GENTRANOUT returns the name(s) of the file(s) represented by *list-of-fnames*; i.e., the current output file(s) after the command has been executed. If there is only one file selected for output, the returned value is an atom; otherwise, it is a list.

Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS

OVERWRITE FILE? (Y/N)

***** WRONG TYPE OF ARG
```

SYM!-GENTRANSHUT Syntax:

SYM!-GENTRANSHUT *list-of-fnames* ;

Function Type:

expr

Argument:

list-of-fnames evaluates to a LISP list containing one or more *fnames*, where each *fname* is one of:

an atom = an output file
NIL = the current output file(s)
ALL!* = all files currently open for output
 by GENTRAN

Side Effects:

SYM!-GENTRANSHUT creates a list of file names from *list-of-fnames*, deletes each from the output file list, and closes the corresponding files. If (all of) the current output file(s) are closed, then the current output file is reset to the terminal.

Returned Value:

SYM!-GENTRANSHUT returns the name(s) of the file(s) selected for output after the command has been executed. If there is only one file selected for output, the returned value is an atom; otherwise, it is a list.

Diagnostic Messages:

```
*** FILE NOT OPEN FOR OUTPUT

***** WRONG TYPE OF ARG
```

SYM!-GENTRANPUSH Syntax:

SYM!-GENTRANPUSH *list-of-fnames*;

Function Type:

expr

Argument:

list-of-fnames evaluates to a LISP list containing one or more *fnames*, each of which is one of:

an atom = an output file
T = the terminal
NIL = the current output file(s)
ALL!* = all files currently open for output
 by GENTRAN

Side Effects:

SYM!-GENTRANPUSH creates a list of file name(s) from *lis-of-fnames* and pushes that list onto the output stack. Each file in the list that is not already open for output is opened at this time. The current output file is reset to this new element on the top of the stack.

Returned Value:

SYM!-GENTRANPUSH returns the name(s) of the file(s) represented by *list-of-fnames*; i.e., the current output file(s) after the command has been executed. If there is only one file selected for output, the returned value is an atom; otherwise, it is a list.

Diagnostic Messages:

```
*** OUTPUT FILE ALREADY EXISTS

OVERWRITE FILE? (Y/N)

***** WRONG TYPE OF ARG
```

SYM!-GENTRANPOP Syntax:

SYM!-GENTRANPOP *list-of-fnames*;

Function Type:

expr

Argument:

list-of-fnames evaluates to a LISP list containing one or more *fnames*, where each *fname* is one of:

<i>an atom</i>	=	an output file
T	=	the terminal
NIL	=	the current output file(s)
ALL!*	=	all files currently open for output by GENTRAN

Side Effects:

SYM!-GENTRANPOP deletes the top-most occurrence of the single element containing the file name(s) represented by *list-of-fnames* from the output stack. Files whose names have been completely removed from the output stack are closed. The current output file is reset to the (new) element on the top of the output stack.

Returned Value:

SYM!-GENTRANPOP returns the name(s) of the file(s) selected for output after the command has been executed. If there is only one file selected for output, the returned value is an atom; otherwise, it is a list.

Diagnostic Messages:

```
*** FILE NOT OPEN FOR OUTPUT
```

```
***** WRONG TYPE OF ARG
```

16.24.8 Translatable REDUCE Expressions & Statements

A substantial subset of all REDUCE expressions and statements can be translated by GENTRAN into semantically equivalent code in the target numerical language¹⁷. This section contains examples and a formal definition of translatable REDUCE expressions and statements.

Examples of Translatable Statements

The following three tables contain listings of REDUCE statement types that can be translated by GENTRAN. An example of each statement type is shown, and FORTRAN, RATFOR, PASCAL and C code generated for each example is also shown.

Formal Definition

The remainder of this section contains a formal definition of all REDUCE expressions, statements, and prefix forms that can be translated by GENTRAN into FORTRAN, RATFOR, PASCAL and C code.

¹⁷ It should be noted that call-by-value parameter passing is used in REDUCE, whereas call-by-address parameter passing is normally used in FORTRAN and RATFOR. GENTRAN does *not* attempt to simulate call-by-value passing in FORTRAN and RATFOR, although this could be done by generating temporary variables, assigning values to them, and using them in subprogram calls.

TYPE	EXAMPLE	FORTRAN CODE
simple	V:=X^2+X\$	V=X**2+X
matrix	M:=MAT((U,V), (W,X))\$	M(1,1)=U M(1,2)=V M(2,1)=W M(2,2)=X
sum	S:=FOR I:=1:10 SUM V(I)\$	S=0.0 DO 25001 I=1,10 S=S+V(I) 25001 CONTINUE
product	P:=FOR I:=2 STEP 2 UNTIL N PRODUCT I\$	P=1 DO 25002 I=2,N,2 P=P*I 25002 CONTINUE
conditional	X := IF A<B THEN A ELSE B\$	IF (A.LT.B) THEN X=A ELSE X=B ENDIF

Table 16.2: REDUCE assignments translatable to FORTRAN

TYPE	EXAMPLE	FORTRAN CODE
for	FOR I:=1:8 DO V(I):=0.0\$	DO 25003 I=1,8 V(I)=0.0 25003 CONTINUE
while	WHILE F(N)>0.0 DO N:=N+1\$	25004 IF (.NOT.F(N).GT.0.0) . GOTO 25005 N=N+1 GOTO 25004 25005 CONTINUE
repeat	REPEAT X:=X/2.0 UNTIL F(X)<0.0\$	25006 CONTINUE X=X/2.0 IF (.NOT.F(X).LT.0.0) . GOTO 25006

Table 16.3: REDUCE Loop structures translatable to FORTRAN

Preliminary Definitions

An *id* is an identifier. Certain *id*'s are reserved words and may not be used as array names or subprogram names. The complete list appears in the *Reserved Words* section.

A *string* consists of any number of characters (excluding double quotes) which are enclosed in double quotes.

Reserved Words

The following reserved words may not be used as array names or subprogram names¹⁸:

AND, BLOCK, COND, DIFFERENCE, EQUAL, EXPT, FOR, GEQ, GO, GREATERP, LEQ, LESSP, MAT, MINUS, NEQ, NOT, OR, PLUS, PROCEDURE, PROGN, QUOTIENT, RECIP, REPEAT, RETURN, SETQ, TIMES, WHILE, WRITE

Translatable REDUCE Expressions and Statements Expressions

¹⁸Note that names of other built-in REDUCE functions *can* be translated, but remember that they will be translated *literally* unless **EVAL**'d first. For example: **GENTRAN DERIV := DF(2*X^2-X-1, X)\$** generates **DERIV=DF(2*X**2-X-1, X)** whereas **GENTRAN DERIV :=: DF(2*X^2-X-1, X)\$** generates **DERIV=4*X-1**

TYPE	EXAMPLE	FORTRAN CODE
Conditionals:		
if	IF X>0.0 THEN Y:=X\$	IF (X.GT.0.0) THEN Y=X ENDIF
if - else	IF X>0.0 THEN Y:=X ELSE Y:=-X\$	IF (X.GT.0.0) THEN Y=X ELSE Y=-X ENDIF
Unconditional Transfer of Control:		
goto	GOTO LOOP\$	GOTO 25010
call	CALCV(V,X,Y,Z)\$	CALL CALCV(V,X,Y,Z)
return	RETURN X^2\$	<i>functionname</i> =X**2 RETURN
Sequences & Groups:		
sequence	<< U:=X^2; V:=Y^2>>\$	U=X**2 V=Y**2
group	BEGIN U:=X^2; V:=Y^2 END\$	U=X**2 V=Y**2

Table 16.4: REDUCE control structures translatable to FORTRAN

TYPE	EXAMPLE	RATFOR CODE
Assignments:		
simple	V:=X^2+X\$	V=X**2+X
matrix	M:=MAT((U,V),(W,X))\$	M(1,1)=U M(1,2)=V M(2,1)=W M(2,2)=X
sum	S:=FOR I:=1:10 SUM V(I)\$	S=0.0 DO I=1,10 S=S+V(I)
product	P:=FOR I:=2 STEP 2 UNTIL N PRODUCT I\$	P=1 DO I=2,N,2 P=P*I
conditional	X := IF A<B THEN A ELSE B\$	IF (A<B) X=A ELSE X=B
Control Structures:		
Loops:		
for	FOR I:=1:8 DO V(I):=0.0\$	DO I=1,8 V(I)=0.0
while	WHILE F(N)>0.0 DO N:=N+1\$	WHILE (F(N)>0.0) N=N+1
repeat	REPEAT X:=X/2.0 UNTIL F(X)<0.0\$	REPEAT X=X/2.0 UNTIL (F(X)<0.0)

Table 16.5: REDUCE forms translatable to RATFOR

TYPE	EXAMPLE	RATFOR CODE
Conditionals:		
if	IF X>0.0 THEN Y:=X\$	IF (X>0.0) Y=X
if - else	IF X>0.0 THEN Y:=X ELSE Y:=-X\$	IF (X>0.0) Y=X ELSE Y=-X
Unconditional Transfer of Control:		
goto	GOTO LOOP\$	GOTO 25010
call	CALCV(V,X,Y,Z)\$	CALL CALCV(V,X,Y,Z)
return	RETURN X^2\$	RETURN (X**2)
Sequences & Groups:		
sequence	<< U:=X^2;V:=Y^2>>\$	U=X**2 V=Y**2
group	BEGIN U:=X^2; V:=Y^2 END\$	{ U=X**2 V=Y**2 }

Table 16.6: REDUCE forms translatable to RATFOR

TYPE	EXAMPLE	PASCAL CODE
Assignments:		
simple	V:=X^2+X\$	V=X**2+X;
matrix	M:=MAT((U,V), (W,X))\$	BEGIN M(1,1)=U; M(1,2)=V; M(2,1)=W; M(2,2)=X; END;
sum	S:=FOR I:=1:10 SUM V(I)\$	BEGIN S=0.0 FOR I:=1 TO 10 DO S:=S+V(I) END;
product	P:=FOR I:=2:N PRODUCT I\$	BEGIN P:=1; FOR I:=2 TO N DO P:=P*I END;
conditional	X := IF A<B THEN A ELSE B\$	IF (A<B) THEN X:=A; ELSE X:=B;

Table 16.7: REDUCE forms translatable to PASCAL

TYPE	EXAMPLE	PASCAL CODE
Control Structures:		
Loops:		
for	FOR I:=1:8 DO V(I):=0.0\$	FOR I:=1 TO 8 DO V(I):=0.0;
while	WHILE F(N)>0.0 DO N:=N+1\$	WHILE (F(N)>0.0) N:=N+1.0;
repeat	REPEAT X:=X/2.0 UNTIL F(X)<0.0\$	REPEAT X:=X/2.0 UNTIL F(X)<0.0;

Table 16.8: REDUCE forms translatable to PASCAL

Arithmetic Expressions:

exp ::= *number* | var | funcall | - exp | / exp | exp + exp |
exp - exp | exp * exp | exp / exp | exp ** exp |
exp ^ exp | (exp)

var ::= *id* | *id* (exp₁, exp₂, ... , exp_n) *n* > 0

funcall ::= *id* (arg₁, arg₂, ... , arg_n) *n* ≥ 0

arg ::= exp | logexp | *string*

Logical Expressions:

logexp ::= *T* | *NIL* | var | funcall | exp > exp | exp ≥ exp |
exp = exp | exp *NEQ* exp | exp < exp |
exp ≤ exp | *NOT* logexp | logexp *AND* logexp |
logexp *OR* logexp | (logexp)

Operator Precedence

The following is a list of REDUCE arithmetic and logical operators in order of decreasing precedence:

TYPE	EXAMPLE	PASCAL CODE
Conditionals:		
if	IF X>0.0 THEN Y:=X\$	IF X>0.0 THEN Y:=X;
if - else	IF X>0.0 THEN Y:=X ELSE Y:=-X\$	IF X>0.0 THEN Y:=X; ELSE Y:=-X;
Unconditional Transfer of Control:		
goto	GOTO LOOP\$	GOTO 25010;
call	CALCV(V,X,Y,Z)\$	CALCV(V,X,Y,Z);
return	RETURN X^2\$	<i>functionname</i> =X**2; GOTO 99999{RETURN} 99999;
Sequences & Groups:		
sequence	<< U:=X^2;V:=Y^2>>\$	BEGIN U:=X**2; V:=Y**2 END;
group	BEGIN U:=X^2; V:=Y^2 END\$	BEGIN U:=X**2; V:=Y**2 END

Table 16.9: REDUCE forms translatable to PASCAL

TYPE	EXAMPLE	C CODE
Assignments:		
simple	V:=X^2+X\$	<code>V=power (X, 2)+X;</code>
matrix	M:=MAT((U,V),(W,X))\$	<code>M[1][1]=U; M[1][2]=V; M[2][1]=W; M[2][2]=X;</code>
sum	S:=FOR I:=1:10 SUM V(I)\$	<code>S=0.0; for (I=1; I<=10; ++I) S+=V[I];</code>
product	P:=FOR I:=2 STEP 2 UNTIL N PRODUCT I\$	<code>P=1; for (I=2; I<=N; ++I) P*=I;</code>
conditional	X := IF A<B THEN A ELSE B\$	<code>if (A<B) X=A; else X=B;</code>
Control Structures:		
Loops:		
for	FOR I:=1:8 DO V(I):=0.0\$	<code>for (I=1; I<=8; ++I) V[I]=0.0;</code>
while	WHILE F(N)>0.0 DO N:=N+1\$	<code>while (F(N)>0.0) N+=1;</code>
repeat	REPEAT X:=X/2.0 UNTIL F(X)<0.0\$	<code>do X/=2.0; while (F(X)>=0.0);</code>

Table 16.10: REDUCE forms translatable to C

TYPE	EXAMPLE	C CODE
Conditionals:		
if	IF X>0.0 THEN Y:=X\$	if (X>0.0) Y=X;
if - else	IF X>0.0 THEN Y:=X ELSE Y:=-X\$	if (X>0.0) Y=X; else Y=-X;
Unconditional Transfer of Control:		
goto	GOTO LOOP\$	goto LOOP;
call	CALCV(V,X,Y,Z)\$	CALCV(V,X,Y,Z);
return	RETURN X^2\$	return(power(X,2));
Sequences & Groups:		
sequence	<< U:=X^2;V:=Y^2>>\$	U=power(X,2); V=power(Y,2);
group	BEGIN U:=X^2; V:=Y^2 END\$	{ U=power(x,2); V=power(Y,2); }

Table 16.11: REDUCE forms translatable to C

**** (or ^) / * — + < <= > >= NEQ = NOT AND OR**

When unparenthesised expressions are translated which contain operators whose precedence in REDUCE differs from that in the target language, parentheses are automatically generated. Thus the meaning of the original expression is preserved¹⁹.

Statements

stmt ::= assign | break | cond | while | repeat | for | goto | label |
call | return | stop | stmtgp

Assignment Statements:

assign ::= var := assign' | matassign | cond

assign' ::= exp | logexp

matassign ::= id := MAT((exp₁₁, ..., exp_{1m}),
(exp₂₁, ..., exp_{2m}),
:
:
(exp_{n1}, ..., exp_{nm})) n, m > 0

Break Statement:

break ::= BREAK()

Conditional Statements:

cond ::= IF logexp THEN stmt
IF logexp THEN stmt ELSE stmt

Loops:

¹⁹ For example in REDUCE, **NOT A = B** and **NOT (A = B)** are equivalent, whereas in C, **! A == B** and **(!A) == B** are equivalent. Therefore, **NOT A = B** is translated into C code which forces the REDUCE precedence rules: **!(A == B)**

while ::= *WHILE* logexp *DO* stmt
 repeat ::= *REPEAT* stmt *UNTIL* logexp
 for ::= *FOR* var := exp *STEP* exp *UNTIL* exp *DO* stmt |
 FOR var := exp *UNTIL* exp *DO* stmt |
 FOR var := exp : exp *DO* stmt |
 var := for' |
 for' ::= var := for' |
 FOR var := exp *STEP* exp *UNTIL* exp *SUM* exp |
 FOR var := exp *UNTIL* exp *SUM* exp |
 FOR var := exp : exp *SUM* exp |
 FOR var := exp *STEP* exp *UNTIL* exp
 PRODUCT exp |
 FOR var := exp *UNTIL* exp *PRODUCT* exp |
 FOR var := exp : exp *PRODUCT* exp

Goto Statement:

goto ::= *GOTO* label | *GO TO* label
 label ::= *id* :

Subprogram Calls & Returns ²⁰:

call ::= *id* (*arg*₁, *arg*₂, ... , *arg*_{*n*}) *n* ≥ 0
 return ::= *RETURN* | *RETURN* arg

Stop & Exit Statements ²¹:

stop ::= *STOP*()

Statement Groups ²²:

stmtgp ::= << stmt₁ ; stmt₂ ; ... ; stmt_{*n*} >> |
 BEGIN stmt₁ ; stmt₂ ; ... ; stmt_{*n*} *END* *n* > 0

Subprogram Definitions

defn ::= *PROCEDURE* *id* (*id*₁, *id*₂, ... , *id*_{*n*}) ; stmt |
 PROCEDURE *id* (*id*₁, *id*₂, ... , *id*_{*n*}) ; exp *n* ≥ 0

²⁰ Note that return statements can only be translated from inside of procedure definitions. The LITERAL function must be used to generate a return statement from anywhere else.

²¹ In certain cases it may be convenient to generate a FORTRAN STOP statement or a C EXIT statement. Since there is no semantically equivalent REDUCE statement, STOP() can be used and will be translated appropriately.

²² Note that REDUCE BEGIN... END statement groups are translated into RATFOR or C {... } statement groups, whereas REDUCE <<... >> statement groups are translated into RATFOR or C statement sequences. When the target language is FORTRAN, both types of REDUCE statement groups are translated into statement sequences.

Translatable REDUCE Prefix Forms Expressions

Arithmetic Expressions:

$$\begin{aligned} \text{exp} ::= & \text{number} \mid \text{funcall} \mid \text{var} \mid (\text{DIFFERENCE exp exp}) \mid \\ & (\text{EXPT exp exp}) \mid (\text{MINUS exp}) \mid (\text{PLUS exp exp}') \mid \\ & (\text{QUOTIENT exp exp}) \mid (\text{RECIP exp}) \mid \\ & (\text{TIMES exp exp exp}') \mid (!*SQ \text{sqform}) \end{aligned}$$

where sqform is a standard quotient form equivalent to any acceptable prefix form.

$$\text{exp}' ::= \text{exp}_1 \text{exp}_2 \dots \text{exp}_n \quad n \geq 0$$

Logical Expressions:

$$\begin{aligned} \text{logexp} ::= & \text{NIL} \mid T \mid \text{funcall} \mid \text{var} \mid \\ & (\text{AND logexp logexp logexp}') \mid (\text{EQUAL exp exp}) \mid \\ & (\text{GEQ exp exp}) \mid (\text{GREATERP exp exp}) \mid \\ & (\text{LEQ exp exp}) \mid (\text{LESSP exp exp}) \mid \\ & (\text{NEQ exp exp}) \mid (\text{NOT logexp}) \mid \\ & (\text{OR logexp logexp logexp}') \end{aligned}$$

$$\text{logexp}' ::= \text{logexp}_1 \text{logexp}_2 \dots \text{logexp}_n \quad n \geq 0$$
Statements

$$\begin{aligned} \text{stmt} ::= & \text{assign} \mid \text{break} \mid \text{call} \mid \text{cond} \mid \text{for} \mid \text{goto} \mid \\ & \text{label} \mid \text{read} \mid \text{repeat} \mid \text{return} \mid \text{stmtgp} \mid \\ & \text{stop} \mid \text{while} \mid \text{write} \end{aligned}$$

$$\text{stmt}' ::= \text{stmt}_1 \text{stmt}_2 \dots \text{stmt}_n \quad n \geq 0$$

Assignment Statements:

$$\text{assign} ::= (\text{SETQ var exp}) \mid (\text{SETQ var logexp}) \mid (\text{SETQ id (MAT list list')})$$

Conditional Statements:

$$\text{cond} ::= (\text{COND (logexp stmt) cond1})$$

$$\text{cond1} ::= (\text{logexp stmt}_1) \dots (\text{logexp stmt}_n) \quad n \geq 0$$

Loops:

for ::= (*FOR* var (exp exp exp) *DO* stmt) |
 (*SETQ* var (*FOR* var (exp exp exp) *SUM* exp) |
 (*SETQ* var (*FOR* var (exp exp exp)
 PRODUCT exp)

repeat ::= (*REPEAT* stmt logexp)

while ::= (*WHILE* logexp stmt)

Go To Statements:

break ::= (*BREAK*)

goto ::= (*GO* label)

label ::= *id*

Subprogram Calls & Returns:

call ::= (*id* arg')

return ::= (*RETURN*) | (*RETURN* arg)

Stop & Exit Statements:

stop ::= (*STOP*)

Statement Groups:

stmtgp ::= (*PROGN* stmt stmt') | (*BLOCK* (id') stmt')

I/O Statements:

read ::= (*SETQ* var (*READ*))

write ::= (*WRITE* arg arg')

Subprogram Definitions:

defn ::= (*PROCEDURE id NIL EXPR* (id') stmt)

Miscellaneous

`funcall ::= (id arg')`
`var ::= id | (id exp exp')`
`arg ::= string | exp | logexp`
`arg' ::= arg1 arg2 ... argn $n \geq 0$`
`list ::= (exp exp')`
`list' ::= list1 list2 ... listn $n \geq 0$`
`id' ::= id1 id2 ... idn $n \geq 0$`

16.24.9 List of Commands, Switches, & Variables

COMMANDS

GENTRAN *stmt* [**OUT***f1,f2,...,fn*];
GENTRANIN *f1,f2,...,fm* [**OUT***f1,f2,...,fn*];
GENTRANOUT *f1,f2,...,fn*;
GENTRANSHUT *f1,f2,...,fn*;
GENTRANPUSH *f1,f2,...,fn*;
GENTRANPOP *f1,f2,...,fn*;

SPECIAL FUNCTIONS & OPERATORS

EVAL *exp*
`var ::= exp;`
`var ::= exp;`
`var ::= exp;`
`var LSETQ exp;`
`var RSETQ exp;`
`var LRSETQ exp;`
DECLARE *v1,v2,...,vn*: *type*;

DECLARE

<<

```

    v11,v12,... ,v1n : type1;
    v12,v22,... ,v2n : type2;
    :
    :
    vm1,vm2,... ,vmn : typen;

```

>>;

LITERAL *arg1,arg2,... ,argn;***MODE SWITCHES****PERIOD****GENTRANSEG****GENDECS****DOUBLE****MAKECALLS****KEEPDECS****GETDECS****VARIABLES****GENTRANLANG!*****MAXEXPPRINTLEN!*****TEMPVARNAME!*****TEMPVARNUM!*****TEMPVARTYPE!*****GENSTMTNUM!*****GENSTMTINCR!*****TABLEN!*****FORTLINELEN!*****RATLINELEN!*****CLINELEN!*****PASCLINELEN!*****MINFORTLINELEN!*****MINRATLINELEN!*****MINCLINELEN!***

MINPASCLINELEN!*

DEFTYPE!*

TEMPORARY VARIABLE GENERATION, MARKING & UNMARKING

TEMPVAR *type*;

MARKVAR *var*;

UNMARKVAR *var*;

EXPLICIT GENERATION OF TYPE DECLARATIONS

GENDECS *subprogname*;

SYMBOLIC MODE FUNCTIONS

SYM!-GENTRAN *form*;

SYM!-GENTRANIN *list-of-fnames*;

SYM!-GENTRANOUT *list-of-fnames*;

SYM!-GENTRANSHUT *list-of-fnames*;

SYM!-GENTRANPUSH *list-of-fnames*;

SYM!-GENTRANPOP *list-of-fnames*;

SYMBOLIC MODE SPECIAL FORMS

(DECLARE (*type1 v11 v12 ... v1n*)
 (*type2 v21 v22 ... v2n*)
 :
 :
 (*typen vn1 vn2 ... vnn*))

(LITERAL *arg1 arg2 ... argn*)

(EVAL *exp*)

(LSETQ *var exp*)

(RSETQ *var exp*)

(LRSETQ *var exp*)

16.24.10 The Programs **M1.F** and **M2.F**.

This section contains the two files generated in chapter 6. Contents of file **m1.f**:

```

M(1,1)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(Q3))**2*Y*J30)+DSIN(DBLE(Q3))**2*J30Z+18.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**2*M30+18.0D0*P**2*M30+P**2*M10+J30Y+J10Y
M(1,2)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(Q3))**2*J30Y)+DSIN(DBLE(Q3))**2*J30Z+9.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**2*M30+9.0D0*P**2*M30+J30Y
M(1,3)=- (9.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**2*M30)
M(2,2)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(Q3))**2*J30Y)+DSIN(DBLE(Q3))**2*J30Z+9.0D0*P**2*M30+J30Y
M(2,3)=0.0D0
M(3,3)=9.0D0*P**2*M30+J30X
MIV(1,1)=-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30X)-(DSIN(DBLE(Q3))**2*J30Y*J30X)+DSIN(DBLE(Q3))**2*J30Z*J30X+81.0D0*P**4*M30**2+9.0D0*P**2*M30*J30Y+9.0D0*P**2*M30*J30X+J30Y*J30X)/(729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30**3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*M30**2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3))**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X-(729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*

```

```

. J10Y)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y+9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0*DSIN(DBLE
. (Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))**2*P**2*
. J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*M10*J30X-(
. DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE(Q3))**2*
. J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*J30X)+DSIN(
. DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(DBLE(Q3))**2
. *DCOS(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DCOS(DBLE(Q3))
. **2*DCOS(DBLE(Q2))**2*P**4*M30**2*J30X)+729.0D0*P**6*
. M30**3+81.0D0*P**6*M30**2*M10+81.0D0*P**4*M30**2*J30Y+
. 81.0D0*P**4*M30**2*J10Y+81.0D0*P**4*M30**2*J30X+9.0D0*
. P**4*M30*J30Y*M10+9.0D0*P**4*M30*M10*J30X+9.0D0*P**2*
. M30*J30Y*J10Y+9.0D0*P**2*M30*J30Y*J30X+9.0D0*P**2*M30*
. J10Y*J30X+P**2*J30Y*M10*J30X+J30Y*J10Y*J30X)
MIV(1,2)=(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2+9.0D0*
. DSIN(DBLE(Q3))**2*P**2*M30*J30Y-(9.0D0*DSIN(DBLE(Q3))
. **2*P**2*M30*J30Z)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*
. J30X+DSIN(DBLE(Q3))**2*J30Y*J30X-(DSIN(DBLE(Q3))**2*
. J30Z*J30X)-(81.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**4*
. M30**2)-(9.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**2*M30*
. J30X)-(81.0D0*P**4*M30**2)-(9.0D0*P**2*M30*J30Y)-(
. 9.0D0*P**2*M30*J30X)-(J30Y*J30X))/(729.0D0*DSIN(DBLE(
. Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30**3+81.0D0*DSIN(DBLE
. (Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y-(81.0D0*
. DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Z)+
. 81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*M30**2*J30-(81.0D0*
. DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)+9.0D0*DSIN(DBLE(Q3
. ))**4*P**2*Y*M30*J30Y*J30-(9.0D0*DSIN(DBLE(Q3))**4*P**
. 2*Y*M30*J30Z*J30)+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*
. J30X*J30-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN
. (DBLE(Q3))**4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*
. J30Y*J30X*J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(
. DSIN(DBLE(Q3))**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y
. *J30Z*J30X-(729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**
. 2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2)
. )**2*P**4*M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6
. *M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*
. DSIN(DBLE(Q3))**2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P
. **4*M30**2*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*
. J30Y*M10)+9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(
. 9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN

```

```

. (DBLE(Q3))**2*P**2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3
. ))**2*P**2*Y*M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**
. 2*M30*J30Y**2-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*
. J10Y)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y+9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0*DSIN(DBLE
. (Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))**2*P**2*
. J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*M10*J30X-(
. DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE(Q3))**2*
. J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*J30X)+DSIN(
. DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(DBLE(Q3))**2
. *DCOS(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DCOS(DBLE(Q3))
. **2*DCOS(DBLE(Q2))**2*P**4*M30**2*J30X)+729.0D0*P**6*
. M30**3+81.0D0*P**6*M30**2*M10+81.0D0*P**4*M30**2*J30Y+
. 81.0D0*P**4*M30**2*J10Y+81.0D0*P**4*M30**2*J30X+9.0D0*
. P**4*M30*J30Y*M10+9.0D0*P**4*M30*M10*J30X+9.0D0*P**2*
. M30*J30Y*J10Y+9.0D0*P**2*M30*J30Y*J30X+9.0D0*P**2*M30*
. J10Y*J30X+P**2*J30Y*M10*J30X+J30Y*J10Y*J30X)
MIV(1,3)=(-(81.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**
. 4*M30**2)-(9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2
. *M30*J30Y)+9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2
. *M30*J30Z+81.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**4*
. M30**2+9.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**2*M30*
. J30Y)/(729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**
. 6*M30**3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P
. **4*M30**2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2)
. ))**2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*
. Y*M30**2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*
. J30Y)+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)+9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30-(9.0D0*DSIN(DBLE
. (Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(DBLE(Q3))**4*P**
. 2*M30*J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y
. *J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*J30-(DSIN(DBLE(Q3)
. ))**4*Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3))**4*J30Y**2*J30X
. )+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X-(729.0D0*DSIN(DBLE(
. Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DSIN(
. DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y)-(
. 729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(81.0D0*DSIN(
. DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN(DBLE(Q3))**
. 2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))**2*P**4*M30
. **2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J10Y)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J30X)-(9.0D0*DSIN
. (DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(Q3))**2*P**4*

```

```

. M30*M10*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30Y
. *J30)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30X*J30)+
. 9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2-(9.0D0*DSIN(
. DBLE(Q3))**2*P**2*M30*J30Y*J10Y)+9.0D0*DSIN(DBLE(Q3))
. **2*P**2*M30*J30Z*J10Y+9.0D0*DSIN(DBLE(Q3))**2*P**2*
. M30*J30Z*J30X-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J10Y*
. J30X)-(DSIN(DBLE(Q3))**2*P**2*J30Y*M10*J30X)+DSIN(DBLE
. (Q3))**2*P**2*J30Z*M10*J30X-(DSIN(DBLE(Q3))**2*Y*J30Y*
. J30X*J30)+DSIN(DBLE(Q3))**2*J30Y**2*J30X-(DSIN(DBLE(Q3
. ))**2*J30Y*J10Y*J30X)+DSIN(DBLE(Q3))**2*J30Z*J10Y*J30X
. -(729.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**6*M30
. **3)-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0*
. P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2*
. M30*J30Y*J30X+9.0D0*P**2*M30*J10Y*J30X+P**2*J30Y*M10*
. J30X+J30Y*J10Y*J30X)
MIV(2,2)=(-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*
. P**4*M30**2)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2)-(
. 9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30)+9.0D0*DSIN(
. DBLE(Q3))**2*P**2*M30*J30Z-(9.0D0*DSIN(DBLE(Q3))**2*P
. **2*M30*J30X)-(DSIN(DBLE(Q3))**2*Y*J30X*J30)+DSIN(DBLE
. (Q3))**2*J30Z*J30X+162.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2
. ))*P**4*M30**2+18.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P
. **2*M30*J30X+162.0D0*P**4*M30**2+9.0D0*P**4*M30*M10+
. 9.0D0*P**2*M30*J30Y+9.0D0*P**2*M30*J10Y+18.0D0*P**2*
. M30*J30X+P**2*M10*J30X+J30Y*J30X+J10Y*J30X)/(729.0D0*
. DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30**3+81.0D0
. *DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y-
. (81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**
. 2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*M30**2*J30-(
. 81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)+9.0D0*DSIN(
. DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(9.0D0*DSIN(DBLE(Q3))
. **4*P**2*Y*M30*J30Z*J30)+9.0D0*DSIN(DBLE(Q3))**4*P**2*
. Y*M30*J30X*J30-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y
. **2)+9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0
. *DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))
. **4*Y*J30Y*J30X*J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30
. )-(DSIN(DBLE(Q3))**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*
. J30Y*J30Z*J30X-(729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2
. ))**2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE
. (Q2))**2*P**4*M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*
. P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10

```



```

. )-(81.0D0*DSIN(DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*
. DSIN(DBLE(Q3))**2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P
. **4*M30**2*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*
. J30Y*M10)+9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(
. 9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN
. (DBLE(Q3))**2*P**2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3
. ))**2*P**2*Y*M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**
. 2*M30*J30Y**2-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*
. J10Y)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y+9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0*DSIN(DBLE
. (Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))**2*P**2*
. J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*M10*J30X-(
. DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE(Q3))**2*
. J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*J30X)+DSIN(
. DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(DBLE(Q3))**2
. *DCOS(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DCOS(DBLE(Q3))
. **2*DCOS(DBLE(Q2))**2*P**4*M30**2*J30X)+729.0D0*P**6*
. M30**3+81.0D0*P**6*M30**2*M10+81.0D0*P**4*M30**2*J30Y+
. 81.0D0*P**4*M30**2*J10Y+81.0D0*P**4*M30**2*J30X+9.0D0*
. P**4*M30*J30Y*M10+9.0D0*P**4*M30*M10*J30X+9.0D0*P**2*
. M30*J30Y*J10Y+9.0D0*P**2*M30*J30Y*J30X+9.0D0*P**2*M30*
. J10Y*J30X+P**2*J30Y*M10*J30X+J30Y*J10Y*J30X)
MIV(2,3)=(81.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**4*
. M30**2+9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2*M30
. *J30Y-(9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2*M30
. *J30Z)-(81.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*DCOS(DBLE
. (Q3))*DCOS(DBLE(Q2))*P**4*M30**2)-(81.0D0*DSIN(DBLE(Q3
. ))*DSIN(DBLE(Q2))*P**4*M30**2)-(9.0D0*DSIN(DBLE(Q3))*
. DSIN(DBLE(Q2))*P**2*M30*J30Y)/(729.0D0*DSIN(DBLE(Q3))
. **4*DSIN(DBLE(Q2))**2*P**6*M30**3+81.0D0*DSIN(DBLE(Q3)
. )**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y-(81.0D0*DSIN(
. DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Z)+
. 81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*M30**2*J30-(81.0D0*
. DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)+9.0D0*DSIN(DBLE(Q3
. ))**4*P**2*Y*M30*J30Y*J30-(9.0D0*DSIN(DBLE(Q3))**4*P**
. 2*Y*M30*J30Z*J30)+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*
. J30X*J30-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN
. (DBLE(Q3))**4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*
. J30Y*J30X*J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(
. DSIN(DBLE(Q3))**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y
. *J30Z*J30X-(729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**
. 2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))

```

```

. ) **2*P**4*M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6
. *M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*
. DSIN(DBLE(Q3))**2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P
. **4*M30**2*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*
. J30Y*M10)+9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(
. 9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN
. (DBLE(Q3))**2*P**2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3
. ))**2*P**2*Y*M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**
. 2*M30*J30Y**2-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*
. J10Y)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y+9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0*DSIN(DBLE
. (Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))**2*P**2*
. J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*M10*J30X-(
. DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE(Q3))**2*
. J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*J30X)+DSIN(
. DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(DBLE(Q3))**2
. *DCOS(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DCOS(DBLE(Q3))
. **2*DCOS(DBLE(Q2))**2*P**4*M30**2*J30X)+729.0D0*P**6*
. M30**3+81.0D0*P**6*M30**2*M10+81.0D0*P**4*M30**2*J30Y+
. 81.0D0*P**4*M30**2*J10Y+81.0D0*P**4*M30**2*J30X+9.0D0*
. P**4*M30*J30Y*M10+9.0D0*P**4*M30*M10*J30X+9.0D0*P**2*
. M30*J30Y*J10Y+9.0D0*P**2*M30*J30Y*J30X+9.0D0*P**2*M30*
. J10Y*J30X+P**2*J30Y*M10*J30X+J30Y*J10Y*J30X)
MIV(3,3)=(9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30-(9.0D0
. *DSIN(DBLE(Q3))**4*P**2*M30*J30Y)+DSIN(DBLE(Q3))**4*Y*
. J30Y*J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30)-(DSIN(DBLE(Q3)
. ))**4*J30Y**2)+DSIN(DBLE(Q3))**4*J30Y*J30Z-(81.0D0*DSIN
. (DBLE(Q3))**2*P**4*M30**2)-(9.0D0*DSIN(DBLE(Q3))**2*P
. **4*M30*M10)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30)+
. 9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z-(9.0D0*DSIN(DBLE
. (Q3))**2*P**2*M30*J10Y)-(DSIN(DBLE(Q3))**2*P**2*J30Y*
. M10)+DSIN(DBLE(Q3))**2*P**2*J30Z*M10-(DSIN(DBLE(Q3))**
. 2*Y*J30Y*J30)+DSIN(DBLE(Q3))**2*J30Y**2-(DSIN(DBLE(Q3)
. ))**2*J30Y*J10Y)+DSIN(DBLE(Q3))**2*J30Z*J10Y-(81.0D0*
. DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*M30**2)+
. 81.0D0*P**4*M30**2+9.0D0*P**4*M30*M10+9.0D0*P**2*M30*
. J30Y+9.0D0*P**2*M30*J10Y+P**2*J30Y*M10+J30Y*J10Y)/(
. 729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30**
. 3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*M30
. **2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P
. **4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*M30**
. 2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)+

```

```

. 9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)+9.0D0*DSIN(DBLE
. (Q3))**4*P**2*Y*M30*J30X*J30-(9.0D0*DSIN(DBLE(Q3))**4*
. P**2*M30*J30Y**2)+9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*
. J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y*J30X)
. +DSIN(DBLE(Q3))**4*Y*J30Y*J30X*J30-(DSIN(DBLE(Q3))**4*
. Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3))**4*J30Y**2*J30X)+DSIN
. (DBLE(Q3))**4*J30Y*J30Z*J30X-(729.0D0*DSIN(DBLE(Q3))**
. 2*DSIN(DBLE(Q2))**2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3)
. )**2*DSIN(DBLE(Q2))**2*P**4*M30**2*J30Y)-(729.0D0*DSIN
. (DBLE(Q3))**2*P**6*M30**3)-(81.0D0*DSIN(DBLE(Q3))**2*P
. **6*M30**2*M10)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*Y*M30**
. 2*J30)+81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J30Z-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*J10Y)-(81.0D0*
. DSIN(DBLE(Q3))**2*P**4*M30**2*J30X)-(9.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30*J30Y*M10)+9.0D0*DSIN(DBLE(Q3))**2*P**
. 4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*M10*
. J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30Y*J30)-(
. 9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30X*J30)+9.0D0*
. DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2-(9.0D0*DSIN(DBLE(Q3)
. ))**2*P**2*M30*J30Y*J10Y)+9.0D0*DSIN(DBLE(Q3))**2*P**2
. *M30*J30Z*J10Y+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*
. J30X-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J10Y*J30X)-(
. DSIN(DBLE(Q3))**2*P**2*J30Y*M10*J30X)+DSIN(DBLE(Q3))**
. 2*P**2*J30Z*M10*J30X-(DSIN(DBLE(Q3))**2*Y*J30Y*J30X*
. J30)+DSIN(DBLE(Q3))**2*J30Y**2*J30X-(DSIN(DBLE(Q3))**2
. *J30Y*J10Y*J30X)+DSIN(DBLE(Q3))**2*J30Z*J10Y*J30X-(
. 729.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**6*M30**
. 3)-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0
. *P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2
. *M30*J30Y*J30X+9.0D0*P**2*M30*J10Y*J30X+P**2*J30Y*M10*
. J30X+J30Y*J10Y*J30X)
DO 25005 J=1,3
    DO 25006 K=J+1,3
        M(K,J)=M(J,K)
        MIV(K,J)=MIV(J,K)
25006     CONTINUE
25005 CONTINUE

```

Contents of file m2.f:

```

M(1,1)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(
. Q3))**2*Y*J30)+DSIN(DBLE(Q3))**2*J30Z+18.0D0*DCOS(DBLE
. (Q3))*DCOS(DBLE(Q2))*P**2*M30+18.0D0*P**2*M30+P**2*M10
. +J30Y+J10Y(1,2)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (
. DSIN(DBLE(Q3))**2*J30Y)+DSIN(DBLE(Q3))**2*J30Z+9.0D0*
. DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**2*M30+9.0D0*P**2*M30+
. J30Y(1,3)=- (9.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**2*
. M30)
M(2,2)=- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30) - (DSIN(DBLE(
. Q3))**2*J30Y)+DSIN(DBLE(Q3))**2*J30Z+9.0D0*P**2*M30+
. J30Y
M(2,3)=0.0D0
M(3,3)=9.0D0*P**2*M30+J30X
T1=- (81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2) - (9.0D0*DSIN(
. DBLE(Q3))**2*P**2*M30*J30Y)+9.0D0*DSIN(DBLE(Q3))**2*P
. **2*M30*J30Z- (9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30X) - (
. DSIN(DBLE(Q3))**2*J30Y*J30X)+DSIN(DBLE(Q3))**2*J30Z*
. J30X+81.0D0*P**4*M30**2+9.0D0*P**2*M30*J30Y+9.0D0*P**2
. *M30*J30X+J30Y*J30X
T0=729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30
. **3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y- (81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**
. 2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*
. M30**2*J30- (81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)
. +9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30- (9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)
T0=T0+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30- (
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(
. DBLE(Q3))**4*P**2*M30*J30Y*J30Z- (9.0D0*DSIN(DBLE(Q3))
. **4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*
. J30- (DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)- (DSIN(DBLE(Q3)
. )**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X- (
. 729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**
. 3)
T0=T0- (81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y)- (729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)- (
. 81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)- (81.0D0*DSIN
. (DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30**2*J30Z- (81.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30**2*J10Y)- (81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*
. J30X)
T0=T0- (9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0

```

```

. *DSIN (DBLE (Q3) ) **2*P**4*M30*J30Z*M10-(9.0D0*DSIN (DBLE (
. Q3) ) **2*P**4*M30*M10*J30X)-(9.0D0*DSIN (DBLE (Q3) ) **2*P
. **2*Y*M30*J30Y*J30)-(9.0D0*DSIN (DBLE (Q3) ) **2*P**2*Y*
. M30*J30X*J30)+9.0D0*DSIN (DBLE (Q3) ) **2*P**2*M30*J30Y**2
. -(9.0D0*DSIN (DBLE (Q3) ) **2*P**2*M30*J30Y*J10Y)+9.0D0*
. DSIN (DBLE (Q3) ) **2*P**2*M30*J30Z*J10Y
T0=T0+9.0D0*DSIN (DBLE (Q3) ) **2*P**2*M30*J30Z*J30X-(9.0D0
. *DSIN (DBLE (Q3) ) **2*P**2*M30*J10Y*J30X)-(DSIN (DBLE (Q3) )
. **2*P**2*J30Y*M10*J30X)+DSIN (DBLE (Q3) ) **2*P**2*J30Z*
. M10*J30X-(DSIN (DBLE (Q3) ) **2*Y*J30Y*J30X*J30)+DSIN (DBLE
. (Q3) ) **2*J30Y**2*J30X-(DSIN (DBLE (Q3) ) **2*J30Y*J10Y*
. J30X)+DSIN (DBLE (Q3) ) **2*J30Z*J10Y*J30X-(729.0D0*DCOS (
. DBLE (Q3) ) **2*DCOS (DBLE (Q2) ) **2*P**6*M30**3)
T0=T0-(81.0D0*DCOS (DBLE (Q3) ) **2*DCOS (DBLE (Q2) ) **2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0*
. P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2*
; M30*J30Y*J30X
MIV (1,1)=T1/(T0+9.0D0*P**2*M30*J10Y*J30X+P**2*J30Y*M10*
. J30X+J30Y*J10Y*J30X)
T0=81.0D0*DSIN (DBLE (Q3) ) **2*P**4*M30**2+9.0D0*DSIN (DBLE
. (Q3) ) **2*P**2*M30*J30Y-(9.0D0*DSIN (DBLE (Q3) ) **2*P**2*
. M30*J30Z)+9.0D0*DSIN (DBLE (Q3) ) **2*P**2*M30*J30X+DSIN (
. DBLE (Q3) ) **2*J30Y*J30X-(DSIN (DBLE (Q3) ) **2*J30Z*J30X)-(
. 81.0D0*DCOS (DBLE (Q3) ) *DCOS (DBLE (Q2) ) *P**4*M30**2)-(
. 9.0D0*DCOS (DBLE (Q3) ) *DCOS (DBLE (Q2) ) *P**2*M30*J30X)-(
. 81.0D0*P**4*M30**2)-(9.0D0*P**2*M30*J30Y)
T1=729.0D0*DSIN (DBLE (Q3) ) **4*DSIN (DBLE (Q2) ) **2*P**6*M30
. **3+81.0D0*DSIN (DBLE (Q3) ) **4*DSIN (DBLE (Q2) ) **2*P**4*
. M30**2*J30Y-(81.0D0*DSIN (DBLE (Q3) ) **4*DSIN (DBLE (Q2) ) **
. 2*P**4*M30**2*J30Z)+81.0D0*DSIN (DBLE (Q3) ) **4*P**4*Y*
. M30**2*J30-(81.0D0*DSIN (DBLE (Q3) ) **4*P**4*M30**2*J30Y)
. +9.0D0*DSIN (DBLE (Q3) ) **4*P**2*Y*M30*J30Y*J30-(9.0D0*
. DSIN (DBLE (Q3) ) **4*P**2*Y*M30*J30Z*J30)
T1=T1+9.0D0*DSIN (DBLE (Q3) ) **4*P**2*Y*M30*J30X*J30-(
. 9.0D0*DSIN (DBLE (Q3) ) **4*P**2*M30*J30Y**2)+9.0D0*DSIN (
. DBLE (Q3) ) **4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN (DBLE (Q3) )
. **4*P**2*M30*J30Y*J30X)+DSIN (DBLE (Q3) ) **4*Y*J30Y*J30X*
. J30-(DSIN (DBLE (Q3) ) **4*Y*J30Z*J30X*J30)-(DSIN (DBLE (Q3)
. ) **4*J30Y**2*J30X)+DSIN (DBLE (Q3) ) **4*J30Y*J30Z*J30X-(
. 729.0D0*DSIN (DBLE (Q3) ) **2*DSIN (DBLE (Q2) ) **2*P**6*M30**
. 3)
T1=T1-(81.0D0*DSIN (DBLE (Q3) ) **2*DSIN (DBLE (Q2) ) **2*P**4*

```

```

. M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN
. (DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*
. J30X)
T1=T1-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0
. *DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P
. **2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*
. M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2
. -(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*J10Y)+9.0D0*
. DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y
T1=T1+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))
. **2*P**2*J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*
. M10*J30X-(DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE
. (Q3))**2*J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*
. J30X)+DSIN(DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(
. DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**6*M30**3)
T1=T1-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0*
. P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2*
. M30*J30Y*J30X
MIV(1,2)=(T0-(9.0D0*P**2*M30*J30X)-(J30Y*J30X))/(T1+
. 9.0D0*P**2*M30*J10Y*J30X+P**2*J30Y*M10*J30X+J30Y*J10Y*
. J30X)
T0=729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30
. **3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**
. 2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*
. M30**2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)
. +9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)
T0=T0+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30-(
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(
. DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))
. **4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*
. J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3)
. )**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X-(
. 729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**
. 3)

```

```

T0=T0-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN
. (DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*
. J30X)
T0=T0-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0
. *DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P
. **2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*
. M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2
. -(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*J10Y)+9.0D0*
. DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y
T0=T0+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))
. **2*P**2*J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*
. M10*J30X-(DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE
. (Q3))**2*J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*
. J30X)+DSIN(DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(
. DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**6*M30**3)
T0=T0-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0*
. P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2*
. M30*J30Y*J30X
MIV(1,3)=(-(81.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**
. 4*M30**2)-(9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2
. *M30*J30Y)+9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2
. *M30*J30Z+81.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**4*
. M30**2+9.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*P**2*M30*
. J30Y)/(T0+9.0D0*P**2*M30*J10Y*J30X+P**2*J30Y*M10*J30X+
. J30Y*J10Y*J30X)
T0=-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*
. M30**2)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2)-(9.0D0*
. DSIN(DBLE(Q3))**2*P**2*Y*M30*J30)+9.0D0*DSIN(DBLE(Q3))
. **2*P**2*M30*J30Z-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*
. J30X)-(DSIN(DBLE(Q3))**2*Y*J30X*J30)+DSIN(DBLE(Q3))**2
. *J30Z*J30X+162.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**4*
. M30**2+18.0D0*DCOS(DBLE(Q3))*DCOS(DBLE(Q2))*P**2*M30*
. J30X+162.0D0*P**4*M30**2
T1=729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30
. **3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*

```

```

. M30**2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**
. 2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*
. M30**2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)
. +9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)
T1=T1+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30-(
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(
. DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))
. **4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*
. J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3)
. )**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X-(
. 729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**
. 3)
T1=T1-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN
. (DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*
. J30X)
T1=T1-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0
. *DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P
. **2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*
. M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2
. -(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*J10Y)+9.0D0*
. DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y
T1=T1+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))
. **2*P**2*J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*
. M10*J30X-(DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE
. (Q3))**2*J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*
. J30X)+DSIN(DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(
. DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**6*M30**3)
T1=T1-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0*
. P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2*
. M30*J30Y*J30X
MIV(2,2)=(T0+9.0D0*P**4*M30*M10+9.0D0*P**2*M30*J30Y+
. 9.0D0*P**2*M30*J10Y+18.0D0*P**2*M30*J30X+P**2*M10*J30X
. +J30Y*J30X+J10Y*J30X)/(T1+9.0D0*P**2*M30*J10Y*J30X+P**
. 2*J30Y*M10*J30X+J30Y*J10Y*J30X)

```



```

T0=729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30
. **3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**
. 2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*
. M30**2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)
. +9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)
T0=T0+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30-(
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(
. DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))
. **4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*
. J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3)
. )**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X-(
. 729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**
. 3)
T0=T0-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN
. (DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*
. J30X)
T0=T0-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0
. *DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(
. Q3))**2*P**4*M30*M10*J30X)-(9.0D0*DSIN(DBLE(Q3))**2*P
. **2*Y*M30*J30Y*J30)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*
. M30*J30X*J30)+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y**2
. -(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Y*J10Y)+9.0D0*
. DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J10Y
T0=T0+9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z*J30X-(9.0D0
. *DSIN(DBLE(Q3))**2*P**2*M30*J10Y*J30X)-(DSIN(DBLE(Q3))
. **2*P**2*J30Y*M10*J30X)+DSIN(DBLE(Q3))**2*P**2*J30Z*
. M10*J30X-(DSIN(DBLE(Q3))**2*Y*J30Y*J30X*J30)+DSIN(DBLE
. (Q3))**2*J30Y**2*J30X-(DSIN(DBLE(Q3))**2*J30Y*J10Y*
. J30X)+DSIN(DBLE(Q3))**2*J30Z*J10Y*J30X-(729.0D0*DCOS(
. DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**6*M30**3)
T0=T0-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2))**2*P**4*
. M30**2*J30X)+729.0D0*P**6*M30**3+81.0D0*P**6*M30**2*
. M10+81.0D0*P**4*M30**2*J30Y+81.0D0*P**4*M30**2*J10Y+
. 81.0D0*P**4*M30**2*J30X+9.0D0*P**4*M30*J30Y*M10+9.0D0*
. P**4*M30*M10*J30X+9.0D0*P**2*M30*J30Y*J10Y+9.0D0*P**2*
. M30*J30Y*J30X
MIV(2,3)=(81.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**4*
. M30**2+9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2*M30

```

```

. *J30Y-(9.0D0*DSIN(DBLE(Q3))**3*DSIN(DBLE(Q2))*P**2*M30
. *J30Z)-(81.0D0*DSIN(DBLE(Q3))*DSIN(DBLE(Q2))*DCOS(DBLE
. (Q3))*DCOS(DBLE(Q2))*P**4*M30**2)-(81.0D0*DSIN(DBLE(Q3)
. ))*DSIN(DBLE(Q2))*P**4*M30**2)-(9.0D0*DSIN(DBLE(Q3))*
. DSIN(DBLE(Q2))*P**2*M30*J30Y)/(T0+9.0D0*P**2*M30*J10Y
. *J30X+P**2*J30Y*M10*J30X+J30Y*J10Y*J30X)
T0=9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30-(9.0D0*DSIN(
. DBLE(Q3))**4*P**2*M30*J30Y)+DSIN(DBLE(Q3))**4*Y*J30Y*
. J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30)-(DSIN(DBLE(Q3))**4*
. J30Y**2)+DSIN(DBLE(Q3))**4*J30Y*J30Z-(81.0D0*DSIN(DBLE
. (Q3))**2*P**4*M30**2)-(9.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30*M10)-(9.0D0*DSIN(DBLE(Q3))**2*P**2*Y*M30*J30)+
. 9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J30Z
T0=T0-(9.0D0*DSIN(DBLE(Q3))**2*P**2*M30*J10Y)-(DSIN(
. DBLE(Q3))**2*P**2*J30Y*M10)+DSIN(DBLE(Q3))**2*P**2*
. J30Z*M10-(DSIN(DBLE(Q3))**2*Y*J30Y*J30)+DSIN(DBLE(Q3))
. **2*J30Y**2-(DSIN(DBLE(Q3))**2*J30Y*J10Y)+DSIN(DBLE(Q3)
. ))**2*J30Z*J10Y-(81.0D0*DCOS(DBLE(Q3))**2*DCOS(DBLE(Q2)
. ))**2*P**4*M30**2)+81.0D0*P**4*M30**2+9.0D0*P**4*M30*
. M10+9.0D0*P**2*M30*J30Y
T1=729.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**6*M30
. **3+81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y-(81.0D0*DSIN(DBLE(Q3))**4*DSIN(DBLE(Q2))**
. 2*P**4*M30**2*J30Z)+81.0D0*DSIN(DBLE(Q3))**4*P**4*Y*
. M30**2*J30-(81.0D0*DSIN(DBLE(Q3))**4*P**4*M30**2*J30Y)
. +9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Y*J30-(9.0D0*
. DSIN(DBLE(Q3))**4*P**2*Y*M30*J30Z*J30)
T1=T1+9.0D0*DSIN(DBLE(Q3))**4*P**2*Y*M30*J30X*J30-(
. 9.0D0*DSIN(DBLE(Q3))**4*P**2*M30*J30Y**2)+9.0D0*DSIN(
. DBLE(Q3))**4*P**2*M30*J30Y*J30Z-(9.0D0*DSIN(DBLE(Q3))
. **4*P**2*M30*J30Y*J30X)+DSIN(DBLE(Q3))**4*Y*J30Y*J30X*
. J30-(DSIN(DBLE(Q3))**4*Y*J30Z*J30X*J30)-(DSIN(DBLE(Q3)
. ))**4*J30Y**2*J30X)+DSIN(DBLE(Q3))**4*J30Y*J30Z*J30X-(
. 729.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**6*M30**
. 3)
T1=T1-(81.0D0*DSIN(DBLE(Q3))**2*DSIN(DBLE(Q2))**2*P**4*
. M30**2*J30Y)-(729.0D0*DSIN(DBLE(Q3))**2*P**6*M30**3)-(
. 81.0D0*DSIN(DBLE(Q3))**2*P**6*M30**2*M10)-(81.0D0*DSIN
. (DBLE(Q3))**2*P**4*Y*M30**2*J30)+81.0D0*DSIN(DBLE(Q3))
. **2*P**4*M30**2*J30Z-(81.0D0*DSIN(DBLE(Q3))**2*P**4*
. M30**2*J10Y)-(81.0D0*DSIN(DBLE(Q3))**2*P**4*M30**2*
. J30X)
T1=T1-(9.0D0*DSIN(DBLE(Q3))**2*P**4*M30*J30Y*M10)+9.0D0
. *DSIN(DBLE(Q3))**2*P**4*M30*J30Z*M10-(9.0D0*DSIN(DBLE(

```

```

. Q3) ) **2 * P **4 * M30 * M10 * J30X) - (9.0D0 * DSIN (DBLE (Q3) ) **2 * P
. **2 * Y * M30 * J30Y * J30) - (9.0D0 * DSIN (DBLE (Q3) ) **2 * P **2 * Y *
. M30 * J30X * J30) + 9.0D0 * DSIN (DBLE (Q3) ) **2 * P **2 * M30 * J30Y **2
. - (9.0D0 * DSIN (DBLE (Q3) ) **2 * P **2 * M30 * J30Y * J10Y) + 9.0D0 *
. DSIN (DBLE (Q3) ) **2 * P **2 * M30 * J30Z * J10Y
T1=T1+9.0D0 * DSIN (DBLE (Q3) ) **2 * P **2 * M30 * J30Z * J30X - (9.0D0
. * DSIN (DBLE (Q3) ) **2 * P **2 * M30 * J10Y * J30X) - (DSIN (DBLE (Q3) )
. **2 * P **2 * J30Y * M10 * J30X) + DSIN (DBLE (Q3) ) **2 * P **2 * J30Z *
. M10 * J30X - (DSIN (DBLE (Q3) ) **2 * Y * J30Y * J30X * J30) + DSIN (DBLE
. (Q3) ) **2 * J30Y **2 * J30X - (DSIN (DBLE (Q3) ) **2 * J30Y * J10Y *
. J30X) + DSIN (DBLE (Q3) ) **2 * J30Z * J10Y * J30X - (729.0D0 * DCOS (
. DBLE (Q3) ) **2 * DCOS (DBLE (Q2) ) **2 * P **6 * M30 **3)
T1=T1 - (81.0D0 * DCOS (DBLE (Q3) ) **2 * DCOS (DBLE (Q2) ) **2 * P **4 *
. M30 **2 * J30X) + 729.0D0 * P **6 * M30 **3 + 81.0D0 * P **6 * M30 **2 *
. M10 + 81.0D0 * P **4 * M30 **2 * J30Y + 81.0D0 * P **4 * M30 **2 * J10Y +
. 81.0D0 * P **4 * M30 **2 * J30X + 9.0D0 * P **4 * M30 * J30Y * M10 + 9.0D0 *
. P **4 * M30 * M10 * J30X + 9.0D0 * P **2 * M30 * J30Y * J10Y + 9.0D0 * P **2 *
. M30 * J30Y * J30X
MIV (3, 3) = (T0 + 9.0D0 * P **2 * M30 * J10Y + P **2 * J30Y * M10 + J30Y *
. J10Y) / (T1 + 9.0D0 * P **2 * M30 * J10Y * J30X + P **2 * J30Y * M10 * J30X +
. J30Y * J10Y * J30X)
DO 25007 J=1, 3
    DO 25008 K=J+1, 3
        M (K, J) =M (J, K)
        MIV (K, J) =MIV (J, K)
25008     CONTINUE
25007 CONTINUE

```

16.25 GNUPLOT: Display of functions and surfaces

This package is an interface to the popular GNUPLOT package. It allows you to display functions in 2D and surfaces in 3D on a variety of output devices including X terminals, PC monitors, and postscript and Latex printer files.

NOTE: The GNUPLOT package may not be included in all versions of REDUCE.

Author: Herbert Melenk.

16.25.1 Introduction

The GNUPLOT system provides easy to use graphics output for curves or surfaces which are defined by formulas and/or data sets. GNUPLOT supports a variety of output devices such as VGA screen, postscript, pic \TeX , MS Windows. The REDUCE GNUPLOT package lets one use the GNUPLOT graphical output directly from inside REDUCE, either for the interactive display of curves/surfaces or for the production of pictures on paper.

REDUCE supports GNUPLOT 3.4 (or higher). For DOS, Windows, Windows 95, Windows NT, OS/2 and Unix versions of REDUCE GNUPLOT binaries are delivered together with REDUCE ²³. However, this is a basic set only. If you intend to use more facilities of the GNUPLOT system you should pick up the full GNUPLOT file tree from a server, e.g.

- dartmouth.edu (129.170.16.4)
- monu1.cc.monash.edu.au (130.194.1.101)
- irisa.irisa.fr (131.254.2.3)

16.25.2 Command PLOT

Under REDUCE GNUPLOT is used as graphical output server, invoked by the command `PLOT (. . .)`. This command can have a variable number of parameters:

- A functions to plot; a function can be
 - an expression with one unknown, e.g. $u * \sin(u) * *2$
 - a list of expressions with one (identical) unknown, e.g. $\{\sin(u), \cos(u)\}$

²³The GNUPLOT developers have agreed that GNUPLOT binaries can be distributed together with REDUCE. As GNUPLOT is a package distributed without cost, the GNUPLOT support of REDUCE also is an add-on to the REDUCE kernel system without charge. We recommend fetching the full GNUPLOT system by anonymous FTP from a file server.

- an expression with two unknowns, e.g. $u * \sin(u) ** 2 + \text{sqrt}(v)$
 - a parametric expression of the form $\text{point}(< u >, < v >)$ or $\text{point}(< u >, < v >, < w >)$ where $< u >$, $< v >$ and $< w >$ are expressions which depend of one or two parameters; if there is one parameter, the object describes a curve in the plane (only $< u >$ and $< v >$) or in the 3D space; if there are two parameters, the object describes a surface in 3D. The parameters are treated as independent variables. Example:
 $\text{point}(\text{ sint}, \text{ cost}, t/10)$
 - an equation with a symbol on the left-hand side and an expression with one or two unknowns on the right-hand side, e.g.
 $\text{dome} = 1/(x ** 2 + y ** 2)$
 - an equation with an expression on the left-hand side and a zero on right-hand side describing implicitly a one dimensional variety in the plane (implicitly given curve), e.g.
 $x^3 + x * y^2 - 9x = 0$, or a two dimensional surface in the 3 dimensional Euclidean space,
 - an equation with an expression in two variables on the left-hand side and a list of numbers on the right-hand side; the contour lines corresponding to the given values are drawn, e.g.
 $x^3 - y^2 + x * y = \{-2, -1, 0, 1, 2\}$,
 - a list of points in 2 or 3 dimensions e.g.
 $\{\{0, 0\}, \{0, 1\}, \{1, 1\}\}$ representing a curve,
 - a list of lists of points in 2 or 3 dimensions e.g.
 $\{\{\{0, 0\}, \{0, 1\}, \{1, 1\}\}, \{\{0, 0\}, \{0, 1\}, \{1, 1\}\}\}$ representing a family of curves.
- A range for a variable; this has the form
 $\text{variable} = (\text{lower_bound} .. \text{upper_bound})$ where lower_bound and upper_bound must be expressions which evaluate to numbers. If no range is specified the default ranges for independent variables are $(-10 .. 10)$ and the range for the dependent variable is set to maximum number of the GNUPLOT executable (using double floats on most IEEE machines and single floats under DOS). Additionally the number of interval subdivisions can be assigned as a formal quotient
 $\text{variable} = (\text{lower_bound} .. \text{upper_bound}) / < it >$ where $< it >$ is a positive integer. E.g. $(1..5)/30$ means the interval from 1 to 5 subdivided into 30 pieces of equal size. A subdivision parameter overrides the value of the variable *points* for this variable.
 - A plot option, either as fixed keyword, e.g. `hidden3d` or as equation e.g. `term=pictex`; free texts such as titles and labels should be enclosed in string quotes.

Please note that a blank has to be inserted between a number and a dot - otherwise the REDUCE translator will be misled.

If a function is given as an equation the left-hand side is mainly used as a label for the axis of the dependent variable.

In two dimensions, `PLOT` can be called with more than one explicit function; all curves are drawn in one picture. However, all these must use the same independent variable name. One of the functions can be a point set or a point set list. Normally all functions and point sets are plotted by lines. A point set is drawn by points only if functions and the point set are drawn in one picture.

In three dimensions only one surface can be shown per call. Also an implicitly given curve must be the sole object for one picture.

The functional expressions are evaluated in `rounded` mode. This is done automatically - it is not necessary to turn on rounded mode explicitly.

Examples:

```

plot(cos x);
plot(s=sin phi,phi=(-3 .. 3));
plot(sin phi,cos phi,phi=(-3 .. 3));
plot (cos sqrt(x**2 + y**2),x=(-3 .. 3),y=(-3 .. 3),hidden3d);
plot {{0,0},{0,1},{1,1},{0,0},{1,0},{0,1},{0.5,1.5},{1,1},{1,0}};

% parametric: screw

on rounded;
w:=for j:=1:200 collect {1/j*sin j,1/j*cos j,j/200}$
plot w;

% parametric: globe
dd:=pi/15$
w:=for u:=dd step dd until pi-dd collect
  for v:=0 step dd until 2pi collect
    {sin(u)*cos(v), sin(u)*sin(v), cos(u)}$
plot w;

% implicit: superposition of polynomials
plot((x^2+y^2-9)*x*y=0);

```

Piecewise defined functions: A composed graph can be defined by a rule-based operator. In that case each rule must contain a clause which restricts the rule application to numeric arguments, e.g.

```

operator my_step1;
let {my_step1(~x) => -1 when numberp x and x<-pi/2,
    my_step1(~x) => 1 when numberp x and x>pi/2,
    my_step1(~x) => sin x
    when numberp x and -pi/2<=x and x<=pi/2};
plot(my_step2(x));

```

Of course, such a rule may call a procedure:

```

procedure my_step3(x);
  if x<-1 then -1 else if x>1 then 1 else x;
operator my_step2;
let my_step2(~x) => my_step3(x) when numberp x;
plot(my_step2(x));

```

The direct use of a procedure with a numeric *if* clause is impossible.

Plot options: The following plot options are supported in the `PLOT` command:

- *points* = < integer >: the number of unconditionally computed data points; for a grid *points*² grid points are used. The default value is 20. The value of *points* is used only for variables for which no individual interval subdivision has been specified in the range specification.
- *refine* = < integer >: the maximum depth of adaptive interval intersections. The default is 8. A value 0 switches any refinement off. Note that a high value may increase the computing time significantly.

The following additional GNUPLOT options are supported in the `PLOT` command:

- *title* = *name*: the title (string) is put on top of the picture.
- axes labels: *xlabel* = "*text1*", *ylabel* = "*text2*", and for surfaces *zlabel* = "*text3*". If omitted the axes are labeled by the independent and dependent variable names from the expression. Note that the axes names *xlabel*, *ylabel* and *zlabel* here are used in the usual sense, *x* for the horizontal and *y* for the vertical axis under 2-d and *z* for the perpendicular axis under 3-d – these names do not refer to the variable names used in the expressions.

```
plot(1,X,(4*X**2-1)/2,(X*(12*X**2-5))/3,
      x=(-1 .. 1), ylabel="L(x,n)",
      title="Legendre Polynomials");
```

- *terminal* = *name*: prepare output for device type *name*. Every installation uses a default terminal as output device; some installations support additional devices such as printers; consult the original GNUPLOT documentation or the GNUPLOT Help for details.
- *output* = "*filename*": redirect the output to a file.
- *size* = "*s_x,s_y*": rescale the graph (not the window) where *s_x* and *s_y* are scaling factors for the size in x or y direction. Defaults are *s_x* = 1, *s_y* = 1. Note that scaling factors greater than one will often cause the picture to be too big for the window.

```
plot(1/(x**2+y**2),x=(0.1 .. 5),
      y=(0.1 .. 5), size="0.7,1");
```

- *view* = "*r_x,r_z*": set viewpoint for 3 dimensions by turning the object around the x or z axis; the values are degrees (integers). Defaults are *r_x* = 60, *r_z* = 30.


```
plot(1/(x**2+y**2),x=(0.1 .. 5),
      y=(0.1 .. 5), view="30,130");
```

- *contour* resp *nocontour*: in 3 dimensions an additional contour map is drawn (default: *nocontour*) Note that *contour* is an option which is executed by GNUPLOT by interpolating the precomputed function values. If you want to draw contour lines of a delicate formula, you better use the contour form of the REDUCE PLOT command.
- *surface* resp *nosurface*: in 3 dimensions the surface is drawn resp suppressed (default: *surface*).
- *hidden3d*: hidden line removal in 3 dimensions.

16.25.3 Paper output

The following example works for a postscript printer. If your printer uses a different communication, please find the correct setting for the *terminal* variable in the *Gnuplot* documentation.

For a postscript printer, add the options *terminal = postscript* and *output = "filename"* to your plot command, e.g.

```
plot(sin x,x=(0 .. 10),terminal=postscript,output="sin.ps");
```

16.25.4 Mesh generation for implicit curves

The basic mesh for finding an implicitly given curve, the x, y plane is subdivided into an initial set of triangles. Those triangles which have an explicit zero point or which have two points with different signs are refined by subdivision. A further refinement is performed for triangles which do not have exactly two zero neighbours because such places may represent crossings, bifurcations, turning points or other difficulties. The initial subdivision and the refinements are controlled by the option *points* which is initially set to 20: the initial grid is refined unconditionally until approximately *points* * *points* equally distributed points in the x, y plane have been generated.

The final mesh can be visualized in the picture by setting

```
on show_grid;
```

16.25.5 Mesh generation for surfaces

By default the functions are computed at predefined mesh points: the ranges are divided by the number associated with the option `points` in both directions.

For two dimensions the given mesh is adaptively smoothed when the curves are too coarse, especially if singularities are present. On the other hand refinement can be rather time consuming if used with complicated expressions. You can control it with the option `refine`. At singularities the graph is interrupted.

In three dimensions no refinement is possible as GNUPLOT supports surfaces only with a fixed regular grid. In the case of a singularity the near neighborhood is tested; if a point there allows a function evaluation, its clipped value is used instead, otherwise a zero is inserted.

When plotting surfaces in three dimensions you have the option of hidden line removal. Because of an error in Gnuplot 3.2 the axes cannot be labeled correctly when `hidden3d` is used ; therefore they aren't labelled at all. Hidden line removal is not available with point lists.

16.25.6 GNUPLOT operation

The command `PLOTRESET;` deletes the current GNUPLOT output window. The next call to `PLOT` will then open a new one.

If GNUPLOT is invoked directly by an output pipe (UNIX and Windows), an eventual error in the GNUPLOT data transmission might cause GNUPLOT to quit. As `REDUCE` is unable to detect the broken pipe, you have to reset the plot system by calling the command `PLOTRESET;` explicitly. Afterwards new graphic output can be produced.

Under Windows 3.1 and Windows NT, GNUPLOT has a text and a graph window. If you don't want to see the text window, iconify it and activate the option `update wgnuplot.ini` from the graph window system menu - then the present screen layout (including the graph window size) will be saved and the text windows will come up iconified in future. You can also select some more features there and so tailor the graphic output. Before you terminate `REDUCE` you should terminate the graphic window by calling `PLOTRESET;.` If you terminate `REDUCE` without deleting the GNUPLOT windows, use the command button from the GNUPLOT text window - it offers an exit function.

16.25.7 Saving GNUPLOT command sequence

If you want to use the internal GNUPLOT command sequence more than once (e.g. for producing a picture for a publication), you may set

```
ON TRPLOT,PLOTKEEP;
```

TRPLOT causes all GNUPLOT commands to be written additionally to the actual REDUCE output. Normally the data files are erased after calling GNUPLOT, however with PLOTKEEP on the files are not erased.

16.25.8 Direct Call of GNUPLOT

GNUPLOT has a lot of facilities which are not accessed by the operators and parameters described above. Therefore genuine GNUPLOT commands can be sent by REDUCE. Please consult the GNUPLOT manual for the available commands and parameters. The general syntax for a GNUPLOT call inside REDUCE is

```
gnuplot(< cmd >,< p1 >,< p2 > ...)
```

where *< cmd >* is a command name and *< p₁ >*, *< p₂ >* ... are the parameters, inside REDUCE separated by commas. The parameters are evaluated by REDUCE and then transmitted to GNUPLOT in GNUPLOT syntax. Usually a drawing is built by a sequence of commands which are buffered by REDUCE or the operating system. For terminating and activating them use the REDUCE command `plotshow`. Example:

```
gnuplot (set,polar);
gnuplot (set,noparametric);
gnuplot (plot,x*sin x);
plotshow;
```

In this example the function expression is transferred literally to GNUPLOT, while REDUCE is responsible for computing the function values when `PLOT` is called. Note that GNUPLOT restrictions with respect to variable and function names have to be taken into account when using this type of operation. **Important:** String quotes are not transferred to the GNUPLOT executable; if the GNUPLOT syntax needs string quotes, you must add doubled stringquotes **inside** the argument string, e.g.

```
gnuplot (plot, "" "mydata" "", "using 2:1");
```

16.25.9 Examples

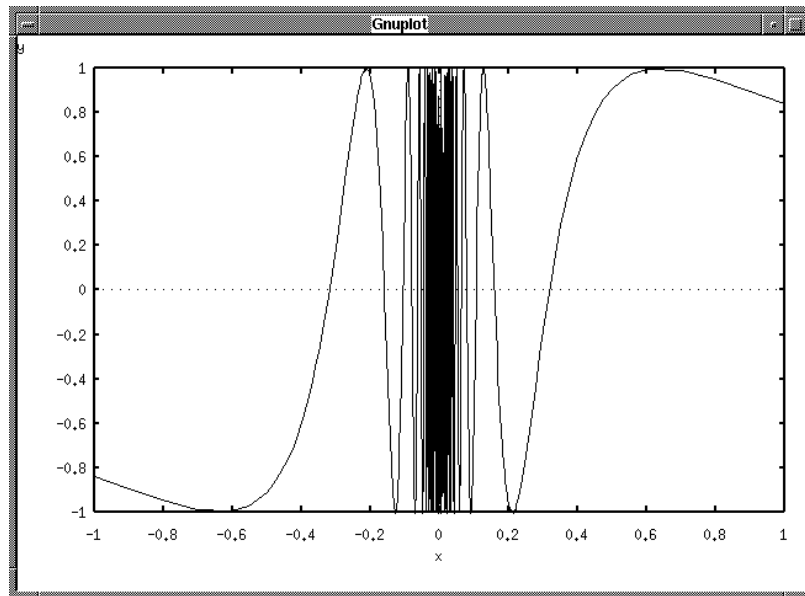
The following examples pictures are taken from a collection of sample plot (`gnuplot.tst`) and a set of tests for plotting special functions (will be published in the REDUCE electronic library).

The pictures are taken directly form the X screen by `xv` or produced by the *Gnuplot*

postscript driver directly.

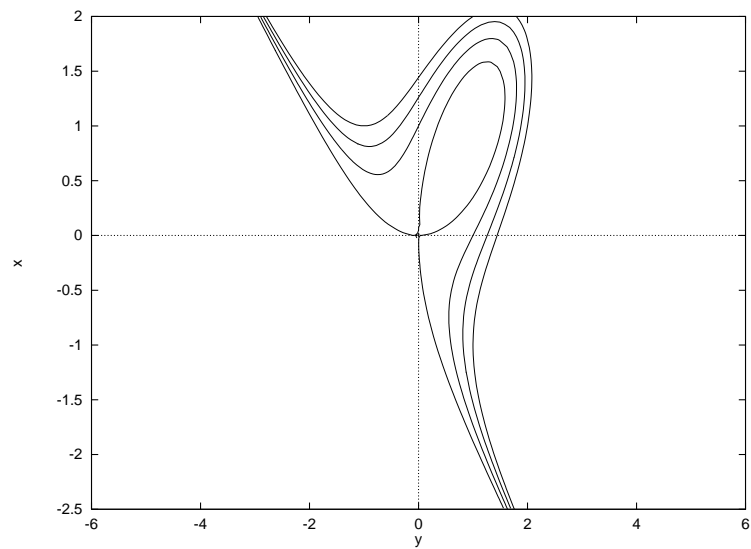
A simple plot for $\sin(1/x)$:

```
plot(sin(1/x),x=(-1 .. 1),y=(-3 .. 3));
```



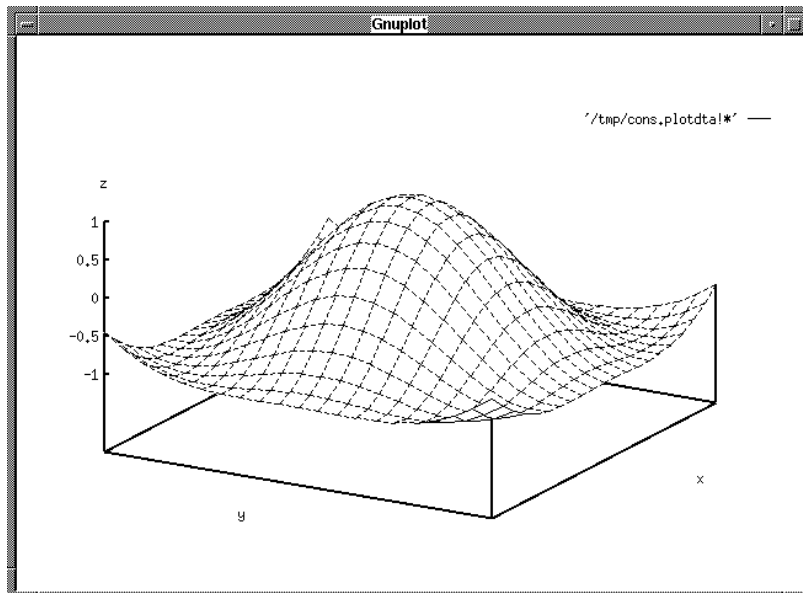
Some implicitly defined curves.

```
plot(x^3+y^3-3*x*y={0,1,2,3},x=(-2.5 .. 2),y=(-5 .. 5));
```



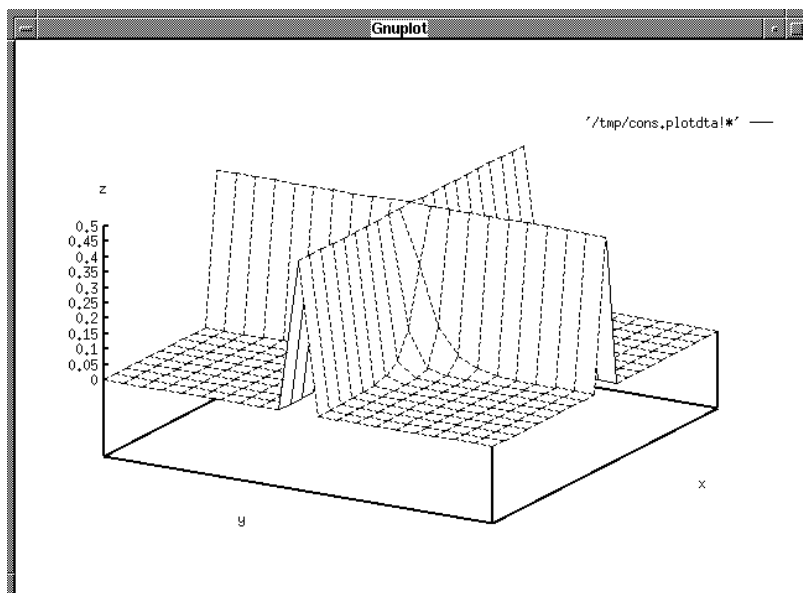
A test for hidden surfaces:

```
plot (cos sqrt(x**2 + y**2),x=(-3 .. 3),y=(-3 .. 3),hidden3d);
```



This may be slow on some machines because of a delicate evaluation context.

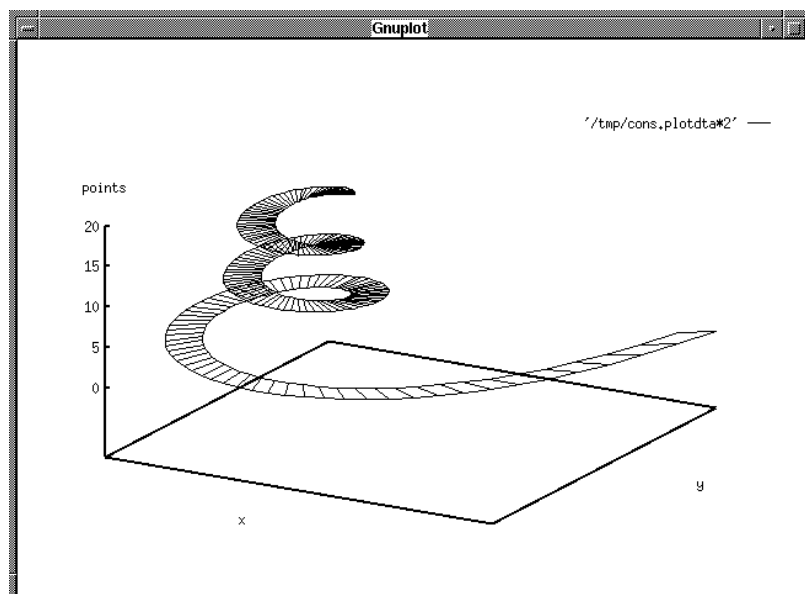
```
plot (sinh(x*y)/sinh(2*x*y),hidden3d);
```



```

on rounded;
w:= {for j:=1 step 0.1 until 20 collect {1/j*sin j,1/j*cos j,j},
      for j:=1 step 0.1 until 20 collect
      {(0.1+1/j)*sin j,(0.1+1/j)*cos j,j} }$
plot w;

```

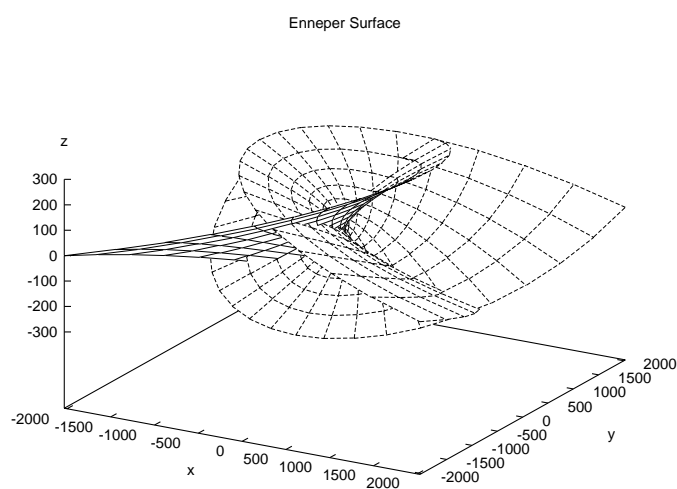


An example taken from: Cox, Little, O'Shea: Ideals, Varieties and Algorithms

```

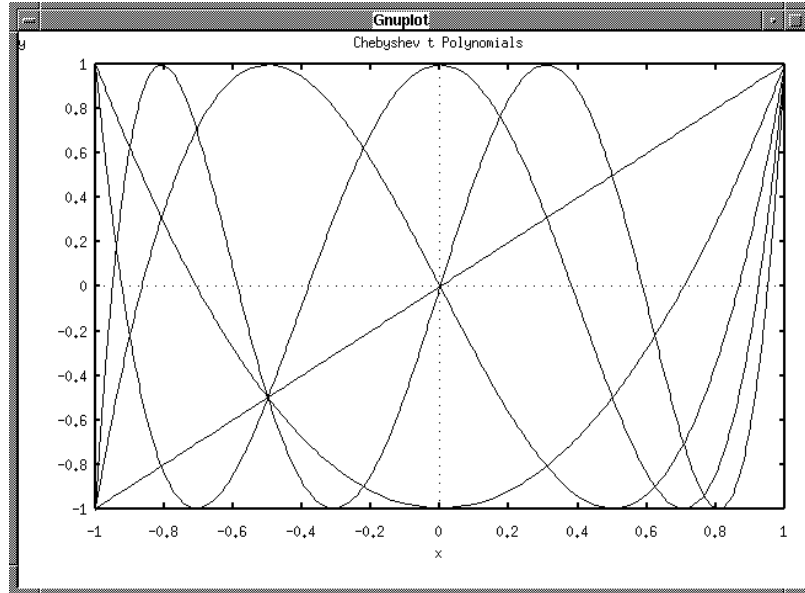
plot (point (3u+3u*v^2-u^3, 3v+3u^2*v-v^3, 3u^2-3v^2), hidden3d, title="Enneper Surface"

```

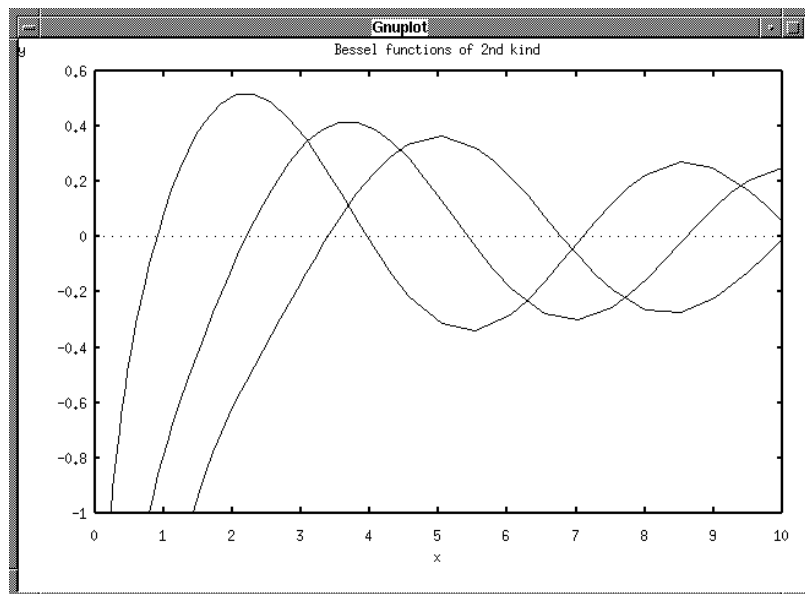


The following examples use the `specfn` package to draw a collection of Chebyshev's T polynomials and Bessel Y Functions. The special function package has to be loaded explicitly to make the operator `ChebyshevT` and `BesselY` available.

```
load_package specfn;
plot (chebyshevt (1,x),chebyshevt (2,x),chebyshevt (3,x),chebyshevt (4,x),
      chebyshevt (5,x),x=(-1 .. 1),title="Chebyshev t Polynomials");
```



```
plot (bessely (0,x),bessely (1,x),bessely (2,x),x=(0.1 .. 10)
      ,y=(-1 .. 1), title="Bessel functions of 2nd kind");
```



16.26 GROEBNER: A Gröbner basis package

GROEBNER is a package for the computation of Gröbner Bases using the Buchberger algorithm and related methods for polynomial ideals and modules. It can be used over a variety of different coefficient domains, and for different variable and term orderings.

Gröbner Bases can be used for various purposes in commutative algebra, e.g. for elimination of variables, converting surd expressions to implicit polynomial form, computation of dimensions, solution of polynomial equation systems etc. The package is also used internally by the SOLVE operator.

Authors: Herbert Melenk, H.M. Möller and Winfried Neun.

16.27 GUARDIAN: Guarded Expressions in Practice

Computer algebra systems typically drop some degenerate cases when evaluating expressions, e.g., x/x becomes 1 dropping the case $x = 0$. We claim that it is feasible in practice to compute also the degenerate cases yielding *guarded expressions*. We work over real closed fields but our ideas about handling guarded expression can be easily transferred to other situations. Using formulas as guards provides a powerful tool for heuristically reducing the combinatorial explosion of cases: equivalent, redundant, tautological, and contradictory cases can be detected by simplification and quantifier elimination. Our approach allows to simplify the expressions on the basis of simplification knowledge on the logical side. The method described in this paper is implemented in the REDUCE package GUARDIAN.

Authors: Andreas Dolzmann and Thomas Sturm.

Guarded Expressions in Practice²⁴

Computer algebra systems typically drop some degenerate cases when evaluating expressions, e.g., x/x becomes 1 dropping the case $x = 0$. We claim that it is feasible in practice to compute also the degenerate cases yielding *guarded expressions*. We work over real closed fields but our ideas about handling guarded expression can be easily transferred to other situations. Using formulas as guards provides a powerful tool for heuristically reducing the combinatorial explosion of cases: equivalent, redundant, tautological, and contradictory cases can be detected by simplification and quantifier elimination. Our approach allows to simplify the expressions on the basis of simplification knowledge on the logical side. The method described in this paper is implemented in the REDUCE package GUARDIAN.

16.27.1 Introduction

It is meanwhile a well-known fact that evaluations obtained with the interactive use of computer algebra systems (CAS) are not entirely correct in general. Typically, some degenerate cases are dropped. Consider for instance the evaluation

$$\frac{x^2}{x} = x,$$

which is correct only if $x \neq 0$. The problem here is that CAS consider variables to be transcendental elements. The user, in contrast, has in mind variables in the sense of logic. In other words: The user does not think of rational functions but of terms.

Next consider the valid expression

$$\frac{\sqrt{x} + \sqrt{-x}}{x}.$$

It is meaningless over the reals. CAS often offer no choice than to interpret surds over the complex numbers even if they distinguish between a *real* and a *complex* mode.

Corless and Jeffrey [4] have examined the behavior of a number of CAS with such input data. They come to the conclusion that simultaneous computation of all cases is exemplary but not feasible due to the combinatorial explosion of cases to be considered. Therefore, they suggest to ignore the degenerate cases but to provide the assumptions to the user on request. We claim, in contrast, that it is in fact feasible to compute all possible cases.

Our setting is as follows: Expressions are evaluated to *guarded expressions* consisting of possibly several conventional expressions guarded by quantifier-free formulas. For the above examples, we would obtain

$$\left[x \neq 0 \mid x \right], \quad \left[\mathbf{F} \mid \frac{\sqrt{x} + \sqrt{-x}}{x} \right].$$

As the second example illustrates, we are working in ordered fields, more precisely in real closed fields. The handling of guarded expressions as described in this paper can, however, be easily transferred to other situations.

Our approach can also deal with redundant guarded expressions, such as

$$\left[\begin{array}{c|c} \mathbf{T} & |x| - x \\ x \geq 0 & 0 \\ x < 0 & -2x \end{array} \right]$$

which leads to algebraic simplification techniques based on logical simplification as proposed by Davenport and Faure [5].

We use *formulas* over the language of ordered rings as guards. This provides powerful tools for heuristically reducing the combinatorial explosion of cases: equivalent, redundant, tautological, and contradictory cases can be detected by *simplification* [6] and *quantifier elimination* [17, 3, 18, 15, 21, 20]. In certain situations, we will allow the formulas also to contain extra functions such as $\sqrt{\cdot}$ or $|\cdot|$. Then we take care that there is no quantifier elimination applied.

Simultaneous computation of several cases concerning certain expressions being zero or not has been extensively investigated as *dynamic evaluation* [12, 10, 11, 2]. It has also been extended to real closed fields [9]. The idea behind the development of these methods is of a more theoretical nature than to overcome the problems with the interactive usage of CAS sketched above: one wishes to compute in algebraic (or real) extension fields of the rationals. Guarded expressions occur naturally when solving problems parametrically. Consider, e.g., the *Gröbner systems* used during the computation of *comprehensive Gröbner bases* [19].

The algorithms described in this paper are implemented in the REDUCE package GUARDIAN. It is based on the REDUCE [13, 16] package REDLOG [7, 8] implementing a formula data type with corresponding algorithms, in particular including simplification and quantifier elimination.

16.27.2 An outline of our method

Guarded expressions

A *guarded expression* is a scheme

$$\left[\begin{array}{c|c} \gamma_0 & t_0 \\ \gamma_1 & t_1 \\ \vdots & \vdots \\ \gamma_n & t_n \end{array} \right]$$

where each γ_i is a quantifier-free formula, the *guard*, and each t_i is an associated *conventional expression*. The idea is that some t_i is a valid interpretation iff γ_i holds. Each pair (γ_i, t_i) is called a *case*.

The first case (γ_0, t_0) is the *generic case*: t_0 is the expression the system would compute without our package, and γ_0 is the corresponding guard.

The guards γ_i need neither exclude one another, nor do we require that they form a complete case distinction. We shall, however, assume that all cases covered by a guarded expression are already covered by the generic case; in other words:

$$\bigwedge_{i=1}^n (\gamma_i \longrightarrow \gamma_0). \quad (16.48)$$

Consider the following evaluation of $|x|$ to a guarded expression:

$$\left[\begin{array}{c|c} \mathbf{T} & |x| \\ x \geq 0 & x \\ x < 0 & -x \end{array} \right].$$

Here the non-generic cases already cover the whole domain. The generic case is in some way *redundant*. It is just present for keeping track of the system's default behavior. Formally we have

$$\left(\bigvee_{i=1}^n \gamma_i \right) \longleftrightarrow \gamma_0. \quad (16.49)$$

As an example for a non-redundant, i.e., *necessary* generic case we have the evaluation of the reciprocal $\frac{1}{x}$:

$$\left[x \neq 0 \mid \frac{1}{x} \right].$$

In every guarded expression, the generic case is explicitly marked as either necessary or redundant. The corresponding tag is inherited during the evaluation process. Unfortunately it can happen that guarded expressions satisfy (16.49) without being tagged redundant, e.g., specialization of

$$\left[\begin{array}{c|c} \mathbf{T} & \sin x \\ x = 0 & 0 \end{array} \right]$$

to $x = 0$ if the system cannot evaluate $\sin(0)$. This does not happen if one claims for necessary generic cases to have, as the reciprocal above, no alternative cases at all. Else, in the sequel “redundant generic case” has to be read as “tagged redundant.”

With guarded expressions, the evaluation splits into two independent parts: *Algebraic evaluation* and a subsequent *simplification* of the guarded expression obtained.

Guarding schemes

In the introduction we have seen that certain operators introduce case distinctions. For this, with each operator f there is a *guarding scheme* associated providing information on how to map $f(t_1, \dots, t_m)$ to a guarded expression provided that one does not have to care for the argument expressions t_1, \dots, t_m . In the easiest case, this is a rewrite rule

$$f(a_1, \dots, a_m) \rightarrow G(a_1, \dots, a_m).$$

The actual terms t_1, \dots, t_m are simply substituted for the formal symbols a_1, \dots, a_m into the generic guarded expression $G(a_1, \dots, a_m)$. We give some examples:

$$\begin{aligned}
 \frac{a_1}{a_2} &\rightarrow \left[\mathbf{a_2 \neq 0} \mid \frac{a_1}{a_2} \right] \\
 \sqrt{a_1} &\rightarrow \left[\mathbf{a_1 \geq 0} \mid \sqrt{a_1} \right] \\
 \text{sign}(a_1) &\rightarrow \left[\begin{array}{c|c} \mathbf{T} & \mathbf{sign(a_1)} \\ a_1 > 0 & 1 \\ a_1 = 0 & 0 \\ a_1 < 0 & -1 \end{array} \right] \\
 |a_1| &\rightarrow \left[\begin{array}{c|c} \mathbf{T} & |a_1| \\ a_1 \geq 0 & a_1 \\ a_1 < 0 & -a_1 \end{array} \right]
 \end{aligned} \tag{16.50}$$

For functions of arbitrary arity, e.g., min or max, we formally assume infinitely many operators of the same name. Technically, we associate a procedure parameterized with the number of arguments m that generates the corresponding rewrite rule. As `min_scheme(2)` we obtain, e.g.,

$$\min(a_1, a_2) \rightarrow \left[\begin{array}{c|c} \mathbf{T} & \mathbf{min(a_1, a_2)} \\ a_1 \leq a_2 & a_1 \\ a_2 \leq a_1 & a_2 \end{array} \right], \tag{16.51}$$

while for higher arities there are more case distinctions necessary.

For later complexity analysis, we state the concept of a guarding scheme formally: a guarding scheme for an m -ary operator f is a map

$$\text{gscheme}_f : E^m \rightarrow \text{GE}$$

where E is the set of expressions, and GE is the set of guarded expressions. This allows to split $f(t_1, \dots, t_m)$ in dependence on the form of the parameter expressions t_1, \dots, t_m .

Algebraic evaluation

Evaluating conventional expressions The evaluation of conventional expressions into guarded expressions is performed recursively: Constants c evaluate to

$$\left[\mathbf{T} \mid c \right].$$

For the evaluation of $f(e_1, \dots, e_m)$ the argument expressions e_1, \dots, e_m are recursively evaluated to guarded expressions

$$e'_i = \left[\begin{array}{c|c} \gamma_{i0} & t_{i0} \\ \gamma_{i1} & t_{i1} \\ \vdots & \vdots \\ \gamma_{in_i} & t_{in_i} \end{array} \right] \quad \text{for } 1 \leq i \leq m. \quad (16.52)$$

Then the operator f is “moved inside” the e'_i by combining all cases, technically a simultaneous Cartesian product computation of both the sets of guards and the sets of terms:

$$\Gamma = \prod_{i=1}^m \{\gamma_{i0}, \dots, \gamma_{in_i}\}, \quad T = \prod_{i=1}^m \{t_{i0}, \dots, t_{in_i}\}. \quad (16.53)$$

This leads to the intermediate result

$$\left[\begin{array}{c|c} \gamma_{10} \wedge \dots \wedge \gamma_{m0} & f(t_{10}, \dots, t_{m0}) \\ \vdots & \vdots \\ \gamma_{1n_1} \wedge \dots \wedge \gamma_{mn_1} & f(t_{1n_1}, \dots, t_{mn_1}) \\ \vdots & \vdots \\ \gamma_{1n_m} \wedge \dots \wedge \gamma_{mn_m} & f(t_{1n_m}, \dots, t_{mn_m}) \end{array} \right]. \quad (16.54)$$

The new generic case is exactly the combination of the generic cases of the e'_i . It is redundant if at least one of these combined cases is redundant.

Next, all non-generic cases containing at least one *redundant* generic constituent γ_{i0} in their guard are deleted. The reason for this is that generic cases are only used to keep track of the system default behavior. All other cases get the status of a non-generic case even if they contain necessary generic constituents in their guard.

At this point, we apply the guarding scheme of f to all remaining expressions $f(t_{1i_1}, \dots, t_{mi_m})$ in the form (16.54) yielding a nested guarded expression

$$\left[\begin{array}{c|c} \Gamma_0 & \left[\begin{array}{c|c} \delta_{00} & u_{00} \\ \vdots & \vdots \\ \delta_{0k_0} & u_{0k_0} \end{array} \right] \\ \vdots & \vdots \\ \Gamma_N & \left[\begin{array}{c|c} \delta_{N0} & u_{N0} \\ \vdots & \vdots \\ \delta_{Nk_N} & u_{Nk_N} \end{array} \right] \end{array} \right], \quad (16.55)$$

which can be straightforwardly resolved to a guarded expression

$$\left[\begin{array}{c|c} \Gamma_0 \wedge \delta_{00} & u_{00} \\ \vdots & \vdots \\ \Gamma_0 \wedge \delta_{0k_0} & u_{0k_0} \\ \vdots & \vdots \\ \Gamma_N \wedge \delta_{N0} & u_{N0} \\ \vdots & \vdots \\ \Gamma_N \wedge \delta_{Nk_N} & u_{Nk_N} \end{array} \right].$$

This form is treated analogously to the form (16.54): The new generic case $(\Gamma_0 \wedge \delta_{00}, u_{00})$ is redundant if at least one of $(\Gamma_0, f(t_{10}, \dots, t_{m0}))$ and (δ_{00}, u_{00}) is redundant. Among the non-generic cases all those containing redundant generic constituents in their guard are deleted, and all those containing necessary generic constituents in their guard get the status of an ordinary non-generic case.

Finally the standard evaluator of the system—`reval` in the case of REDUCE—is applied to all contained expressions, which completes the algebraic part of the evaluation.

Evaluating guarded expressions The previous section was concerned with the evaluation of pure conventional expressions into guarded expressions. Our system currently combines both conventional and guarded expressions. We are thus faced with the problem of treating guarded subexpressions during evaluation.

When there is a *guarded* subexpression e_i detected during evaluation, all contained expressions are recursively evaluated to guarded expressions yielding a nested guarded expression of the form (16.55). This is resolved as described above yielding the evaluation subresult e'_i .

As a special case, this explains how guarded expressions are (re)evaluated to guarded expressions.

Example

We describe the evaluation of the expression $\min(x, |x|)$. The first argument $e_1 = x$ evaluates recursively to

$$e'_1 = [\mathbf{T} \mid x] \quad (16.56)$$

with a necessary generic case. The nested x inside $e_2 = |x|$ evaluates to the same form (16.56). For obtaining e'_2 , we apply the guarding scheme (16.50) of the absolute value to the only term of (16.56) yielding

$$\left[\mathbf{T} \mid \left[\begin{array}{c|c} \mathbf{T} & |x| \\ x \geq 0 & x \\ x < 0 & -x \end{array} \right] \right],$$

where the inner generic case is redundant. This form is resolved to

$$e'_2 = \left[\begin{array}{c|c} \mathbf{T} \wedge \mathbf{T} & |x| \\ \hline \mathbf{T} \wedge x \geq 0 & x \\ \mathbf{T} \wedge x < 0 & -x \end{array} \right]$$

with a redundant generic case. The next step is the combination of cases by Cartesian product computation. We obtain

$$\left[\begin{array}{c|c} \mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T}) & \mathbf{min}(x, |x|) \\ \hline \mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0) & \mathbf{min}(x, x) \\ \mathbf{T} \wedge (\mathbf{T} \wedge x < 0) & \mathbf{min}(x, -x) \end{array} \right],$$

which corresponds to (16.54) above. For the outer min, we apply the guarding scheme (16.51) to all terms yielding the nested guarded expression

$$\left[\begin{array}{c|c} \mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T}) & \left[\begin{array}{c|c} \mathbf{T} & \mathbf{min}(x, |x|) \\ \hline x \leq |x| & x \\ |x| \leq x & |x| \end{array} \right] \\ \hline \mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0) & \left[\begin{array}{c|c} \mathbf{T} & \mathbf{min}(x, x) \\ \hline x \leq x & x \\ x \leq x & x \end{array} \right] \\ \hline \mathbf{T} \wedge (\mathbf{T} \wedge x < 0) & \left[\begin{array}{c|c} \mathbf{T} & \mathbf{min}(x, -x) \\ \hline x \leq -x & x \\ -x \leq x & -x \end{array} \right] \end{array} \right],$$

which is in turn resolved to

$$\left[\begin{array}{c|c} (\mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T})) \wedge \mathbf{T} & \mathbf{min}(x, |x|) \\ \hline (\mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T})) \wedge x \leq |x| & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T})) \wedge |x| \leq x & |x| \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0)) \wedge \mathbf{T} & \mathbf{min}(x, x) \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x < 0)) \wedge \mathbf{T} & \mathbf{min}(x, -x) \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x < 0)) \wedge x \leq -x & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x < 0)) \wedge -x \leq x & -x \end{array} \right].$$

From this, we delete the two non-generic cases obtained by combination with the redundant generic case of the min. The final result of the algebraic evaluation step is the following:

$$\left[\begin{array}{c|c} (\mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T})) \wedge \mathbf{T} & \mathbf{min}(x, |x|) \\ \hline (\mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T})) \wedge x \leq |x| & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge \mathbf{T})) \wedge |x| \leq x & |x| \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x \geq 0)) \wedge x \leq x & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x < 0)) \wedge x \leq -x & x \\ (\mathbf{T} \wedge (\mathbf{T} \wedge x < 0)) \wedge -x \leq x & -x \end{array} \right]. \quad (16.57)$$

Worst-case complexity

Our measure of complexity $|G|$ for guarded expressions G is the number of contained cases:

$$\left| \left[\begin{array}{c|c} \gamma_0 & t_0 \\ \gamma_1 & t_1 \\ \vdots & \vdots \\ \gamma_n & t_n \end{array} \right] \right| = n + 1.$$

As in Section 16.27.2, consider an m -ary operator f , guarded expression arguments e'_1, \dots, e'_m as in equation (16.52), and the Cartesian product T as in equation (16.53). Then

$$\begin{aligned} |f(e'_1, \dots, e'_m)| &\leq \sum_{(t_1, \dots, t_m) \in T} |\text{gscheme}_f(t_1, \dots, t_m)| \\ &\leq \max_{(t_1, \dots, t_m) \in T} |\text{gscheme}_f(t_1, \dots, t_m)| \cdot \#T \\ &= \max_{(t_1, \dots, t_m) \in T} |\text{gscheme}_f(t_1, \dots, t_m)| \cdot \prod_{j=1}^m |e'_j| \\ &\leq \max_{(t_1, \dots, t_m) \in T} |\text{gscheme}_f(t_1, \dots, t_m)| \cdot \left(\max_{1 \leq j \leq m} |e'_j| \right)^m. \end{aligned}$$

In the important special case that the guarding scheme of f is a rewrite rule $f(a_1, \dots, a_m) \rightarrow G$, the above complexity estimation simplifies to

$$|f(e'_1, \dots, e'_m)| \leq |G| \cdot \prod_{j=1}^m |e'_j| \leq |G| \cdot \left(\max_{1 \leq j \leq m} |e'_j| \right)^m.$$

In other words: $|G|$ plays the role of a factor, which, however, depends on f , and $|f(e'_1, \dots, e'_m)|$ is polynomial in the size of the e_i but exponential in the arity of f .

Simplification

In view of the increasing size of the guarded expressions coming into existence with subsequent computations, it is indispensable to apply simplification strategies. There are two different algorithms involved in the simplification of guarded expressions:

1. A *formula simplifier* mapping quantifier-free formulas to equivalent simpler ones.
2. Effective *quantifier elimination* for real closed fields over the language of ordered rings.

It is not relevant, which simplifier and which quantifier elimination procedure is actually used. We use the formula simplifier described in [6]. Our quantifier elimination uses test point methods developed by Weispfenning [18, 15, 21]. It is restricted to formulas obeying certain degree restrictions wrt. the quantified variables. As an alternative, REDLOG provides an interface to Hong’s QEPCAD quantifier elimination package [14]. Compared to the simplification, the quantifier elimination is more time consuming. It can be turned off by a *switch*.

The following simplification steps are applied in the given order:

Contraction of cases This is restricted to the non-generic cases of the considered guarded expression. We contract different cases containing the same terms:

$$\left[\begin{array}{c|c} \gamma_0 & t_0 \\ \vdots & \vdots \\ \gamma_i & t_i \\ \vdots & \vdots \\ \gamma_j & t_i \\ \vdots & \vdots \end{array} \right] \quad \text{becomes} \quad \left[\begin{array}{c|c} \gamma_0 & t_0 \\ \vdots & \vdots \\ \gamma_i \vee \gamma_j & t_i \\ \vdots & \vdots \end{array} \right].$$

Simplification of the guards The simplifier is applied to all guards replacing them by simplified equivalents. Since our simplifier maps $\gamma \vee \gamma$ to γ , this together with the contraction of cases takes care for the deletion of duplicate cases.

Keep one tautological case If the guard of some non-generic case becomes “T,” we delete all other non-generic cases. Else, if quantifier elimination is turned on, we try to detect a tautology by eliminating the universal closures $\forall \gamma$ of the guards γ . This quantifier elimination is also applied to the guards of generic cases. These are, in case of success, simply replaced by “T” without deleting the case.

Remove contradictory cases A non-generic case is deleted if its guard has become “F.” If quantifier elimination is turned on, we try to detect further contradictory cases by eliminating the existential closure $\exists \gamma$ for each guard γ . This quantifier elimination is also applied to generic cases. In case of success they are not deleted but their guards are replaced by “F.” Our assumption (16.48) allows then to delete all non-generic cases.

Example revisited

We turn back to the form (16.57) of our example $\min(x, |x|)$. Contraction of cases with subsequent simplification automatically yields

$$\left[\begin{array}{c|c} \mathbf{T} & \mathbf{min}(x, |x|) \\ \mathbf{T} & x \\ |x| - x \leq 0 & |x| \\ \mathbf{F} & -x \end{array} \right],$$

of which only the tautological non-generic case survives:

$$\left[\begin{array}{c|c} \mathbf{T} & \mathbf{min}(x, |x|) \\ \mathbf{T} & x \end{array} \right]. \quad (16.58)$$

Output modes

An *output mode* determines which part of the information contained in the guarded expressions is provided to the user. GUARDIAN knows the following output modes:

Matrix Output matrices in the style used throughout this paper. We have already seen that these can become very large in general.

Generic case Output only the generic case.

Generic term Output only the generic term. Thus the output is exactly the same as without the guardian package. If the condition of the generic case becomes “F,” a *warning* “contradictive situation” is given. The computation can, however, be continued.

Note that output modes are restrictions concerning only the output; internally the system still computes with the complete guarded expressions.

A smart mode

Consider the evaluation result (16.58) of $\min(x, |x|)$. The *generic term* output mode would output $\min(x, |x|)$, although more precise information could be given, namely x . The problem is caused by the fact that generic cases are used to keep track of the system’s default behavior. In this section we will describe an optional *smart mode* with a different notion of *generic case*. To begin with, we show why the problem can not be overcome by a “smart output mode.”

Assume that there is an output mode which outputs x for (16.58). As the next computation involving (16.58) consider division by y . This would result in

$$\left[\begin{array}{l|l} y \neq 0 & \frac{\min(x, |x|)}{y} \\ y \neq 0 & \frac{x}{y} \end{array} \right].$$

Again, there are identic conditions for the generic case and some non-generic case, and, again, the term belonging to the latter is simpler. Our mode would output $\frac{x}{y}$. Next, we apply the absolute value once more yielding

$$\left[\begin{array}{l|l} y \neq 0 & \frac{|\min(x, |x|)|}{|y|} \\ xy \geq 0 \wedge y \neq 0 & \frac{x}{y} \\ xy < 0 \wedge y \neq 0 & \frac{-x}{y} \end{array} \right].$$

Here, the condition of the generic case differs from all other conditions. We thus have to output the generic term. For the user, the evaluation of $|\frac{x}{y}|$ results in $\frac{|\min(x, |x|)|}{|y|}$.

The smart mode can turn a non-generic case into a necessary generic one dropping the original generic case and all other non-generic cases. Consider, e.g., (16.58), where the conditions are equal, and the non-generic term is “simpler.”

In fact, the relevant relationship between the conditions is that the generic condition *implies* the non-generic one. In other words: Some non-generic condition is not more restrictive than the generic condition, and thus covers the whole domain of the guarded expression. Note that from the implication and (16.48) we may conclude that the cases are even equivalent.

Implication is heuristically checked by simplification. If this fails, quantifier elimination provides a decision procedure. Note that our test point methods are incomplete in this regard due to the degree restrictions. Also it cannot be applied straightforwardly to guards containing operators that do not belong to the language of ordered rings.

Whenever we happen to detect a relevant implication, we actually turn the corresponding non-generic case into the generic one. From our motivation of non-generic cases, we may expect that non-generic expressions are generally more convenient than generic ones.

16.27.3 Examples

We give the results for the following computations as they are printed in the output mode *matrix* providing the full information on the computation result. The reader can derive himself what the output in the mode *generic case* or *generic term* would be.

- Smart mode or not:

$$\frac{1}{x^2 + 2x + 1} = \left[x + 1 \neq 0 \mid \frac{1}{x^2 + 2x + 1} \right].$$

The simplifier recognizes that the denominator is a square.

- Smart mode or not:

$$\frac{1}{x^2 + 2x + 2} = \left[\mathbf{T} \mid \frac{1}{x^2 + 2x + 2} \right].$$

Quantifier elimination recognizes the positive definiteness of the denominator.

- Smart mode:

$$|x| - \sqrt{x} = \left[x \geq 0 \mid -\sqrt{x} + x \right].$$

The square root allows to forget about the negative branch of the absolute value.

- Smart mode:

$$|x^2 + 2x + 1| = \left[\mathbf{T} \mid x^2 + 2x + 1 \right].$$

The simplifier recognizes the positive semidefiniteness of the argument. REDUCE itself recognizes squares within absolute values only in very special cases such as $|x^2|$.

- Smart mode:

$$\min(x, \max(x, y)) = \left[\mathbf{T} \mid x \right].$$

Note that REDUCE does not know any rules about nested minima and maxima.

- Smart mode:

$$\min(\text{sign}(x), -1) = \left[\mathbf{T} \mid -1 \right].$$

- Smart mode or not:

$$|x| - x = \left[\begin{array}{c|c} \mathbf{T} & |x| - x \\ x \geq 0 & 0 \\ x < 0 & -2x \end{array} \right].$$

This example is taken from [5].

- Smart mode or not:

$$\sqrt{1 + x^2 y^2 (x^2 + y^2 - 3)} = \left[\mathbf{T} \mid \sqrt{x^4 y^2 + x^2 y^4 - 3x^2 y^2 + 1} \right]$$

The *Motzkin polynomial* is recognized to be positive semidefinite by quantifier elimination.

The evaluation time for the last example is 119 ms on a SUN SPARC-4. This illustrates that efficiency is no problem with such interactive examples.

16.27.4 Outlook

This section describes possible extensions of the GUARDIAN. The extensions proposed in Section 16.27.4 on simplification of terms and Section 16.27.4 on a background theory are clear from a theoretical point of view but not yet implemented. Section 16.27.4 collects some ideas on the application of our ideas to the REDUCE integrator. In this field, there is some more theoretical work necessary.

Simplification of terms

Consider the expression $\text{sign}(x)x - |x|$. It evaluates to the following guarded expression:

$$\left[\begin{array}{l|l} \mathbf{T} & -|x| + \text{sign}(x)x \\ x \neq 0 & 0 \\ x = 0 & -x \end{array} \right].$$

This suggests to substitute $-x$ by 0 in the third case, which would in turn allow to contract the two non-generic cases yielding

$$\left[\begin{array}{l|l} \mathbf{T} & -|x| + \text{sign}(x)x \\ \mathbf{T} & 0 \end{array} \right].$$

In smart mode second case would then become the only generic case.

Generally, one would proceed as follows: If the guard is a conjunction containing as toplevel equations

$$t_1 = 0, \quad \dots, \quad t_k = 0,$$

reduce the corresponding expression modulo the set of univariate linear polynomials among t_1, \dots, t_k .

A more general approach would reduce the expression modulo a Gröbner basis of all the t_1, \dots, t_k . This leads, however, to larger expressions in general.

One can also imagine to make use of non-conjunctive guards in the following way:

1. Compute a DNF of the guard.
2. Split the case into several cases corresponding to the conjunctions in the DNF.
3. Simplify the terms.
4. Apply the standard simplification procedure to the resulting guarded expression. Note that it includes *contraction of cases*.

According to experiences with similar ideas in the “Gröbner simplifier” described in [6], this should work well.

Background theory

In practice one often computes with quantities guaranteed to lie in a certain range. For instance, when computing an electrical resistance, one knows in advance that it will not be negative. For such cases one would like to have some facility to provide external information to the system. This can then be used to reduce the complexity of the guarded expressions.

One would provide a function `assert(φ)`, which asserts the formula φ to hold. Successive applications of `assert` establish a *background theory*, which is a set of formulas considered conjunctively. The information contained in the background theory can be used with the guarded expression computation. The user must, however, not rely on all the background information to be actually used.

Technically, denote by Φ the (conjunctive) background theory. For the *simplification of the guards*, we can make use of the fact that our simplifier is designed to simplify wrt. a theory, cf. [6]. For proving that some guard γ is *tautological*, we try to prove

$$\forall(\Phi \longrightarrow \gamma)$$

instead of $\forall\gamma$. Similarly, for proving that γ is *contradictive*, we try to disprove

$$\exists(\Phi \wedge \gamma).$$

Instead of proving $\forall(\gamma_1 \longrightarrow \gamma_2)$ in smart mode, we try to prove

$$\forall((\Phi \wedge \gamma_1) \longrightarrow \gamma_2).$$

Independently, one can imagine to use a background theory for reducing the *output* with the *matrix* output mode. For this, one simplifies each guard wrt. the theory at the output stage treating contradictions and tautologies appropriately. Using the theory for replacing all cases by one at output stage in a smart mode manner leads once more to the problem of expressions or even guarded expressions “mysteriously” getting more complicated. Applying the theory only at the output stage makes it possible to implement a procedure `unassert(φ)` in a reasonable way.

Integration

CAS integrators make “mistakes” similar to those we have examined. Consider, e.g., the typical result

$$\int x^a dx = \frac{1}{a+1} x^{a+1}.$$

It does not cover the case $a = -1$, for which one wishes to obtain

$$\int x^{-1} dx = \ln x.$$

This problem can also be solved by using guarded expressions for integration results.

Within the framework of this paper, we would have to associate a guarding scheme to the integrator `int`. It is not hard to see that this cannot be done in a reasonable way without putting as much knowledge into the scheme as into the integrator itself. Thus for treating integration, one has to modify the integrator to provide guarded expressions.

Next, we have to clarify what the guarded expression for the above integral would look like. Since we know that the integral is defined for all interpretations of the variables, our assumption (16.48) implies that the generic condition be “T.” We obtain the guarded expression

$$\left[\begin{array}{c|c} \mathbf{T} & \int x^a dx \\ a \neq -1 & \frac{1}{a+1} x^{a+1} \\ a = -1 & \ln x \end{array} \right].$$

Note that the redundant generic case does not model the system’s current behavior.

Combining algebra with logic

Our method, in the described form, uses an already implemented algebraic evaluator. In the previous section, we have seen that this point of view is not sufficient for treating integration appropriately.

Also our approach runs into trouble with built-in knowledge such as

$$\sqrt{x^2} = |x|, \quad (16.59)$$

$$\text{sign}(|x|) = 1. \quad (16.60)$$

Equation (16.59) introduces an absolute value operator within a non-generic term without making a case distinction. Equation (16.60) is wrong when not considering x transcendental. In contrast to the situation with reciprocals, our technique cannot be used to avoid this “mistake.” We obtain

$$\text{sign}(|x|) = \left[\begin{array}{c|c} \mathbf{T} & \mathbf{1} \\ x \neq 0 & 1 \\ x = 0 & 0 \end{array} \right]$$

yielding two different answers for $x = 0$.

We have already seen in the example Section 16.27.3 that the implementation of knowledge such as (16.59) and (16.60) is usually quite *ad hoc*, and can be mostly covered by using guarded expressions. This observation gives rise to the following question: When designing a new CAS based on guarded expressions, how should the knowledge be distributed between the algebraic side and the logic side?

16.27.5 Conclusions

Guarded expressions can be used to overcome well-known problems with interpreting expressions as terms. We have explained in detail how to compute with guarded expressions including several simplification techniques. Moreover we gain algebraic simplification power from the logical simplifications. Numerous examples illustrate the power of our simplification methods. The largest part of our ideas is efficiently implemented, and the software is published. The outlook on background theories and on the treatment of integration by guarded expressions points on interesting future extensions.

Bibliography

- [1] Bradford, R. Algebraic simplification of multiple valued functions. In *Design and Implementation of Symbolic Computation Systems* (1992), J. Fitch, Ed., vol. 721 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 13–21. Proceedings of the DISCO 92.
- [2] Broadberry, P., Gómez-Díaz, T., and Watt, S. On the implementation of dynamic evaluation. In *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation (ISSAC 95)* (New York, N.Y., 1995), A. Lev-elt, Ed., ACM Press, pp. 77–89.
- [3] Collins, G. E. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Automata Theory and Formal Languages. 2nd GI Conference* (Berlin, Heidelberg, New York, May 1975), H. Brakhage, Ed., vol. 33 of *Lecture Notes in Computer Science*, Gesellschaft für Informatik, Springer-Verlag, pp. 134–183.
- [4] Corless, R. M., and Jeffrey, D. J. Well ...it isn't quite that simple. *ACM SIGSAM Bulletin* 26, 3 (Aug. 1992), 2–6. Feature.
- [5] Davenport, J. H., and Faure, C. The “unknown” in computer algebra. *Programirovanie* 1, 1 (1994).
- [6] Dolzmann, A., and Sturm, T. Simplification of quantifier-free formulas over ordered fields. Technical Report MIP-9517, FMI, Universität Passau, D-94030 Passau, Germany, Oct. 1995. To appear in the *Journal of Symbolic Computation*.
- [7] Dolzmann, A., and Sturm, T. Redlog—computer algebra meets computer logic. Technical Report MIP-9603, FMI, Universität Passau, D-94030 Passau, Germany, Feb. 1996.

- [8] Dolzmann, A., and Sturm, T. Redlog user manual. Technical Report MIP-9616, FMI, Universität Passau, D-94030 Passau, Germany, Oct. 1996. Edition 1.0 for Version 1.0.
- [9] Duval, D., and Gonzáles-Vega, L. Dynamic evaluation and real closure. In *Proceedings of the IMACS Symposium on Symbolic Computation* (1993).
- [10] Duval, D., and Reynaud, J.-C. Sketches and computation I: Basic definitions and static evaluation. *Mathematical Structures in Computer Science* 4, 2 (1994), 185–238.
- [11] Duval, D., and Reynaud, J.-C. Sketches and computation II: Dynamic evaluation and applications. *Mathematical Structures in Computer Science* 4, 2 (1994), 239–271.
- [12] Gómez-Díaz, T. Examples of using dynamic constructible closure. In *Proceedings of the IMACS Symposium on Symbolic Computation* (1993).
- [13] Hearn, A. C., and Fitch, J. P. *Reduce User's Manual for Version 3.6*. RAND, Santa Monica, CA 90407-2138, July 1995. RAND Publication CP78.
- [14] Hong, H., Collins, G. E., Johnson, J. R., and Encarnacion, M. J. QEPCAD interactive version 12. Kindly communicated to us by Hoon Hong, Sept. 1993.
- [15] Loos, R., and Weispfenning, V. Applying linear quantifier elimination. *The Computer Journal* 36, 5 (1993), 450–462. Special issue on computational quantifier elimination.
- [16] Melenk, H. Reduce symbolic mode primer. In *REDUCE 3.6 User's Guide for UNIX*. Konrad-Zuse-Institut, Berlin, 1995.
- [17] Tarski, A. A decision method for elementary algebra and geometry. Tech. rep., University of California, 1948. Second edn., rev. 1951.
- [18] Weispfenning, V. The complexity of linear problems in fields. *Journal of Symbolic Computation* 5, 1 (Feb. 1988), 3–27.
- [19] Weispfenning, V. Comprehensive Gröbner bases. *Journal of Symbolic Computation* 14 (July 1992), 1–29.
- [20] Weispfenning, V. Quantifier elimination for real algebra—the cubic case. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation in Oxford* (New York, July 1994), ACM Press, pp. 258–263.
- [21] Weispfenning, V. Quantifier elimination for real algebra—the quadratic case and beyond. To appear in AAECC.

16.28 IDEALS: Arithmetic for polynomial ideals

This package implements the basic arithmetic for polynomial ideals by exploiting the Gröbner bases package of REDUCE. In order to save computing time all intermediate Gröbner bases are stored internally such that time consuming repetitions are inhibited.

Author: Herbert Melenk.

16.28.1 Introduction

This package implements the basic arithmetic for polynomial ideals by exploiting the Gröbner bases package of REDUCE. In order to save computing time all intermediate Gröbner bases are stored internally such that time consuming repetitions are inhibited. A uniform setting facilitates the access.

16.28.2 Initialization

Prior to any computation the set of variables has to be declared by calling the operator *I_setting*. E.g. in order to initiate computations in the polynomial ring $Q[x, y, z]$ call

```
I_setting(x,y,z);
```

A subsequent call to *I_setting* allows one to select another set of variables; at the same time the internal data structures are cleared in order to free memory resources.

16.28.3 Bases

An ideal is represented by a basis (set of polynomials) tagged with the symbol *I*, e.g.

```
u := I(x*z-y**2, x**3-y*z);
```

Alternatively a list of polynomials can be used as input basis; however, all arithmetic results will be presented in the above form. The operator *ideal2list* allows one to convert an ideal basis into a conventional REDUCE list.

Operators

Because of syntactical restrictions in REDUCE, special operators have to be used for ideal arithmetic:

<code>.+</code>	ideal sum (infix)
<code>.*</code>	ideal product (infix)
<code>.:</code>	ideal quotient (infix)
<code>./</code>	ideal quotient (infix)
<code>.=</code>	ideal equality test (infix)
<code>subset</code>	ideal inclusion test (infix)
<code>intersection</code>	ideal intersection (prefix, binary)
<code>member</code>	test for membership in an ideal (infix: polynomial and ideal)
<code>gb</code>	Groebner basis of an ideal (prefix, unary)
<code>ideal2list</code>	convert ideal basis to polynomial list (prefix, unary)

Example:

```
I (x+y, x^2) .* I (x-z);

      2          2      2
I (X  + X*Y - X*Z - Y*Z, X*Y  - Y *Z)
```

The test operators return the values 1 (=true) or 0 (=false) such that they can be used in REDUCE *if – then – else* statements directly.

The results of *sum*, *product*, *quotient*, *intersection* are ideals represented by their Gröbner basis in the current setting and term order. The term order can be modified using the operator *torder* from the Gröbner package. Note that ideal equality cannot be tested with the REDUCE equal sign:

```
I (x, y)  = I (y, x)      is false
I (x, y) .= I (y, x)      is true
```

16.28.4 Algorithms

The operators *groebner*, *preduce* and *idealquotient* of the REDUCE Gröbner package support the basic algorithms:

$$GB(Iu_1, u_2 \dots) \rightarrow groebner(\{u_1, u_2 \dots\}, \{x, \dots\})$$

$$p \in I_1 \rightarrow p = 0 \bmod I_1$$

$$I_1 : I(p) \rightarrow (I_1 \cap I(p)) / p \text{ elementwise}$$

On top of these the Ideals package implements the following operations:

$$I(u_1, u_2 \dots) + I(v_1, v_2 \dots) \rightarrow GB(I(u_1, u_2 \dots, v_1, v_2 \dots))$$

$$I(u_1, u_2 \dots) * I(v_1, v_2 \dots) \rightarrow GB(I(u_1 * v_1, u_1 * v_2, \dots, u_2 * v_1, u_2 * v_2 \dots))$$

$$I_1 \cap I_2 \rightarrow Q[x, \dots] \cap GB_{lex}(t * I_1 + (1 - t) * I_2, \{t, x, \dots\})$$

$$I_1 : I(p_1, p_2, \dots) \rightarrow I_1 : I(p_1) \cap I_1 : I(p_2) \cap \dots$$

$$I_1 = I_2 \rightarrow GB(I_1) = GB(I_2)$$

$$I_1 \subseteq I_2 \rightarrow u_i \in I_2 \forall u_i \in I_1 = I(u_1, u_2 \dots)$$

16.28.5 Examples

Please consult the file `ideals.tst`.

16.29 INEQ: Support for solving inequalities

This package supports the operator **ineq.solve** that tries to solves single inequalities and sets of coupled inequalities.

Author: Herbert Melenk.

This package supports the operator **ineq.solve** that tries to solves single inequalities and sets of coupled inequalities. The following types of systems are supported ²⁵:

- only numeric coefficients (no parametric system),
- a linear system of mixed equations and $\leq - \geq$ inequalities, applying the method of Fourier and Motzkin ²⁶,
- a univariate inequality with $\leq, \geq, >$ or $<$ operator and polynomial or rational left-hand and right-hand sides, or a system of such inequalities with only one variable.

Syntax:

$$\text{INEQ_SOLVE}(\langle \text{expr} \rangle \quad [, \langle \text{vl} \rangle])$$

where $\langle \text{expr} \rangle$ is an inequality or a list of coupled inequalities and equations, and the optional argument $\langle \text{vl} \rangle$ is a single variable (kernel) or a list of variables (kernels). If not specified, they are extracted automatically from $\langle \text{expr} \rangle$. For multivariate input an explicit variable list specifies the elimination sequence: the last member is the most specific one.

An error message occurs if the input cannot be processed by the currently implemented algorithms.

The result is a list. It is empty if the system has no feasible solution. Otherwise the result presents the admissible ranges as set of equations where each variable is equated to one expression or to an interval. The most specific variable is the first one in the result list and each form contains only preceding variables (resolved form). The interval limits can be formal **max** or **min** expressions. Algebraic numbers are encoded as rounded number approximations.

Examples:

```
ineq_solve({(2*x^2+x-1)/(x-1) >= (x+1/2)^2, x>0});

{x=(0 .. 0.326583), x=(1 .. 2.56777)}
```

²⁵For linear optimization problems please use the operator **simplex** of the **linalg** package

²⁶described by G.B. Dantzig in *Linear Programming and Extensions*.

```

reg:=
{a + b - c>=0, a - b + c>=0, - a + b + c>=0, 0>=0, 2>=0,
 2*c - 2>=0, a - b + c>=0, a + b - c>=0, - a + b + c - 2>=0,
 2>=0, 0>=0, 2*b - 2>=0, k + 1>=0, - a - b - c + k>=0,
  - a - b - c + k + 2>=0, - 2*b + k>=0,
  - 2*c + k>=0, a + b + c - k>=0,
 2*b + 2*c - k - 2>=0, a + b + c - k>=0}$

ineq_solve (reg,{k,a,b,c});

{c=(1 .. infinity),

b=(1 .. infinity),

a=(max( - b + c,b - c) .. b + c - 2),

k=a + b + c}

```

16.30 INVBASE: A package for computing involutive bases

Involutive bases are a new tool for solving problems in connection with multivariate polynomials, such as solving systems of polynomial equations and analyzing polynomial ideals. An involutive basis of polynomial ideal is nothing but a special form of a redundant Gröbner basis. The construction of involutive bases reduces the problem of solving polynomial systems to simple linear algebra.

Authors: A.Yu. Zharkov and Yu.A. Blinkov.

16.30.1 Introduction

Involutive bases are a new tool for solving problems in connection with multivariate polynomials, such as solving systems of polynomial equations and analyzing polynomial ideals, see [1]. An involutive basis of polynomial ideal is nothing but a special form of a redundant Gröbner basis. The construction of involutive bases reduces the problem of solving polynomial systems to simple linear algebra.

The INVBASE package²⁷ calculates involutive bases of polynomial ideals using an algorithm described in [1] which may be considered as an alternative to the well-known Buchberger algorithm [2]. The package can be used over a variety of different coefficient domains, and for different variable and term orderings.

The algorithm implemented in the INVBASE package is proved to be valid for any zero-dimensional ideal (finite number of solutions) as well as for positive-dimensional ideals in generic form. However, the algorithm does not terminate for “sparse” positive-dimensional systems. In order to stop the process we use the maximum degree bound for the Gröbner bases of generic ideals in the total-degree term ordering established in [3]. In this case, it is reasonable to call the GROEBNER package with the answer of INVBASE as input information in order to compute the reduced Gröbner basis under the same variable and term ordering. Though the INVBASE package supports computing involutive bases in any admissible term ordering, it is reasonable to compute them only for the total-degree term orderings. The package includes a special algorithm for conversion of total-degree involutive bases into the triangular bases in the lexicographical term ordering that is desirable for finding solutions. Normally the sum of timings for these two computations is much less than the timing for direct computation of the lexicographical involutive bases. As a rule, the result of the conversion algorithm is a reduced Gröbner basis in the lexicographical term ordering. However, because of some gaps in the current version of the algorithm, there may be rare situations when the resulting triangular set does not possess the formal property of Gröbner bases. Anyway, we recommend using the GROEBNER package with the result of the

²⁷The REDUCE implementation has been supported by the Konrad-Zuse-Zentrum Berlin

conversion algorithm as input in order either to check the Gröbner bases property or to transform the result into a lexicographical Gröbner basis.

16.30.2 The Basic Operators

Term Ordering

The following term order modes are available:

REVGRADLEX, GRADLEX, LEX

These modes have the same meaning as for the GROEBNER package.

All orderings are based on an ordering among the variables. For each pair of variables an order relation $>$ must be defined, e.g. $x > y$. The term ordering mode as well as the order of variables are set by the operator

INVTORDER $< mode >, \{x_1, \dots, x_n\}$

where $< mode >$ is one of the term order modes listed above. The notion of $\{x_1, \dots, x_n\}$ as a list of variables at the same time means $x_1 > \dots > x_n$.

Example 1.

INVTORDER REVGRADLEX, \{x, y, z\}

sets the reverse graduated term ordering based on the variable order $x > y > z$.

The operator *INVTORDER* may be omitted. The default term order mode is *REVGRADLEX* and the default decreasing variable order is alphabetical (or, more generally, the default REDUCE kernel order). Furthermore, the list of variables in the *INVTORDER* may be omitted. In this case the default variable order is used.

Computing Involutive Bases

To compute the involutive basis of ideal generated by the set of polynomials $\{p_1, \dots, p_m\}$ one should type the command

INVBASE $\{p_1, \dots, p_m\}$

where p_i are polynomials in variables listed in the *INVTORDER* operator. If some kernels in p_i were not listed previously in the *INVTORDER* operator they are considered as parameters, i.e. they are considered part of the coefficients of polynomials. If *INVTORDER* was omitted, all the kernels in p_i are considered as variables with the default REDUCE kernel order.

The coefficients of polynomials p_i may be integers as well as rational numbers (or,

accordingly, polynomials and rational functions in the parametric case). The computations modulo prime numbers are also available. For this purpose one should type the REDUCE commands

$$ON MODULAR; SETMOD p;$$

where p is a prime number.

The value of the *INVBASE* function is a list of integer polynomials $\{g_1, \dots, g_n\}$ representing an involutive basis of a given ideal.

Example 2.

```

INVTORDER REVGRADLEX, {x, y, z};
g := INVBASE {4 * x ** 2 + x * y ** 2 - z + 1/4, 2 * x + y ** 2 * z + 1/2,
x ** 2 * z - 1/2 * x - y ** 2};

```

The resulting involutive basis in the reverse graduate ordering is

```

g := {
  8 * x * y * z^3 - 2 * x * y * z^2 + 4 * y^3 -
  4 * y * z^2 + 16 * x * y + 17 * y * z - 4 * y,
  8 * y^4 - 8 * x * z^2 - 256 * y^2 + 2 * x * z + 64 * z^2 - 96 * x + 20 * z - 9,
  2 * y^3 * z + 4 * x * y + y,
  8 * x * z^3 - 2 * x * z^2 + 4 * y^2 - 4 * z^2 + 16 * x + 17 * z - 4,
  -4 * y * z^3 - 8 * y^3 + 6 * x * y * z + y * z^2 - 36 * x * y - 8 * y,
  4 * x * y^2 + 32 * y^2 - 8 * z^2 + 12 * x - 2 * z + 1,
  2 * y^2 * z + 4 * x + 1,
  -4 * z^3 - 8 * y^2 + 6 * x * z + z^2 - 36 * x - 8,
  8 * x^2 - 16 * y^2 + 4 * z^2 - 6 * x - z }

```

To convert it into a lexicographical Gröbner basis one should type

```
h := INVLEX g;
```

The result is

```

h := {
  3976 * x + 37104 * z^6 - 600 * z^5 + 2111 * z^4 +
  122062 * z^3 + 232833 * z^2 - 680336 * z + 288814,
  1988 * y^2 - 76752 * z^6 + 1272 * z^5 - 4197 * z^4 -
  251555 * z^3 - 481837 * z^2 + 1407741 * z - 595666,
  16 * z^7 - 8 * z^6 + z^5 + 52 * z^4 + 75 * z^3 - 342 * z^2 + 266 * z - 60 }

```

In the case of “sparse” positive-dimensional system when the involutive basis in the sense of [1] does not exist, you get the error message

```
***** MAXIMUM DEGREE BOUND EXCEEDED
```

The resulting list of polynomials which is not an involutive basis is stored in the share variable INVTEMPBASIS. In this case it is reasonable to call the GROEBNER package with the value of INVTEMPBASIS as input under the same variable and term ordering.

Bibliography

- [1] Zharkov A.Yu., Blinkov Yu.A. Involution Approach to Solving Systems of Algebraic Equations. Proceedings of the IMACS '93, 1993, 11-16.
- [2] Buchberger B. Gröbner bases: an Algorithmic Method in Polynomial Ideal Theory. In: (Bose N.K., ed.) Recent Trends in Multidimensional System Theory, Reidel, 1985.
- [3] Lazard D. Gröbner Bases, Gaussian Elimination and Resolution of Systems of Algebraic Equations. Proceedings of EUROCAL '83. Lecture Notes in Computer Science 162, Springer 1983, 146-157.

16.31 LAPLACE: Laplace transforms

This package can calculate ordinary and inverse Laplace transforms of expressions. Documentation is in plain text.

Authors: C. Kazasov, M. Spiridonova, V. Tomov.

Reference: **Christomir Kazasov**, Laplace Transformations in REDUCE 3, Proc. Eurocal '87, Lecture Notes in Comp. Sci., Springer-Verlag (1987) 132-133.

Some hints on how to use to use this package:

Syntax:

LAPLACE ($\langle exp \rangle$, $\langle var - s \rangle$, $\langle var - t \rangle$)

INVLAP ($\langle exp \rangle$, $\langle var - s \rangle$, $\langle var - t \rangle$)

where $\langle exp \rangle$ is the expression to be transformed, $\langle var - s \rangle$ is the source variable (in most cases $\langle exp \rangle$ depends explicitly of this variable) and $\langle var - t \rangle$ is the target variable. If $\langle var - t \rangle$ is omitted, the package uses an internal variable `lp!&` or `il!&`, respectively.

The following switches can be used to control the transformations:

- `lmon:` If on, `sin`, `cos`, `sinh` and `cosh` are converted by `LAPLACE` into exponentials,
- `lhyp:` If on, expressions \tilde{e}^x are converted by `INVLAP` into hyperbolic functions `sinh` and `cosh`,
- `ltrig:` If on, expressions \tilde{e}^x are converted by `INVLAP` into trigonometric functions `sin` and `cos`.

The system can be extended by adding Laplace transformation rules for single functions by rules or rule sets. In such a rule the source variable **MUST** be free, the target variable **MUST** be `il!&` for `LAPLACE` and `lp!&` for `INVLAP` and the third parameter should be omitted. Also rules for transforming derivatives are entered in such a form.

Examples:

```

let {laplace(log(~x),x) => -log(gam * il!&)/il!&,
    invlap(log(gam * ~x)/x,x) => -log(lp!&)};

operator f;

let{
    laplace(df(f(~x),x),x) => il!&*laplace(f(x),x) - sub(x=0,f(x)),
    laplace(df(f(~x),x,~n),x) => il!&**n*laplace(f(x),x) -
    for i:=n-1 step -1 until 0 sum
    sub(x=0, df(f(x),x,n-1-i)) * il!&**i
    when fixp n,
    laplace(f(~x),x) = f(il!&)
};

```

Remarks about some functions:

The DELTA and GAMMA functions are known.

ONE is the name of the unit step function.

INTL is a parametrized integral function

$$\text{intl}(< \text{expr} >, < \text{var} >, 0, < \text{obj.var} >)$$

which means "Integral of $< \text{expr} >$ wrt. $< \text{var} >$ taken from 0 to $< \text{obj.var} >$ ",
 e.g. $\text{intl}(2*y^2, y, 0, x)$ which is formally a function in x .

We recommend reading the file LAPLACE.TST for a further introduction.

16.32 LIE: Functions for the classification of real n-dimensional Lie algebras

LIE is a package of functions for the classification of real n-dimensional Lie algebras. It consists of two modules: **liendmc1** and **lie1234**. With the help of the functions in the **liendmc1** module, real n-dimensional Lie algebras L with a derived algebra $L^{(1)}$ of dimension 1 can be classified.

Authors: Carsten and Franziska Schöbel.

LIE is a package of functions for the classification of real n-dimensional Lie algebras. It consists of two modules: **liendmc1** and **lie1234**.

liendmc1

With the help of the functions in this module real n-dimensional Lie algebras L with a derived algebra $L^{(1)}$ of dimension 1 can be classified. L has to be defined by its structure constants c_{ij}^k in the basis $\{X_1, \dots, X_n\}$ with $[X_i, X_j] = c_{ij}^k X_k$. The user must define an ARRAY LIENSTRUCIN(n, n, n) with n being the dimension of the Lie algebra L . The structure constants $\text{LIENSTRUCIN}(i, j, k) := c_{ij}^k$ for $i < j$ should be given. Then the procedure LIENDIMCOM1 can be called. Its syntax is:

LIENDIMCOM1 (<number>) .

<number> corresponds to the dimension n . The procedure simplifies the structure of L performing real linear transformations. The returned value is a list of the form

- (i) {LIE_ALGEBRA(2), COMMUTATIVE(n-2)} or
- (ii) {HEISENBERG(k), COMMUTATIVE(n-k)}

with $3 \leq k \leq n$, k odd.

The concepts correspond to the following theorem ($\text{LIE_ALGEBRA}(2) \rightarrow L_2$, $\text{HEISENBERG}(k) \rightarrow H_k$ and $\text{COMMUTATIVE}(n-k) \rightarrow C_{n-k}$):

Theorem. Every real n -dimensional Lie algebra L with a 1-dimensional derived algebra can be decomposed into one of the following forms:

- (i) $C(L) \cap L^{(1)} = \{0\} : L_2 \oplus C_{n-2}$ or
- (ii) $C(L) \cap L^{(1)} = L^{(1)} : H_k \oplus C_{n-k} \quad (k = 2r - 1, r \geq 2)$, with

1. $C(L) = C_j \oplus (L^{(1)} \cap C(L))$ and $\dim C_j = j$,
2. L_2 is generated by Y_1, Y_2 with $[Y_1, Y_2] = Y_1$,
3. H_k is generated by $\{Y_1, \dots, Y_k\}$ with
 $[Y_2, Y_3] = \dots = [Y_{k-1}, Y_k] = Y_1$.

(cf. [?])

The returned list is also stored as `LIE_LIST`. The matrix `LIENTRANS` gives the transformation from the given basis $\{X_1, \dots, X_n\}$ into the standard basis $\{Y_1, \dots, Y_n\}$: $Y_j = (\text{LIENTRANS})_j^k X_k$.

A more detailed output can be obtained by turning on the switch `TR_LIE`:

```
ON TR_LIE;
```

before the procedure `LIENDIMCOM1` is called.

The returned list could be an input for a data bank in which mathematical relevant properties of the obtained Lie algebras are stored.

lie1234

This part of the package classifies real low-dimensional Lie algebras L of the dimension $n := \dim L = 1, 2, 3, 4$. L is also given by its structure constants c_{ij}^k in the basis $\{X_1, \dots, X_n\}$ with $[X_i, X_j] = c_{ij}^k X_k$. An ARRAY `LIESTRIN`(n, n, n) has to be defined and `LIESTRIN`(i, j, k):= c_{ij}^k for $i < j$ should be given. Then the procedure `LIECLASS` can be performed whose syntax is:

```
LIECLASS (<number>) .
```

`<number>` should be the dimension of the Lie algebra L . The procedure stepwise simplifies the commutator relations of L using properties of invariance like the dimension of the centre, of the derived algebra, unimodularity etc. The returned value has the form:

```
{ LIEALG (n) , COMTAB (m) } ,
```

where m corresponds to the number of the standard form (basis: $\{Y_1, \dots, Y_n\}$) in an enumeration scheme. The corresponding enumeration schemes are listed below (cf. [?],[?]). In case that the standard form in the enumeration scheme depends on one (or two) parameter(s) p_1 (and p_2) the list is expanded to:

```
{ LIEALG (n) , COMTAB (m) , p1 , p2 } .
```

This returned value is also stored as `LIE_CLASS`. The linear transformation from the basis $\{X_1, \dots, X_n\}$ into the basis of the standard form $\{Y_1, \dots, Y_n\}$ is given by the matrix `LIEMAT`: $Y_j = (\text{LIEMAT})_j^k X_k$.

By turning on the switch TR_LIE:

```
ON TR_LIE;
```

before the procedure LIECLASS is called the output contains not only the list LIE_CLASS but also the non-vanishing commutator relations in the standard form.

By the value m and the parameters further examinations of the Lie algebra are possible, especially if in a data bank mathematical relevant properties of the enumerated standard forms are stored.

Enumeration schemes for lie1234

returned list LIE_CLASS	the corresponding commutator relations
LIEALG(1),COMTAB(0)	commutative case
LIEALG(2),COMTAB(0)	commutative case
LIEALG(2),COMTAB(1)	$[Y_1, Y_2] = Y_2$
LIEALG(3),COMTAB(0)	commutative case
LIEALG(3),COMTAB(1)	$[Y_1, Y_2] = Y_3$
LIEALG(3),COMTAB(2)	$[Y_1, Y_3] = Y_3$
LIEALG(3),COMTAB(3)	$[Y_1, Y_3] = Y_1, [Y_2, Y_3] = Y_2$
LIEALG(3),COMTAB(4)	$[Y_1, Y_3] = Y_2, [Y_2, Y_3] = Y_1$
LIEALG(3),COMTAB(5)	$[Y_1, Y_3] = -Y_2, [Y_2, Y_3] = Y_1$
LIEALG(3),COMTAB(6)	$[Y_1, Y_3] = -Y_1 + p_1 Y_2, [Y_2, Y_3] = Y_1, p_1 \neq 0$
LIEALG(3),COMTAB(7)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = -Y_2, [Y_2, Y_3] = Y_1$
LIEALG(3),COMTAB(8)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = Y_2, [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(0)	commutative case
LIEALG(4),COMTAB(1)	$[Y_1, Y_4] = Y_1$
LIEALG(4),COMTAB(2)	$[Y_2, Y_4] = Y_1$
LIEALG(4),COMTAB(3)	$[Y_1, Y_3] = Y_1, [Y_2, Y_4] = Y_2$
LIEALG(4),COMTAB(4)	$[Y_1, Y_3] = -Y_2, [Y_2, Y_4] = Y_2,$ $[Y_1, Y_4] = [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(5)	$[Y_2, Y_4] = Y_2, [Y_1, Y_4] = [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(6)	$[Y_2, Y_4] = Y_1, [Y_3, Y_4] = Y_2$
LIEALG(4),COMTAB(7)	$[Y_2, Y_4] = Y_2, [Y_3, Y_4] = Y_1$
LIEALG(4),COMTAB(8)	$[Y_1, Y_4] = -Y_2, [Y_2, Y_4] = Y_1$
LIEALG(4),COMTAB(9)	$[Y_1, Y_4] = -Y_1 + p_1 Y_2, [Y_2, Y_4] = Y_1, p_1 \neq 0$
LIEALG(4),COMTAB(10)	$[Y_1, Y_4] = Y_1, [Y_2, Y_4] = Y_2$
LIEALG(4),COMTAB(11)	$[Y_1, Y_4] = Y_2, [Y_2, Y_4] = Y_1$

returned list LIE_CLASS	the corresponding commutator relations
LIEALG(4),COMTAB(12)	$[Y_1, Y_4] = Y_1 + Y_2, [Y_2, Y_4] = Y_2 + Y_3,$ $[Y_3, Y_4] = Y_3$
LIEALG(4),COMTAB(13)	$[Y_1, Y_4] = Y_1, [Y_2, Y_4] = p_1 Y_2, [Y_3, Y_4] = p_2 Y_3,$ $p_1, p_2 \neq 0$
LIEALG(4),COMTAB(14)	$[Y_1, Y_4] = p_1 Y_1 + Y_2, [Y_2, Y_4] = -Y_1 + p_1 Y_2,$ $[Y_3, Y_4] = p_2 Y_3, p_2 \neq 0$
LIEALG(4),COMTAB(15)	$[Y_1, Y_4] = p_1 Y_1 + Y_2, [Y_2, Y_4] = p_1 Y_2,$ $[Y_3, Y_4] = Y_3, p_1 \neq 0$
LIEALG(4),COMTAB(16)	$[Y_1, Y_4] = 2Y_1, [Y_2, Y_3] = Y_1,$ $[Y_2, Y_4] = (1 + p_1)Y_2, [Y_3, Y_4] = (1 - p_1)Y_3,$ $p_1 \geq 0$
LIEALG(4),COMTAB(17)	$[Y_1, Y_4] = 2Y_1, [Y_2, Y_3] = Y_1,$ $[Y_2, Y_4] = Y_2 - p_1 Y_3, [Y_3, Y_4] = p_1 Y_2 + Y_3,$ $p_1 \neq 0$
LIEALG(4),COMTAB(18)	$[Y_1, Y_4] = 2Y_1, [Y_2, Y_3] = Y_1,$ $[Y_2, Y_4] = Y_2 + Y_3, [Y_3, Y_4] = Y_3$
LIEALG(4),COMTAB(19)	$[Y_2, Y_3] = Y_1, [Y_2, Y_4] = Y_3, [Y_3, Y_4] = Y_2$
LIEALG(4),COMTAB(20)	$[Y_2, Y_3] = Y_1, [Y_2, Y_4] = -Y_3, [Y_3, Y_4] = Y_2$
LIEALG(4),COMTAB(21)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = -Y_2, [Y_2, Y_3] = Y_1$
LIEALG(4),COMTAB(22)	$[Y_1, Y_2] = Y_3, [Y_1, Y_3] = Y_2, [Y_2, Y_3] = Y_1$

16.33 LIMITS: A package for finding limits

This package loads automatically.

Author: Stanley L. Kameny.

LIMITS is a fast limit package for REDUCE for functions which are continuous except for computable poles and singularities, based on some earlier work by Ian Cohen and John P. Fitch. The Truncated Power Series package is used for non-critical points, at which the value of the function is the constant term in the expansion around that point. l'Hôpital's rule is used in critical cases, with preprocessing of $\infty - \infty$ forms and reformatting of product forms in order to apply l'Hôpital's rule. A limited amount of bounded arithmetic is also employed where applicable.

16.33.1 Normal entry points

`LIMIT(EXPRN:algebraic, VAR:kernel, LIMPOINT:algebraic):algebraic`

This is the standard way of calling limit, applying all of the methods. The result is the limit of EXPRN as VAR approaches LIMPOINT.

16.33.2 Direction-dependent limits

`LIMIT!+(EXPRN:algebraic, VAR:kernel, LIMPOINT:algebraic):algebraic`
`LIMIT!-(EXPRN:algebraic, VAR:kernel, LIMPOINT:algebraic):algebraic`

If the limit depends upon the direction of approach to the LIMPOINT, the functions `LIMIT!+` and `LIMIT!-` may be used. They are defined by:

$$\text{LIMIT!+ (LIMIT!-)} (\text{EXP}, \text{VAR}, \text{LIMPOINT}) \rightarrow \text{LIMIT}(\text{EXP}^*, \epsilon, 0),$$

$$\text{EXP}^* = \text{sub}(\text{VAR} = \text{VAR} + (-)\epsilon^2, \text{EXP})$$

16.33.3 Diagnostic Functions

`LIMIT0(EXPRN:algebraic, VAR:kernel, LIMPOINT:algebraic):algebraic`

This function will use all parts of the limits package, but it does not combine log terms before taking limits, so it may fail if there is a sum of log terms which have a removable singularity in some of the terms.

`LIMIT1(EXPRN:algebraic, VAR:kernel, LIMPOINT:algebraic):algebraic`

This function uses the TPS branch only, and will fail if the limit point is singular.

```
LIMIT2 (TOP:algebraic,  
        BOT:algebraic,  
        VAR:kernel,  
        LIMPOINT:algebraic):algebraic
```

This function applies l'Hôpital's rule to the quotient (TOP/BOT).

16.34 LINALG: Linear algebra package

This package provides a selection of functions that are useful in the world of linear algebra.

Author: Matt Rebbeck.

16.34.1 Introduction

This package provides a selection of functions that are useful in the world of linear algebra. These functions are described alphabetically in subsection [16.34.3](#) and are labelled [16.34.3.1](#) to [16.34.3.51](#). They can be classified into four sections (n.b: the numbers after the dots signify the function label in section [16.34.3](#)).

Contributions to this package have been made by Walter Tietze (ZIB).

Basic matrix handling

add_columns	...	16.34.3	add_rows	...	3.2
add_to_columns	...	3.3	add_to_rows	...	3.4
augment_columns	...	3.5	char_poly	...	3.9
column_dim	...	3.12	copy_into	...	3.14
diagonal	...	3.15	extend	...	3.16
find_companion	...	3.17	get_columns	...	3.18
get_rows	...	3.19	hermitian_tp	...	3.21
matrix_augment	...	3.28	matrix_stack	...	3.30
minor	...	3.31	mult_columns	...	3.32
mult_rows	...	3.33	pivot	...	3.34
remove_columns	...	3.37	remove_rows	...	3.38
row_dim	...	3.39	rows_pivot	...	3.40
stack_rows	...	3.43	sub_matrix	...	3.44
swap_columns	...	3.46	swap_entries	...	3.47
swap_rows	...	3.48			

Constructors

Functions that create matrices.

band_matrix	...	3. 6	block_matrix	...	3. 7
char_matrix	...	3. 8	coeff_matrix	...	3. 11
companion	...	3. 13	hessian	...	3. 22
hilbert	...	3. 23	mat_jacobian	...	3. 24
jordan_block	...	3. 25	make_identity	...	3. 27
random_matrix	...	3. 36	toeplitz	...	3. 50
Vandermonde	...	3. 51	Kronecker_Product	...	3. 52

High level algorithms

char_poly	...	3.9	cholesky	...	3.10
gram_schmidt	...	3.20	lu_decom	...	3.26
pseudo_inverse	...	3.35	simplex	...	3.41
svd	...	3.45	triang_adjoint	...	3.51

There is a separate NORMFORM[1] package for computing the following matrix normal forms in REDUCE.

smithex, smithex_int, frobenius, ratjordan, jordansymbolic, jordan.

Predicates

matrixp	...	3.29	squarep	...	3.42
symmetricp	...	3.49			

Note on examples:

In the examples the matrix \mathcal{A} will be

$$\mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Notation

Throughout \mathcal{I} is used to indicate the identity matrix and \mathcal{A}^T to indicate the transpose of the matrix \mathcal{A} .

16.34.2 Getting started

If you have not used matrices within REDUCE before then the following may be helpful.

Creating matrices

Initialisation of matrices takes the following syntax:

```
mat1 := mat((a,b,c),(d,e,f),(g,h,i));
```

will produce

$$mat1 := \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

Getting at the entries

The (i,j)'th entry can be accessed by:

```
mat1(i,j);
```

Loading the linear_algebra package

The package is loaded by:

```
load_package linalg;
```

16.34.3 What's available

add_columns, add_rows

```
add_columns(A, c1, c2, expr);
```

\mathcal{A} :- a matrix.

c1,c2 :- positive integers.

expr :- a scalar expression.

Synopsis:

add_columns replaces column c2 of \mathcal{A} by $\text{expr} * \text{column}(\mathcal{A}, c1) + \text{column}(\mathcal{A}, c2)$.

add_rows performs the equivalent task on the rows of \mathcal{A} .

Examples:

$$\text{add_columns}(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & x+2 & 3 \\ 4 & 4*x+5 & 6 \\ 7 & 7*x+8 & 9 \end{pmatrix}$$

$$\text{add_rows}(\mathcal{A}, 2, 3, 5) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 27 & 33 & 39 \end{pmatrix}$$

Related functions:

`add_to_columns`, `add_to_rows`, `mult_columns`, `mult_rows`.

add_rows

see: `add_columns`.

add_to_columns, add_to_rows

`add_to_columns` (\mathcal{A} , `column_list`, `expr`);

\mathcal{A} :- a matrix.

`column_list` :- a positive integer or a list of positive integers.

`expr` :- a scalar expression.

Synopsis:

`add_to_columns` adds `expr` to each column specified in `column_list` of \mathcal{A} .

`add_to_rows` performs the equivalent task on the rows of \mathcal{A} .

Examples:

$$\text{add_to_columns}(\mathcal{A}, \{1, 2\}, 10) = \begin{pmatrix} 11 & 12 & 3 \\ 14 & 15 & 6 \\ 17 & 18 & 9 \end{pmatrix}$$

$$\text{add_to_rows}(\mathcal{A}, 2, -x) = \begin{pmatrix} 1 & 2 & 3 \\ -x+4 & -x+5 & -x+6 \\ 7 & 8 & 9 \end{pmatrix}$$

Related functions:

`add_columns`, `add_rows`, `mult_rows`, `mult_columns`.

add_to_rows

see: `add_to_columns`.

augment_columns, stack_rows

```
augment_columns( $\mathcal{A}$ , column_list);
```

\mathcal{A} :- a matrix.

column_list :- either a positive integer or a list of positive integers.

Synopsis:

`augment_columns` gets hold of the columns of \mathcal{A} specified in `column_list` and sticks them together.

`stack_rows` performs the same task on rows of \mathcal{A} .

Examples:

$$\text{augment_columns}(\mathcal{A}, \{1, 2\}) = \begin{pmatrix} 1 & 2 \\ 4 & 5 \\ 7 & 8 \end{pmatrix}$$

$$\text{stack_rows}(\mathcal{A}, \{1, 3\}) = \begin{pmatrix} 1 & 2 & 3 \\ 7 & 8 & 9 \end{pmatrix}$$

Related functions:

`get_columns`, `get_rows`, `sub_matrix`.

band_matrix

```
band_matrix(expr_list, square_size);
```

`expr_list` :- either a single scalar expression or a list of an odd number of scalar expressions.

`square_size` :- a positive integer.

Synopsis:

`band_matrix` creates a square matrix of dimension `square_size`. The diagonal consists of the middle `expr` of the `expr_list`. The `exprs` to the left of this fill the required number of sub-diagonals and the `exprs` to the right the super-diagonals.

Examples:

$$\text{band_matrix}(\{x, y, z\}, 6) = \begin{pmatrix} y & z & 0 & 0 & 0 & 0 \\ x & y & z & 0 & 0 & 0 \\ 0 & x & y & z & 0 & 0 \\ 0 & 0 & x & y & z & 0 \\ 0 & 0 & 0 & x & y & z \\ 0 & 0 & 0 & 0 & x & y \end{pmatrix}$$

Related functions:

`diagonal.`

block_matrix

`block_matrix(r, c, matrix_list);`

`r, c` :- positive integers.

`matrix_list` :- a list of matrices.

Synopsis:

`block_matrix` creates a matrix that consists of `r` by `c` matrices filled from the `matrix_list` row wise.

Examples:

$$\mathcal{B} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad \mathcal{C} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}, \quad \mathcal{D} = \begin{pmatrix} 22 & 33 \\ 44 & 55 \end{pmatrix}$$

$$\text{block_matrix}(2, 3, \{\mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{D}, \mathcal{C}, \mathcal{B}\}) = \begin{pmatrix} 1 & 0 & 5 & 22 & 33 \\ 0 & 1 & 5 & 44 & 55 \\ 22 & 33 & 5 & 1 & 0 \\ 44 & 55 & 5 & 0 & 1 \end{pmatrix}$$

char_matrix

`char_matrix(\mathcal{A} , λ);`

\mathcal{A} :- a square matrix.

λ :- a symbol or algebraic expression.

Synopsis:

`char_matrix` creates the characteristic matrix \mathcal{C} of \mathcal{A} .

This is $\mathcal{C} = \lambda * \mathcal{I} - \mathcal{A}$.

Examples:

$$\text{char_matrix}(\mathcal{A}, x) = \begin{pmatrix} x-1 & -2 & -3 \\ -4 & x-5 & -6 \\ -7 & -8 & x-9 \end{pmatrix}$$

Related functions:

`char_poly`.

char_poly

`char_poly(A, λ);`

\mathcal{A} :- a square matrix.

λ :- a symbol or algebraic expression.

Synopsis:

`char_poly` finds the characteristic polynomial of \mathcal{A} .

This is the determinant of $\lambda * \mathcal{I} - \mathcal{A}$.

Examples:

$$\text{char_poly}(A, x) = x^3 - 15 * x^2 - 18 * x$$

Related functions:

`char_matrix`.

cholesky

`cholesky(A);`

\mathcal{A} :- a positive definite matrix containing numeric entries.

Synopsis:

`cholesky` computes the cholesky decomposition of \mathcal{A} .

It returns $\{\mathcal{L}, \mathcal{U}\}$ where \mathcal{L} is a lower matrix, \mathcal{U} is an upper matrix, $\mathcal{A} = \mathcal{L}\mathcal{U}$, and $\mathcal{U} = \mathcal{L}^T$.

Examples:

$$\mathcal{F} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$\text{cholesky}(\mathcal{F}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 1 & \sqrt{2} & 0 \\ 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}, \begin{pmatrix} 1 & 1 & 0 \\ 0 & \sqrt{2} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \right\}$$

Related functions:

`lu_decom.`

coeff_matrix

`coeff_matrix({lin_eqn1, lin_eqn2, ..., lin_eqnn});` ²⁸

`lin_eqn1, lin_eqn2, ..., lin_eqnn` :- linear equations. Can be of the form *equation = number* or just *equation*.

Synopsis:

`coeff_matrix` creates the coefficient matrix \mathcal{C} of the linear equations.

It returns $\{\mathcal{C}, \mathcal{X}, \mathcal{B}\}$ such that $\mathcal{C}\mathcal{X} = \mathcal{B}$.

Examples:

`coeff_matrix({x + y + 4 * z = 10, y + x - z = 20, x + y + 4}) =`

$$\left\{ \begin{pmatrix} 4 & 1 & 1 \\ -1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}, \begin{pmatrix} z \\ y \\ x \end{pmatrix}, \begin{pmatrix} 10 \\ 20 \\ -4 \end{pmatrix} \right\}$$

column_dim, row_dim

`column_dim(\mathcal{A});`

\mathcal{A} :- a matrix.

Synopsis:

`column_dim` finds the column dimension of \mathcal{A} .

`row_dim` finds the row dimension of \mathcal{A} .

Examples:

`column_dim(\mathcal{A}) = 3`

²⁸If you're feeling lazy then the `{}`'s can be omitted.

companion

```
companion(poly, x);
```

poly :- a monic univariate polynomial in x.

x :- the variable.

Synopsis:

companion creates the companion matrix \mathcal{C} of poly.

This is the square matrix of dimension n, where n is the degree of poly w.r.t. x.

The entries of \mathcal{C} are: $\mathcal{C}(i,n) = -\text{coeffn}(\text{poly}, x, i-1)$ for $i = 1 \dots n$, $\mathcal{C}(i,i-1) = 1$ for $i = 2 \dots n$ and the rest are 0.

Examples:

$$\text{companion}(x^4 + 17 * x^3 - 9 * x^2 + 11, x) = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

Related functions:

find.companion.

copy_into

```
copy_into(A, B, r, c);
```

A, B :- matrices.

r, c :- positive integers.

Synopsis:

copy_into copies matrix A into B with A(1,1) at B(r,c).

Examples:

$$\mathcal{G} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\text{copy_into}(\mathcal{A}, \mathcal{G}, 1, 2) = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 4 & 5 & 6 \\ 0 & 7 & 8 & 9 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Related functions:

`augment_columns`, `extend`, `matrix_augment`, `matrix_stack`, `stack_rows`, `sub_matrix`.

diagonal

`diagonal({mat1, mat2, ..., matn});`²⁹

`mat1, mat2, ..., matn` :- each can be either a scalar expr or a square matrix.

Synopsis:

`diagonal` creates a matrix that contains the input on the diagonal.

Examples:

$$\mathcal{H} = \begin{pmatrix} 66 & 77 \\ 88 & 99 \end{pmatrix}$$

$$\text{diagonal}(\{\mathcal{A}, x, \mathcal{H}\}) = \begin{pmatrix} 1 & 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 & 0 \\ 7 & 8 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 66 & 77 \\ 0 & 0 & 0 & 0 & 88 & 99 \end{pmatrix}$$

Related functions:

`jordan_block`.

extend

`extend(\mathcal{A} , r, c, expr);`

\mathcal{A} :- a matrix.

r, c :- positive integers.

expr :- algebraic expression or symbol.

Synopsis:

`extend` returns a copy of \mathcal{A} that has been extended by r rows and c columns. The new entries are made equal to expr.

Examples:

²⁹If you're feeling lazy then the {}'s can be omitted.

$$\text{extend}(\mathcal{A}, 1, 2, x) = \begin{pmatrix} 1 & 2 & 3 & x & x \\ 4 & 5 & 6 & x & x \\ 7 & 8 & 9 & x & x \\ x & x & x & x & x \end{pmatrix}$$

Related functions:

`copy_into`, `matrix_augment`, `matrix_stack`, `remove_columns`,
`remove_rows`.

find_companion

`find_companion(\mathcal{A} , x);`

\mathcal{A} :- a matrix.

x :- the variable.

Synopsis:

Given a companion matrix, `find_companion` finds the polynomial from which it was made.

Examples:

$$\mathcal{C} = \begin{pmatrix} 0 & 0 & 0 & -11 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 9 \\ 0 & 0 & 1 & -17 \end{pmatrix}$$

$$\text{find_companion}(\mathcal{C}, x) = x^4 + 17 * x^3 - 9 * x^2 + 11$$

Related functions:

`companion`.

get_columns, get_rows

`get_columns(\mathcal{A} , column_list);`

\mathcal{A} :- a matrix.

c :- either a positive integer or a list of positive integers.

Synopsis:

`get_columns` removes the columns of \mathcal{A} specified in `column_list` and returns them as a list of column matrices.

`get_rows` performs the same task on the rows of \mathcal{A} .

Examples:

$$\text{get_columns}(\mathcal{A}, \{1, 3\}) = \left\{ \begin{pmatrix} 1 \\ 4 \\ 7 \end{pmatrix}, \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix} \right\}$$

$$\text{get_rows}(\mathcal{A}, 2) = \{ (4 \ 5 \ 6) \}$$

Related functions:

`augment_columns`, `stack_rows`, `sub_matrix`.

get_rows

see: `get_columns`.

gram_schmidt

`gram_schmidt({vec1, vec2, ..., vecn});` ³⁰

`vec1, vec2, ..., vecn` :- linearly independent vectors. Each vector must be written as a list, eg: {1,0,0}.

Synopsis:

`gram_schmidt` performs the `gram_schmidt` orthonormalisation on the input vectors.

It returns a list of orthogonal normalised vectors.

Examples:

$$\text{gram_schmidt}(\{\{1, 0, 0\}, \{1, 1, 0\}, \{1, 1, 1\}\}) = \{\{1, 0, 0\}, \{0, 1, 0\}, \{0, 0, 1\}\}$$

$$\text{gram_schmidt}(\{\{1, 2\}, \{3, 4\}\}) = \{\{\frac{1}{\sqrt{5}}, \frac{2}{\sqrt{5}}\}, \{\frac{2*\sqrt{5}}{5}, \frac{-\sqrt{5}}{5}\}\}$$

hermitian_tp

`hermitian_tp(\mathcal{A});`

\mathcal{A} :- a matrix.

Synopsis:

³⁰If you're feeling lazy then the {}'s can be omitted.

`hermitian_tp` computes the hermitian transpose of \mathcal{A} .

This is a matrix in which the (i,j) 'th entry is the conjugate of the (j,i) 'th entry of \mathcal{A} .

Examples:

$$\mathcal{J} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 4 & 5 & 2 \\ 1 & i & 0 \end{pmatrix}$$

$$\text{hermitian_tp}(\mathcal{J}) = \begin{pmatrix} -i+1 & 4 & 1 \\ -i+2 & 5 & -i \\ -i+3 & 2 & 0 \end{pmatrix}$$

Related functions:

`tp`³¹.

hessian

`hessian(expr, variable_list);`

`expr` :- a scalar expression.

`variable_list` :- either a single variable or a list of variables.

Synopsis:

`hessian` computes the hessian matrix of `expr` w.r.t. the variables in `variable_list`.

This is an n by n matrix where n is the number of variables and the (i,j) 'th entry is `df(expr,variable_list(i),variable_list(j))`.

Examples:

$$\text{hessian}(x * y * z + x^2, \{w, x, y, z\}) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & z & y \\ 0 & z & 0 & x \\ 0 & y & x & 0 \end{pmatrix}$$

Related functions:

`df`³².

³¹standard reduce call for the transpose of a matrix - see REDUCE User's Manual[2].

³²standard reduce call for differentiation - see REDUCE User's Manual[2]

hilbert

```
hilbert(square_size, expr);
```

square_size :- a positive integer.
 expr :- an algebraic expression.

Synopsis:

hilbert computes the square hilbert matrix of dimension square_size.

This is the symmetric matrix in which the (i,j)'th entry is 1/(i+j-expr).

Examples:

$$\text{hilbert}(3, y + x) = \begin{pmatrix} \frac{-1}{x+y-2} & \frac{-1}{x+y-3} & \frac{-1}{x+y-4} \\ \frac{-1}{x+y-3} & \frac{-1}{x+y-4} & \frac{-1}{x+y-5} \\ \frac{-1}{x+y-4} & \frac{-1}{x+y-5} & \frac{-1}{x+y-6} \end{pmatrix}$$

jacobian

```
mat_jacobian(expr_list, variable_list);
```

expr_list :- either a single algebraic expression or a list of algebraic expressions.
 variable_list :- either a single variable or a list of variables.

Synopsis:

mat_jacobian computes the jacobian matrix of expr_list w.r.t. variable_list.

This is a matrix whose (i,j)'th entry is df(expr_list(i), variable_list(j)).

The matrix is n by m where n is the number of variables and m the number of expressions.

Examples:

```
mat_jacobian({x^4, x * y^2, x * y * z^3}, {w, x, y, z}) =
```

$$\begin{pmatrix} 0 & 4 * x^3 & 0 & 0 \\ 0 & y^2 & 2 * x * y & 0 \\ 0 & y * z^3 & x * z^3 & 3 * x * y * z^2 \end{pmatrix}$$

Related functions:

hessian, df³³.

³³standard reduce call for differentiation - see REDUCE User's Manual[2].

NOTE: The function `mat_jacobian` used to be called just "jacobian" however us of that name was in conflict with another Reduce package.

jordan_block

```
jordan_block(expr, square_size);
```

`expr` :- an algebraic expression or symbol.
`square_size` :- a positive integer.

Synopsis:

`jordan_block` computes the square jordan block matrix \mathcal{J} of dimension `square_size`.

The entries of \mathcal{J} are: $\mathcal{J}(i,i) = \text{expr}$ for $i=1 \dots n$, $\mathcal{J}(i,i+1) = 1$ for $i=1 \dots n-1$, and all other entries are 0.

Examples:

$$\text{jordan_block}(x, 5) = \begin{pmatrix} x & 1 & 0 & 0 & 0 \\ 0 & x & 1 & 0 & 0 \\ 0 & 0 & x & 1 & 0 \\ 0 & 0 & 0 & x & 1 \\ 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Related functions:

`diagonal`, `companion`.

lu_decom

```
lu_decom(A);
```

\mathcal{A} :- a matrix containing either numeric entries or imaginary entries with numeric coefficients.

Synopsis:

`lu_decom` performs LU decomposition on \mathcal{A} , ie: it returns $\{\mathcal{L}, \mathcal{U}\}$ where \mathcal{L} is a lower diagonal matrix, \mathcal{U} an upper diagonal matrix and $\mathcal{A} = \mathcal{L}\mathcal{U}$.

caution:

The algorithm used can swap the rows of \mathcal{A} during the calculation. This means that $\mathcal{L}\mathcal{U}$ does not equal \mathcal{A} but a row equivalent of it. Due to this, `lu_decom` returns $\{\mathcal{L}, \mathcal{U}, \text{vec}\}$. The call `convert(A, vec)` will return the matrix that has been decomposed, ie: $\mathcal{L}\mathcal{U} = \text{convert}(\mathcal{A}, \text{vec})$.

Examples:

$$\mathcal{K} = \begin{pmatrix} 1 & 3 & 5 \\ -4 & 3 & 7 \\ 8 & 6 & 4 \end{pmatrix}$$

$$\text{lu} := \text{lu_decom}(\mathcal{K}) = \left\{ \begin{pmatrix} 8 & 0 & 0 \\ -4 & 6 & 0 \\ 1 & 2.25 & 1.1251 \end{pmatrix}, \begin{pmatrix} 1 & 0.75 & 0.5 \\ 0 & 1 & 1.5 \\ 0 & 0 & 1 \end{pmatrix}, [3 \ 2 \ 3] \right\}$$

$$\text{first lu} * \text{second lu} = \begin{pmatrix} 8 & 6 & 4 \\ -4 & 3 & 7 \\ 1 & 3 & 5 \end{pmatrix}$$

$$\text{convert}(\mathcal{K}, \text{third lu}) = \begin{pmatrix} 8 & 6 & 4 \\ -4 & 3 & 7 \\ 1 & 3 & 5 \end{pmatrix}$$

$$\mathcal{P} = \begin{pmatrix} i+1 & i+2 & i+3 \\ 4 & 5 & 2 \\ 1 & i & 0 \end{pmatrix}$$

$$\text{lu} := \text{lu_decom}(\mathcal{P}) = \left\{ \begin{pmatrix} 1 & 0 & 0 \\ 4 & -4*i+5 & 0 \\ i+1 & 3 & 0.41463*i+2.26829 \end{pmatrix}, \begin{pmatrix} 1 & i & 0 \\ 0 & 1 & 0.19512*i+0.24390 \\ 0 & 0 & 1 \end{pmatrix}, [3 \ 2 \ 3] \right\}$$

$$\text{first lu} * \text{second lu} = \begin{pmatrix} 1 & i & 0 \\ 4 & 5 & 2 \\ i+1 & i+2 & i+3 \end{pmatrix}$$

$$\text{convert}(\mathcal{P}, \text{third lu}) = \begin{pmatrix} 1 & i & 0 \\ 4 & 5 & 2 \\ i+1 & i+2 & i+3 \end{pmatrix}$$

Related functions:

`cholesky.`

make_identity

```
make_identity(square_size);
```

`square_size` :- a positive integer.

Synopsis:

`make_identity` creates the identity matrix of dimension `square_size`.

Examples:

$$\text{make_identity}(4) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Related functions:

`diagonal.`

matrix_augment, matrix_stack

```
matrix_augment({mat1, mat2, ..., matn});34
```

`mat1, mat2, ..., matn` :- matrices.

Synopsis:

`matrix_augment` sticks the matrices in `matrix_list` together horizontally.

`matrix_stack` sticks the matrices in `matrix_list` together vertically.

Examples:

$$\text{matrix_augment}(\{\mathcal{A}, \mathcal{A}\}) = \begin{pmatrix} 1 & 2 & 3 & 1 & 2 & 3 \\ 4 & 4 & 6 & 4 & 5 & 6 \\ 7 & 8 & 9 & 7 & 8 & 9 \end{pmatrix}$$

³⁴If you're feeling lazy then the `{}`'s can be omitted.

$$\text{matrix_stack}(\{\mathcal{A}, \mathcal{A}\}) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Related functions:

`augment_columns`, `stack_rows`, `sub_matrix`.

matrixp

```
matrixp(test_input);
test_input  :- anything you like.
```

Synopsis:

`matrixp` is a boolean function that returns `t` if the input is a matrix and `nil` otherwise.

Examples:

```
matrixp(A) = t
matrixp(doodlesackbanana) = nil
```

Related functions:

`squarep`, `symmetricp`.

matrix_stack

see: `matrix_augment`.

minor

```
minor(A, r, c);
A    :- a matrix.
r,c  :- positive integers.
```

Synopsis:

`minor` computes the (r,c)'th minor of \mathcal{A} .

This is created by removing the r'th row and the c'th column from \mathcal{A} .

Examples:

$$\text{minor}(\mathcal{A}, 1, 3) = \begin{pmatrix} 4 & 5 \\ 7 & 8 \end{pmatrix}$$

Related functions:

`remove_columns`, `remove_rows`.

mult_columns, mult_rows

`mult_columns`(\mathcal{A} , `column_list`, `expr`);

\mathcal{A} :- a matrix.

`column_list` :- a positive integer or a list of positive integers.

`expr` :- an algebraic expression.

Synopsis:

`mult_columns` returns a copy of \mathcal{A} in which the columns specified in `column_list` have been multiplied by `expr`.

`mult_rows` performs the same task on the rows of \mathcal{A} .

Examples:

$$\text{mult_columns}(\mathcal{A}, \{1, 3\}, x) = \begin{pmatrix} x & 2 & 3 * x \\ 4 * x & 5 & 6 * x \\ 7 * x & 8 & 9 * x \end{pmatrix}$$

$$\text{mult_rows}(\mathcal{A}, 2, 10) = \begin{pmatrix} 1 & 2 & 3 \\ 40 & 50 & 60 \\ 7 & 8 & 9 \end{pmatrix}$$

Related functions:

`add_to_columns`, `add_to_rows`.

`mult_rows`

see: `mult_columns`.

pivot

`pivot`(\mathcal{A} , `r`, `c`);

\mathcal{A} :- a matrix.
 r, c :- positive integers such that $\mathcal{A}(r, c) \neq 0$.

Synopsis:

`pivot` pivots \mathcal{A} about its (r, c) 'th entry.

To do this, multiples of the r 'th row are added to every other row in the matrix.

This means that the c 'th column will be 0 except for the (r, c) 'th entry.

Examples:

$$\text{pivot}(\mathcal{A}, 2, 3) = \begin{pmatrix} -1 & -0.5 & 0 \\ 4 & 5 & 6 \\ 1 & 0.5 & 0 \end{pmatrix}$$

Related functions:

`rows_pivot`.

pseudo_inverse

`pseudo_inverse`(\mathcal{A});

\mathcal{A} :- a matrix.

Synopsis:

`pseudo_inverse`, also known as the Moore-Penrose inverse, computes the pseudo inverse of \mathcal{A} .

Given the singular value decomposition of \mathcal{A} , i.e: $\mathcal{A} = \mathcal{U} \Sigma \mathcal{V}^T$, then the pseudo inverse \mathcal{A}^{-1} is defined by $\mathcal{A}^{-1} = \mathcal{V}^T \Sigma^{-1} \mathcal{U}$.

Thus $\mathcal{A} * \text{pseudo_inverse}(\mathcal{A}) = \mathcal{I}$.

Examples:

$$\text{pseudo_inverse}(\mathcal{A}) = \begin{pmatrix} -0.2 & 0.1 \\ -0.05 & 0.05 \\ 0.1 & 0 \\ 0.25 & -0.05 \end{pmatrix}$$

Related functions:

`svd`.

random_matrix

```
random_matrix(r,c,limit);
```

`r,c,limit` :- positive integers.

Synopsis:

`random_matrix` creates an `r` by `c` matrix with random entries in the range $-\text{limit} < \text{entry} < \text{limit}$.

switches:

`imaginary` :- if on then matrix entries are $x+iy$ where $-\text{limit} < x,y < \text{limit}$.

`not_negative` :- if on then $0 < \text{entry} < \text{limit}$. In the imaginary case we have $0 < x,y < \text{limit}$.

`only_integer` :- if on then each entry is an integer. In the imaginary case `x` and `y` are integers.

`symmetric` :- if on then the matrix is symmetric.

`upper_matrix` :- if on then the matrix is upper triangular.

`lower_matrix` :- if on then the matrix is lower triangular.

Examples:

$$\text{random_matrix}(3,3,10) = \begin{pmatrix} -4.729721 & 6.987047 & 7.521383 \\ -5.224177 & 5.797709 & -4.321952 \\ -9.418455 & -9.94318 & -0.730980 \end{pmatrix}$$

```
on only_integer, not_negative, upper_matrix, imaginary;
```

$$\text{random_matrix}(4,4,10) = \begin{pmatrix} 2*i+5 & 3*i+7 & 7*i+3 & 6 \\ 0 & 2*i+5 & 5*i+1 & 2*i+1 \\ 0 & 0 & 8 & i \\ 0 & 0 & 0 & 5*i+9 \end{pmatrix}$$

remove_columns, remove_rows

```
remove_columns(A, column_list);
```

`A` :- a matrix.

`column_list` :- either a positive integer or a list of positive integers.

Synopsis:

`remove_columns` removes the columns specified in `column_list` from \mathcal{A} .

`remove_rows` performs the same task on the rows of \mathcal{A} .

Examples:

$$\text{remove_columns}(\mathcal{A}, 2) = \begin{pmatrix} 1 & 3 \\ 4 & 6 \\ 7 & 9 \end{pmatrix}$$

$$\text{remove_rows}(\mathcal{A}, \{1, 3\}) = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix}$$

Related functions:

`minor`.

remove_rows

see: `remove_columns`.

row_dim

see: `column_dim`.

rows_pivot

`rows_pivot(\mathcal{A}, r, c, \{row_list\});`

\mathcal{A} :- a matrix.

r, c :- positive integers such that $\mathcal{A}(r, c) \neq 0$.

`row_list` :- positive integer or a list of positive integers.

Synopsis:

`rows_pivot` performs the same task as `pivot` but applies the pivot only to the rows specified in `row_list`.

Examples:

$$\mathcal{N} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

$$\text{rows_pivot}(\mathcal{N}, 2, 3, \{4, 5\}) = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ -0.75 & 0 & 0.75 \\ -0.375 & 0 & 0.375 \end{pmatrix}$$

Related functions:

`pivot.`

simplex

`simplex(max/min, objective function, {linear inequalities}, [{bounds}]);`

`max/min` :- either max or min (signifying maximise and minimise).
`objective function` :- the function you are maximising or minimising.
`linear inequalities` :- the constraint inequalities. Each one must be of the form *sum of variables* (\leq , $=$, \geq) *number*.
`bounds` :- bounds on the variables as specified for the LP file format. Each bound is of one of the forms $l \leq v$, $v \leq u$, or $l \leq v \leq u$, where v is a variable and l, u are numbers or `infinity` or `-infinity`

Synopsis:

`simplex` applies the revised simplex algorithm to find the optimal (either maximum or minimum) value of the objective function under the linear inequality constraints.

It returns {optimal value, { values of variables at this optimal}}.

The {`bounds`} argument is optional and admissible only when the switch `fastsimplex` is on, which is the default.

Without a {`bounds`} argument, the algorithm implies that all the variables are non-negative.

Examples:

```
simplex(max, x + y, {x >= 10, y >= 20, x + y <= 25});
```

```
***** Error in simplex: Problem has no feasible solution.
```

```
simplex(max, 10x + 5y + 5.5z, {5x + 3z <= 200, x + 0.1y + 0.5z <= 12,
    0.1x + 0.2y + 0.3z <= 9, 30x + 10y + 50z <= 1500});
```

```
{525.0, {x = 40.0, y = 25.0, z = 0}}
```

squarep

```
squarep( $\mathcal{A}$ ) ;
```

\mathcal{A} :- a matrix.

Synopsis:

`squarep` is a boolean function that returns `t` if the matrix is square and `nil` otherwise.

Examples:

$$\mathcal{L} = \begin{pmatrix} 1 & 3 & 5 \end{pmatrix}$$

```
squarep( $\mathcal{A}$ ) = t
```

```
squarep( $\mathcal{L}$ ) = nil
```

Related functions:

`matrixp`, `symmetricp`.

stack_rows

see: `augment_columns`.

sub_matrix

```
sub_matrix( $\mathcal{A}$ , row_list, column_list) ;
```

\mathcal{A} :- a matrix.

row_list, column_list :- either a positive integer or a list of positive integers.

Synopsis:

`sub_matrix` produces the matrix consisting of the intersection of the rows specified in `row_list` and the columns specified in `column_list`.

Examples:

$$\text{sub_matrix}(\mathcal{A}, \{1, 3\}, \{2, 3\}) = \begin{pmatrix} 2 & 3 \\ 8 & 9 \end{pmatrix}$$
Related functions:

`augment_columns`, `stack_rows`.

svd (singular value decomposition)

`svd(A)` ;

\mathcal{A} :- a matrix containing only numeric entries.

Synopsis:

`svd` computes the singular value decomposition of \mathcal{A} .

It returns $\{\mathcal{U}, \Sigma, \mathcal{V}\}$ where $\mathcal{A} = \mathcal{U} \Sigma \mathcal{V}^T$ and $\Sigma = \text{diag}(\sigma_1, \dots, \sigma_n)$. σ_i for $i = (1 \dots n)$ are the singular values of \mathcal{A} .

n is the column dimension of \mathcal{A} .

The singular values of \mathcal{A} are the non-negative square roots of the eigenvalues of $\mathcal{A}^T \mathcal{A}$.

\mathcal{U} and \mathcal{V} are such that $\mathcal{U}\mathcal{U}^T = \mathcal{V}\mathcal{V}^T = \mathcal{V}^T \mathcal{V} = \mathcal{I}_n$.

Examples:

$$\mathcal{Q} = \begin{pmatrix} 1 & 3 \\ -4 & 3 \end{pmatrix}$$

$$\text{svd}(\mathcal{Q}) = \left\{ \begin{pmatrix} 0.289784 & 0.957092 \\ -0.957092 & 0.289784 \end{pmatrix}, \begin{pmatrix} 5.149162 & 0 \\ 0 & 2.913094 \end{pmatrix}, \begin{pmatrix} -0.687215 & 0.726453 \\ -0.726453 & -0.687215 \end{pmatrix} \right\}$$

swap_columns, swap_rows

`swap_columns(A, c1, c2)` ;

\mathcal{A} :- a matrix.

$c1, c2$:- positive integers.

Synopsis:

`swap_columns` swaps column $c1$ of \mathcal{A} with column $c2$.

`swap_rows` performs the same task on 2 rows of \mathcal{A} .

Examples:

$$\text{swap_columns}(\mathcal{A}, 2, 3) = \begin{pmatrix} 1 & 3 & 2 \\ 4 & 6 & 5 \\ 7 & 9 & 8 \end{pmatrix}$$

Related functions:

`swap_entries.`

swap_entries

`swap_entries(\mathcal{A} , {r1, c1}, {r2, c2}) ;`

\mathcal{A} :- a matrix.

r1,c1,r2,c2 :- positive integers.

Synopsis:

`swap_entries` swaps $\mathcal{A}(r1,c1)$ with $\mathcal{A}(r2,c2)$.

Examples:

$$\text{swap_entries}(\mathcal{A}, \{1, 1\}, \{3, 3\}) = \begin{pmatrix} 9 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 1 \end{pmatrix}$$

Related functions:

`swap_columns, swap_rows.`

swap_rows

see: `swap_columns.`

symmetricp

`symmetricp(\mathcal{A}) ;`

\mathcal{A} :- a matrix.

Synopsis:

`symmetricp` is a boolean function that returns `t` if the matrix is symmetric and `nil` otherwise.

Examples:

$$\mathcal{M} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

`symmetricp(\mathcal{A}) = nil`

`symmetricp(\mathcal{M}) = t`

Related functions:

`matrixp, squarep.`

toeplitz

`toeplitz({ $\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n$ });` ³⁵

$\text{expr}_1, \text{expr}_2, \dots, \text{expr}_n$:- algebraic expressions.

Synopsis:

`toeplitz` creates the toeplitz matrix from the expression list.

This is a square symmetric matrix in which the first expression is placed on the diagonal and the i 'th expression is placed on the $(i-1)$ 'th sub and super diagonals.

It has dimension n where n is the number of expressions.

Examples:

$$\text{toeplitz}(\{w, x, y, z\}) = \begin{pmatrix} w & x & y & z \\ x & w & x & y \\ y & x & w & x \\ z & y & x & w \end{pmatrix}$$

triang_adjoint

`triang_adjoint(\mathcal{A});`

\mathcal{A} :- a matrix.

Synopsis:

`triang_adjoint` computes the triangularizing adjoint \mathcal{F} of matrix \mathcal{A} due to the algorithm of Arne Storjohann. \mathcal{F} is lower triangular matrix and the resulting matrix \mathcal{T} of $\mathcal{F} * \mathcal{A} = \mathcal{T}$ is upper triangular with the property that the i -th entry in the diagonal of \mathcal{T} is the determinant of the principal i -th submatrix of the matrix \mathcal{A} .

Examples:

$$\text{triang_adjoint}(\mathcal{A}) = \begin{pmatrix} 1 & 0 & 0 \\ -4 & 1 & 0 \\ -3 & 6 & -3 \end{pmatrix}$$

³⁵If you're feeling lazy then the $\{\}$'s can be omitted.

$$\mathcal{F} * \mathcal{A} = \begin{pmatrix} 1 & 2 & 3 \\ 0 & -3 & -6 \\ 0 & 0 & 0 \end{pmatrix}$$

Vandermonde

`vandermonde({expr1,expr2,...,exprn});` ³⁵

`expr1,expr2,...,exprn` :- algebraic expressions.

Synopsis:

`Vandermonde` creates the Vandermonde matrix from the expression list.

This is the square matrix in which the (i,j)'th entry is `expr_list(i)` ^(j-1).

It has dimension n where n is the number of expressions.

Examples:

$$\text{vandermonde}(\{x, 2 * y, 3 * z\}) = \begin{pmatrix} 1 & x & x^2 \\ 1 & 2 * y & 4 * y^2 \\ 1 & 3 * z & 9 * z^2 \end{pmatrix}$$

kronecker_product

`kronecker_product(Mat1, Mat2)`

`Mat1, Mat2` :- Matrices

Synopsis:

`kronecker_product` creates a matrix containing the Kronecker product (also called direct product or tensor product) of its arguments.

Examples:

```
a1 := mat((1,2),(3,4),(5,6))$
a2 := mat((1,1,1),(2,z,2),(3,3,3))$
kronecker_product(a1,a2);
```


$$\begin{pmatrix} 1 & 1 & 1 & 2 & 2 & 2 \\ 2 & z & 2 & 4 & 2 * z & 4 \\ 3 & 3 & 3 & 6 & 6 & 6 \\ 3 & 3 & 3 & 4 & 4 & 4 \\ 6 & 3 * z & 6 & 8 & 4 * z & 8 \\ 9 & 9 & 9 & 12 & 12 & 12 \\ 5 & 5 & 5 & 6 & 6 & 6 \\ 10 & 5 * z & 10 & 12 & 6 * z & 12 \\ 15 & 15 & 15 & 18 & 18 & 18 \end{pmatrix}$$

16.34.4 Fast Linear Algebra

By turning the `fast_la` switch on, the speed of the following functions will be increased:

<code>add_columns</code>	<code>add_rows</code>	<code>augment_columns</code>	<code>column_dim</code>
<code>copy_into</code>	<code>make_identity</code>	<code>matrix_augment</code>	<code>matrix_stack</code>
<code>minor</code>	<code>mult_column</code>	<code>mult_row</code>	<code>pivot</code>
<code>remove_columns</code>	<code>remove_rows</code>	<code>rows_pivot</code>	<code>squarep</code>
<code>stack_rows</code>	<code>sub_matrix</code>	<code>swap_columns</code>	<code>swap_entries</code>
<code>swap_rows</code>	<code>symmetricp</code>		

The increase in speed will be insignificant unless you are making a significant number (i.e: thousands) of calls. When using this switch, error checking is minimised. This means that illegal input may give strange error messages. Beware.

16.34.5 Acknowledgments

Many of the ideas for this package came from the Maple[3] Linalg package [4].

The algorithms for `cholesky`, `lu_decom`, and `svd` are taken from the book Linear Algebra - J.H. Wilkinson & C. Reinsch[5].

The `gram_schmidt` code comes from Karin Gatermann's Symmetry package[6] for REDUCE.

Bibliography

- [1] Matt Rebbeck: NORMFORM: A REDUCE package for the computation of various matrix normal forms. ZIB, Berlin. (1993)
- [2] Anthony C. Hearn: REDUCE User's Manual 3.6. RAND (1995)
- [3] Bruce W. Char. . . [et al.]: Maple (Computer Program). Springer-Verlag (1991)
- [4] Linalg - a linear algebra package for Maple[3].
- [5] J. H. Wilkinson & C. Reinsch: Linear Algebra (volume II). Springer-Verlag (1971)
- [6] Karin Gatermann: Symmetry: A REDUCE package for the computation of linear representations of groups. ZIB, Berlin. (1992)

16.35 LPDO: Linear Partial Differential Operators

Author: Thomas Sturm

16.35.1 Introduction

Consider the field $F = \mathbb{Q}(x_1, \dots, x_n)$ of rational functions and a set $\Delta = \{\partial_{x_1}, \dots, \partial_{x_n}\}$ of *commuting derivations* acting on F . That is, for all $\partial_{x_i}, \partial_{x_j} \in \Delta$ and all $f, g \in F$ the following properties are satisfied:

$$\begin{aligned}\partial_{x_i}(f + g) &= \partial_{x_i}(f) + \partial_{x_i}(g), \\ \partial_{x_i}(f \cdot g) &= f \cdot \partial_{x_i}(g) + \partial_{x_i}(f) \cdot g,\end{aligned}\tag{16.61}$$

$$\partial_{x_i}(\partial_{x_j}(f)) = \partial_{x_j}(\partial_{x_i}(f)).\tag{16.62}$$

Consider now the set $F[\partial_{x_1}, \dots, \partial_{x_n}]$, where the derivations are used as variables. This set forms a non-commutative *linear partial differential operator ring* with pointwise addition, and multiplication defined as follows: For $f \in F$ and $\partial_{x_i}, \partial_{x_j} \in \Delta$ we have for any $g \in F$ that

$$\begin{aligned}(f\partial_{x_i})(g) &= f \cdot \partial_{x_i}(g), \\ (\partial_{x_i}f)(g) &= \partial_{x_i}(f \cdot g),\end{aligned}\tag{16.63}$$

$$(\partial_{x_i}\partial_{x_j})(g) = \partial_{x_i}(\partial_{x_j}(g)).\tag{16.64}$$

Here “ \cdot ” denotes the multiplication in F . From (16.64) and (16.62) it follows that $\partial_{x_i}\partial_{x_j} = \partial_{x_j}\partial_{x_i}$, and using (16.63) and (16.61) the following *commutator* can be proved:

$$\partial_{x_i}f = f\partial_{x_i} + \partial_{x_i}(f).$$

A *linear partial differential operator* (LPDO) of order k is an element

$$D = \sum_{|j| \leq k} a_j \partial^j \in F[\partial_{x_1}, \dots, \partial_{x_n}]$$

in canonical form. Here the expression $|j| \leq k$ specifies the set of all tuples of the form $j = (j_1, \dots, j_n) \in \mathbb{N}^n$ with $\sum_{i=1}^n j_i \leq k$, and we define $\partial^j = \partial_{x_1}^{j_1} \cdots \partial_{x_n}^{j_n}$.

A *factorization* of D is a non-trivial decomposition

$$D = D_1 \cdots D_r \in F[\partial_{x_1}, \dots, \partial_{x_n}]$$

into multiplicative factors, each of which is an LPDO D_i of order greater than 0 and less than k . If such a factorization exists, then D is called *reducible* or *factorable*, else *irreducible*.

For the purpose of factorization it is helpful to temporarily consider as regular commutative polynomials certain summands of the LPDO under consideration. Consider a commutative polynomial ring over F in new indeterminates y_1, \dots, y_n . Adopting the notational conventions above, for $m \leq k$ the *symbol of D of order m* is defined as

$$\text{Sym}_m(D) = \sum_{|j|=m} a_j y^j \in F[y_1, \dots, y_n].$$

For $m = k$ we obtain as a special case the *symbol* $\text{Sym}(D)$ of D .

16.35.2 Operators

partial

There is a unary operator `partial(·)` denoting ∂ .

$$\langle \text{partial-term} \rangle \rightarrow \mathbf{partial} \ (\langle id \rangle)$$

There is a binary operator `***` for the non-commutative multiplication involving partials ∂_x . All expressions involving `***` are implicitly transformed into LPDOs, i.e., into the following normal form:

$$\begin{aligned} \langle \text{normalized-lpdo} \rangle &\rightarrow \langle \text{normalized-mon} \rangle [+ \langle \text{normalized-lpdo} \rangle] \\ \langle \text{normalized-mon} \rangle &\rightarrow \langle F\text{-element} \rangle [*** \langle \text{partial-termprod} \rangle] \\ \langle \text{partial-termprod} \rangle &\rightarrow \langle \text{partial-term} \rangle [*** \langle \text{partial-termprod} \rangle] \end{aligned}$$

The summands of the *normalized-lpdo* are ordered in some canonical way. As an example consider

```
input: a()***partial(y)***b()***partial(x);
(a()*b()) *** partial(x) *** partial(y) + (a()*diff(b(),y,1)) *** partial(x)
```

Here the *F-elements* are polynomials, where the unknowns are of the type *constant-operator* denoting functions from F :

$$\langle \text{constant-operator} \rangle \rightarrow \langle id \rangle \ (\)$$

We do not admit division of such constant operators since we cannot exclude that such a constant operator denotes 0.

The operator notation on the one hand emphasizes the fact that the denoted elements are functions. On the other hand it distinguishes $a()$ from the variable a of a rational function, which specifically denotes the corresponding projection. Consider e.g.

```
input: (x+y)***partial(y)***(x-y)***partial(x);

      2      2
(x  - y ) *** partial(x) *** partial(y) + ( - x - y) *** partial(x)
```

Here we use as *F-elements* specific elements from $F = \mathbb{Q}(x, y)$.

diff

In our example with constant operators, the transformation into normal form introduces a formal derivative operation `diff(·, ·, ·)`, which cannot be evaluated. Notice that we do not use the Reduce operator `df(·, ·, ·)` here, which for technical reasons cannot smoothly handle our constant operators.

In our second example with rational functions as *F-elements*, derivative occurring with commutation can be computed such that `diff` does not occur in the output.

16.35.3 Shapes of F-elements

Besides the generic computations with constant operators, we provide a mechanism to globally fix a certain *shape* for *F-elements* and to expand constant operators according to that shape.

lpdoset

We give an example for a shape that fixes all constant operators to denote generic bivariate affine linear functions:

```
input: d := (a()+b())***partial(x1)***partial(x2)**2;

      2
d := (a() + b()) *** partial(x1) *** partial(x2)

input: lpdoset {!#10*x1+!#01*x2+!#00,x1,x2};

{-1}

input: d;
```

```
(a00 + a01*x2 + a10*x1 + b00 + b01*x2 + b10*x1) *** partial(x1) *** partial(x2)
```

Notice that the placeholder # must be escaped with !, which is a general convention for Rlisp/Reduce. Notice that `lpdoset` returns the old shape and that `{-1}` denotes the default state that there is no shape selected.

lpdowey1

The command `lpdowey1 {n, x1, x2, ...}` creates a shape for generic polynomials of total degree `n` in variables `x1, x2, ...`.

```
input: lpdowey1(2, x1, x2);
```

```
{#_00_ + #_01_*x2 + #_02_*x22 + #_10_*x1 + #_11_*x1*x2 + #_20_*x12, x1, x2}
```

```
input: lpdoset ws;
```

```
{#10*x1 + #01*x2 + #00, x1, x2}
```

```
input: d;
```

```
(a_00_ + a_01_*x2 + a_02_*x22 + a_10_*x1 + a_11_*x1*x2 + a_20_*x12 + b_00_
+ b_01_*x2 + b_02_*x22 + b_10_*x1 + b_11_*x1*x2 + b_20_*x12) *** partial(x1)
*** partial(x2)2
```

16.35.4 Commands

General

lpdoord The *order* of an `lpdo`:

```
input: lpdoord((a()+b())***partial(x1)***partial(x2)**2+3***partial(x1));
3
```

lpdopt1 Returns the list of derivations (partials) occurring in its argument LPDO *d*.

```
input: lpdopt1(a()***partial(x1)***partial(x2)+partial(x4)+diff(a(), x3, 1));
{partial(x1), partial(x2), partial(x4)}
```

That is the smallest set $\{\dots, \partial_{x_i}, \dots\}$ such that d is defined in $F[\dots, \partial_{x_i}, \dots]$. Notice that formal derivatives are not derivations in that sense.

lpdogp Given a starting symbol a , a list of variables l , and a degree n , `lpdogp(a, l, n)` generates a generic (commutative) polynomial of degree n in variables l with coefficients generated from the starting symbol a :

```
input: lpdogp(a, {x1, x2}, 2);
```

$$a_{00_} + a_{01_}x_2 + a_{02_}x_2^2 + a_{10_}x_1 + a_{11_}x_1x_2 + a_{20_}x_1^2$$

lpdogdp Given a starting symbol a , a list of variables l , and a degree n , `lpdogdp(a, l, n)` generates a generic differential polynomial of degree n in variables l with coefficients generated from the starting symbol a :

```
input: lpdogdp(a, {x1, x2}, 2);
```

$$a_{20_} \text{ *** } \partial(x_1)^2 + a_{02_} \text{ *** } \partial(x_2)^2$$

$$+ a_{11_} \text{ *** } \partial(x_1) \text{ *** } \partial(x_2) + a_{10_} \text{ *** } \partial(x_1)$$

$$+ a_{01_} \text{ *** } \partial(x_2) + a_{00_}$$

Symbols

lpdosym The *symbol* of an lpdo. That is the differential monomial of highest order with the partials replaced by corresponding commutative variables:

```
input: lpdosym((a()+b())***partial(x1)***partial(x2)**2+3***partial(x1));
```

$$y_{x1_}y_{x2_}^2*(a() + b())$$

More generally, one can use a second optional arguments to specify a the order of a different differential monomial to form the symbol of:

```
input: lpdosym((a()+b())***partial(x1)***partial(x2)**2+3***partial(x1), 1);
```

$$3*y_{x1_}$$

Finally, a third optional argument can be used to specify an alternative starting symbol for the commutative variable, which is y by default. Altogether, the optional arguments default like `lpdosym(\cdot)=lpdosym(\cdot , lpdoord(\cdot), y)`.

lpdosym2dp This converts a symbol obtained via `lpdosym` back into an LPDO resulting in the corresponding differential monomial of the original LPDO.

```
input: d := a()***partial(x1)***partial(x2)+partial(x3)$
input: s := lpdosym d;
s := a()*y_x1_*y_x2_
input: lpdosym2dp s;
a() *** partial(x1) *** partial(x2)
```

In analogy to `lpdosym` there is an optional argument for specifying an alternative starting symbol for the commutative variable, which is `y` by default.

lpdos Given LPDOs p, q and $m \in \mathbb{N}$ the function `lpdos(p, q, m)` computes the commutative polynomial

$$S_m = \sum_{\substack{|j|=m \\ |j|<k}} \left(\sum_{i=1}^n p_i \partial_i(q_j) + p_0 q_j \right) y^j.$$

This is useful for the factorization of LPDOs [?].

```
input: p := a()***partial(x1)+b()$
input: q := c()***partial(x1)+d()***partial(x2)$
input: lpdos(p,q,1);
a()*diff(c(),x1,1)*y_x1_ + a()*diff(d(),x1,1)*y_x2_ + b()*c()*y_x1_
+ b()*d()*y_x2_
```

Factorization

lpdofactorize Factorize the argument LPDO d . The ground field F must be fixed via `lpdoset`. The result is a list of lists $\{\dots, (A_i, L_i), \dots\}$. A_i is generally the identifiers `true`, which indicates reducibility. The respective L_i is a list of two differential polynomial factors, the first of which has order 1.

```
input: bk := (partial(x)+partial(y)+(a10-a01)/2) ***
            (partial(x)-partial(y)+(a10+a01)/2);
bk := partial(x)2 - partial(y)2 + a10 *** partial(x) + a01 *** partial(y)
```


$$- a_{01}^2 + a_{10}^2 + \frac{\quad}{4}$$

```

input: lpdoset lpdoweyl(1,x,y);
{#_00_ + #_01_*y + #_10_*x,x,y}

input: lpdofactorize bk;

{{true,

      a01 - a10
{ - partial(x) - partial(y) + -----,
      2

      - a01 - a10
- partial(x) + partial(y) + -----}}}
      2

```

If the result is the empty list, then this guarantees that there is no approximate factorization possible. In general it is possible to obtain several sample factorizations. Note, however, that the result does not provide a complete list of possible factorizations with a left factor of order 1 but only at least one such sample factorization in case of reducibility.

Furthermore, the procedure might fail due to polynomial degrees exceeding certain bounds for the extended quantifier elimination by virtual substitution used internally. In this case there is the identifier `failed` returned. This must not be confused with the empty list indicating irreducibility as described above.

Besides

1. the LPDO d ,

`lpdofactorizex` accepts several optional arguments:

2. An LPDO of order 1, which serves as a template for the left (linear) factor. The default is a generic linear LPDO with generic coefficient functions according from the ground field specified via `lpdoset`. The principle idea is to support the factorization by guessing that certain differential monomials are not present.
3. An LPDO of order $\text{ord}(d) - 1$, which serves as a template for the right factor. Similarly to the previous argument the default is fully generic.

lpdofac This is a low-level entry point to the factorization `lpdofactorize`. It accepts the same arguments as `lpdofactorize`. It generates factorization conditions as a quite large first-order formula over the reals. This can be passed to extended quantifier elimination. For example, consider `bk` as in the example for `lpdofactorize` above:

```
input: faccond := lpdofac bk$

input: rlqea faccond;

{{true,

      a01 - a10
  {p_00_00_ = -----,
      2

  p_00_01_ = 0, p_00_10_ = 0, p_01_00_ = -1, p_01_01_ = 0, p_01_10_ = 0,

  p_10_00_ = -1, p_10_01_ = 0, p_10_10_ = 0,

      - a01 - a10
  q_00_00_ = -----,
      2

  q_00_01_ = 0, q_00_10_ = 0, q_01_00_ = 1, q_01_01_ = 0, q_01_10_ = 0,

  q_10_00_ = -1, q_10_01_ = 0, q_10_10_ = 0}}}}
```

The result of the extended quantifier elimination provides coefficient values for generic factor polynomials p and q . These are automatically interpreted and converted into differential polynomials by `lpdofactorize`.

Approximate Factorization

lpdofactorizex Approximately factorize the argument LPDO d . The ground field F must be fixed via `lpdoset`. The result is a list of lists $\{\dots, (A_i, L_i), \dots\}$. Each A_i is quantifier-free formula possibly containing a variable `epsilon`, which describes the precision of corresponding factorization L_i . L_i is a list containing two factors, the first of which is linear.

```
input: off lpdocoeffnorm$

input: lpdoset lpdoweyl(0,x1,x2)$

input: f2 := partial(x1)***partial(x2) + 1$

input: lpdofactorizex f2;
```

```
{{epsilon - 1 >= 0, {partial(x1), partial(x2)}},
 {epsilon - 1 >= 0, {partial(x2), partial(x1)}}}
```

If the result is the empty list, then this guarantees that there is no approximate factorization possible. In our example we happen to obtain two possible factorizations. Note, however, that the result in general does not provide a complete list of factorizations with a left factor of order 1 but only at least one such sample factorization.

Furthermore, the procedure might fail due to polynomial degrees exceeding certain bounds for the extended quantifier elimination by virtual substitution used internally. If this happens, the corresponding A_i will contain existential quantifiers ex , and L_i will be meaningless.

Da sollte besser ein failed kommen ...

The first of the two subresults above has the semantics that $\partial_{x_1}\partial_{x_2}$ is an approximate factorization of f_2 for all $\varepsilon \geq 1$. Formally, $\|f_2 - \partial_{x_1}\partial_{x_2}\| \leq \varepsilon$ for all $\varepsilon \geq 1$, which is equivalent to $\|f_2 - \partial_{x_1}\partial_{x_2}\| \leq 1$. That is, 1 is an upper bound for the approximation error over \mathbb{R}^2 . Where there are two possible choices for the seminorm $\|\cdot\|$:

1. ...
2. ...

explain switch lpdcoeffnorm ...

Besides

1. the LPDO d ,

`lpdofactorizex` accepts several optional arguments:

2. A Boolean combination ψ of equations, negated equations, and (possibly strict) ordering constraints. This ψ describes a (semialgebraic) region over which to factorize approximately. The default is `true` specifying the entire \mathbb{R}^n . It is possible to choose ψ parametrically. Then the parameters will in general occur in the conditions A_i in the result.
- 3., 4. An LPDO of order 1, which serves as a template for the left (linear) factor, and an LPDO of order $\text{ord}(d) - 1$, which serves as a template for the right factor. See the documentation of `lpdofactorize` for defaults and details.
5. A bound ε for describing the desired precision for approximate factorization. The default is the symbol `epsilon`, i.e., a symbolic choice such that

the optimal choice (with respect to parameters in ψ) is obtained during factorization. It is possible to fix $\varepsilon \in \mathbb{Q}$. This does, however, not considerably simplify the factorization process in most cases.

```
input: f3 := partial(x1) *** partial(x2) + x1$
input: psi1 := 0<=x1<=1 and 0<=x2<=1$
input: lpdofactorizex(f3,psi1,a()***partial(x1),b()***partial(x2));
{{epsilon - 1 >= 0,{partial(x1),partial(x2)}}}
```

lpdofacx This is a low-level entry point to the factorization `lpdofactorizex`. It is analogous to `lpdofac` for `lpdofactorize`; see the documentation there for details.

lpdohrect

lpdohcirc

16.36 MODSR: Modular solve and roots

This package supports solve (M_SOLVE) and roots (M_ROOTS) operators for modular polynomials and modular polynomial systems. The moduli need not be primes. M_SOLVE requires a modulus to be set. M_ROOTS takes the modulus as a second argument. For example:

```
on modular; setmod 8;
m_solve(2x=4);           ->  {{X=2},{X=6}}
m_solve({x^2-y^3=3});
  ->  {{X=0,Y=5},{X=2,Y=1},{X=4,Y=5},{X=6,Y=1}}
m_solve({x=2,x^2-y^3=3}); ->  {{X=2,Y=1}}
off modular;
m_roots(x^2-1,8);        ->  {1,3,5,7}
m_roots(x^3-x,7);        ->  {0,1,6}
```

There is no further documentation for this package.

Author: Herbert Melenk.

16.37 NCPOLY: Non-commutative polynomial ideals

This package allows the user to set up automatically a consistent environment for computing in an algebra where the non-commutativity is defined by Lie-bracket commutators. The package uses the REDUCE **noncom** mechanism for elementary polynomial arithmetic; the commutator rules are automatically computed from the Lie brackets.

Authors: Herbert Melenk and Joachim Apel.

16.37.1 Introduction

REDUCE supports a very general mechanism for computing with objects under a non-commutative multiplication, where commutator relations must be introduced explicitly by rule sets when needed. The package **NCPOLY** allows you to set up automatically a consistent environment for computing in an algebra where the non-commutativity is defined by Lie-bracket commutators. The package uses the REDUCE **noncom** mechanism for elementary polynomial arithmetic; the commutator rules are automatically computed from the Lie brackets. You can perform polynomial arithmetic directly, including **division** and **factorization**. Additionally **NCPOLY** supports computations in a one sided ideal (left or right), especially one sided **Gröbner** bases and **polynomial reduction**.

16.37.2 Setup, Cleanup

Before the computations can start the environment for a non-commutative computation must be defined by a call to **nc_setup**:

```
nc_setup(<vars>[, <comms>][, <dir>]);
```

where

$\langle vars \rangle$ is a list of variables; these must include the non-commutative quantities.

$\langle comms \rangle$ is a list of equations $\langle u \rangle * \langle v \rangle - \langle v \rangle * \langle u \rangle = \langle rh \rangle$ where $\langle u \rangle$ and $\langle v \rangle$ are members of $\langle vars \rangle$, and $\langle rh \rangle$ is a polynomial.

$\langle dir \rangle$ is either *left* or *right* selecting a left or a right one sided ideal. The initial direction is *left*.

nc_setup generates from $\langle comms \rangle$ the necessary rules to support an algebra where all monomials are ordered corresponding to the given variable sequence. All pairs of variables which are not explicitly covered in the commutator set are considered as commutative and the corresponding rules are also activated.

The second parameter in **nc_setup** may be omitted if the operator is called for the second time, e.g. with a reordered variable sequence. In such a case the last commutator set is used again.

Remarks:

- The variables need not be declared **noncom** - **nc_setup** performs all necessary declarations.
- The variables need not be formal operator expressions; **nc_setup** encapsulates a variable x internally as $\text{nc}! * (!_x)$ expressions anyway where the operator $\text{nc}!*$ keeps the noncom property.
- The commands **order** and **korder** should be avoided because **nc_setup** sets these such that the computation results are printed in the correct term order.

Example:

```
nc_setup ({KK, NN, k, n},
          {NN*n-n*NN= NN, KK*k-k*KK= KK});

NN*n;           ->   NN*n
n*NN;           ->   NN*n - NN
nc_setup ({k, n, KK, NN});
NN*n - NN       ->   n*NN;
```

Here KK, NN, k, n are non-commutative variables where the commutators are described as $[NN, n] = NN$, $[KK, k] = KK$.

The current term order must be compatible with the commutators: the product $\langle u \rangle * \langle v \rangle$ must precede all terms on the right hand side $\langle rh \rangle$ under the current term order. Consequently

- the maximal degree of $\langle u \rangle$ or $\langle v \rangle$ in $\langle rh \rangle$ is 1,
- in a total degree ordering the total degree of $\langle rh \rangle$ may be not higher than 1,
- in an elimination degree order (e.g. *lex*) all variables in $\langle rh \rangle$ must be below the minimum of $\langle u \rangle$ and $\langle v \rangle$.
- If $\langle rh \rangle$ does not contain any variables or has at most $\langle u \rangle$ or $\langle v \rangle$, any term order can be selected.

If you want to use the non-commutative variables or results from non-commutative computations later in commutative operations it might be necessary to switch off

the non-commutative evaluation mode because not all operators in **REDUCE** are prepared for that environment. In such a case use the command

```
nc_cleanup;
```

without parameters. It removes all internal rules and definitions which **nc_setup** had introduced. To reactive non-commutative call **nc_setup** again.

16.37.3 Left and right ideals

A (polynomial) left ideal L is defined by the axioms

$$u \in L, v \in L \implies u + v \in L$$

$$u \in L \implies k * u \in L \text{ for an arbitrary polynomial } k$$

where “*” is the non-commutative multiplication. Correspondingly, a right ideal R is defined by

$$u \in R, v \in R \implies u + v \in R$$

$$u \in R \implies u * k \in R \text{ for an arbitrary polynomial } k$$

16.37.4 Gröbner bases

When a non-commutative environment has been set up by **nc_setup**, a basis for a left or right polynomial ideal can be transformed into a Gröbner basis by the operator **nc_groebner**:

```
nc_groebner(<plist>);
```

Note that the variable set and variable sequence must be defined before in the **nc_setup** call. The term order for the Gröbner calculation can be set by using the **torder** declaration. The internal steps of the Gröbner calculation can be watched by setting the switches **trgroeb** (=list all internal basis polynomials) or **trgroebbs** (=list additionally the S -polynomials) ³⁶.

For details about **torder**, **trgroeb** and **trgroebbs** see the **REDUCE GROEBNER** manual.

```
2: nc_setup({k,n,NN,KK},{NN*n-n*NN=NN,KK*k-k*KK=KK},left);
```

```
3: p1 := (n-k+1)*NN - (n+1);
```

³⁶The command `lisp(!*trgroebfull:=t);` causes additionally all elementary polynomial operations to be printed.


```

p1 := - k*nn + n*nn - n + nn - 1
4: p2 := (k+1)*KK - (n-k);
p2 := k*kk + k - n + kk
5: nc_groebner ({p1,p2});
{k*nn - n*nn + n - nn + 1,
 k*kk + k - n + kk,
 n*nn*kk - n*kk - n + nn*kk - kk - 1}

```

Important: Do not use the operators of the GROEBNER package directly as they would not consider the non-commutative multiplication.

16.37.5 Left or right polynomial division

The operator **nc_divide** computes the one sided quotient and remainder of two polynomials:

```
nc_divide(<p1>, <p2>);
```

The result is a list with quotient and remainder. The division is performed as a pseudo-division, multiplying $\langle p1 \rangle$ by coefficients if necessary. The result $\{\langle q \rangle, \langle r \rangle\}$ is defined by the relation

$\langle c \rangle * \langle p1 \rangle = \langle q \rangle * \langle p2 \rangle + \langle r \rangle$ for direction *left* and

$\langle c \rangle * \langle p1 \rangle = \langle p2 \rangle * \langle q \rangle + \langle r \rangle$ for direction *right*,

where $\langle c \rangle$ is an expression that does not contain any of the ideal variables, and the leading term of $\langle r \rangle$ is lower than the leading term of $\langle p2 \rangle$ according to the actual term order.

16.37.6 Left or right polynomial reduction

For the computation of the one sided remainder of a polynomial modulo a given set of other polynomials the operator **nc_preduce** may be used:

```
nc_preduce(<polynomial>, <plist>);
```

The result of the reduction is unique (canonical) if and only if $\langle plist \rangle$ is a one sided Gröbner basis. Then the computation is at the same time an ideal membership test: if the result is zero, the polynomial is member of the ideal, otherwise not.

16.37.7 Factorization

Technique

Polynomials in a non-commutative ring cannot be factored using the ordinary **factorize** command of REDUCE. Instead one of the operators of this section must be used:

```
nc_factorize(<polynomial>);
```

The result is a list of factors of $\langle polynomial \rangle$. A list with the input expression is returned if it is irreducible.

As non-commutative factorization is not unique, there is an additional operator which computes all possible factorizations

```
nc_factorize_all(<polynomial>);
```

The result is a list of factor decompositions of $\langle polynomial \rangle$. If there are no factors at all the result list has only one member which is a list containing the input polynomial.

Control of the factorization

In contrast to factoring in commutative polynomial rings, the non-commutative factorization is rather time consuming. Therefore two additional operators allow you to reduce the amount of computing time when you look only for isolated factors in special context, e.g. factors with a limited degree or factors which contain only explicitly specified variables:

```
left_factor(<polynomial>[, <deg>[, <vars>]])
right_factor(<polynomial>[, <deg>[, <vars>]])
left_factors(<polynomial>[, <deg>[, <vars>]])
right_factors(<polynomial>[, <deg>[, <vars>]])
```

where $\langle polynomial \rangle$ is the form under investigation, $\langle vars \rangle$ is an optional list of variables which must appear in the factor, and $\langle deg \rangle$ is an optional integer degree bound for the total degree of the factor, a zero for an unbounded search, or a monomial (product of powers of the variables) where each exponent is

an individual degree bound for its base variable; unmentioned variables are allowed in arbitrary degree. The operators **_factor* stop when they have found one factor, while the operators **_factors* select all one-sided factors within the given range. If there is no factor of the desired type, an empty list is returned by **_factors* while the routines **_factor* return the input polynomial.

Time of the factorization

The share variable *nc_factor_time* sets an upper limit for the time to be spent for a call to the non-commutative factorizer. If the value is a positive integer, a factorization is terminated with an error message as soon as the time limit is reached. The time units are milliseconds.

Usage of SOLVE

The factorizer internally uses *solve*, which is controlled by the REDUCE switch *varopt*. This switch (which per default is set *on*) allows, to reorder the variable sequence, which is favourable for the normal system. It should be avoided to set *varopt off*, when using the non-commutative factorizer, unless very small polynomials are used.

16.37.8 Output of expressions

It is often desirable to have the commutative parts (coefficients) in a non-commutative operation condensed by factorization. The operator

```
nc_compact (<polynomial>)
```

collects the coefficients to the powers of the lowest possible non-commutative variable.

```
load ncpoly;
```

```
nc_setup({n, NN}, {NN*n-n*NN=NN}) $
```

```
p1 := n**4 + n**2*nn + 4*n**2 + 4*n*nn + 4*nn + 4;
```

```

      4      2      2
p1 := n  + n *nn + 4*n  + 4*n*nn + 4*nn + 4
```

```
nc_compact p1;
```

```

      2      2      2
```

$$(n + 2) + (n + 2) * nn$$

16.38 NORMFORM: Computation of matrix normal forms

This package contains routines for computing the following normal forms of matrices:

- `smithex_int`
- `smithex`
- `frobenius`
- `ratjordan`
- `jordansymbolic`
- `jordan`.

Author: Matt Rebbeck.

16.38.1 Introduction

When are two given matrices similar? Similar matrices have the same trace, determinant, characteristic polynomial, and eigenvalues, but the matrices

$$\mathcal{U} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \quad \text{and} \quad \mathcal{V} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

are the same in all four of the above but are not similar. Otherwise there could exist a nonsingular $\mathcal{N} \in M_2$ (the set of all 2×2 matrices) such that $\mathcal{U} = \mathcal{N} \mathcal{V} \mathcal{N}^{-1} = \mathcal{N} \mathbf{0} \mathcal{N}^{-1} = \mathbf{0}$, which is a contradiction since $\mathcal{U} \neq \mathbf{0}$.

Two matrices can look very different but still be similar. One approach to determining whether two given matrices are similar is to compute the normal form of them. If both matrices reduce to the same normal form they must be similar.

NORMFORM is a package for computing the following normal forms of matrices:

- `smithex`
- `smithex_int`
- `frobenius`
- `ratjordan`
- `jordansymbolic`
- `jordan`

The package is loaded by `load_package normform;`

By default all calculations are carried out in \mathbb{Q} (the rational numbers). For `smithex`, `frobenius`, `ratjordan`, `jordansymbolic`, and `jordan`, this field can be extended. Details are given in the respective sections.

The `frobenius`, `ratjordan`, and `jordansymbolic` normal forms can also be computed in a modular base. Again, details are given in the respective sections.

The algorithms for each routine are contained in the source code.

NORMFORM has been converted from the `normform` and `Normform` packages written by T.M.L. Mulders and A.H.M. Levelt. These have been implemented in Maple [4].

16.38.2 smithex

function

`smithex`(\mathcal{A} , x) computes the Smith normal form \mathcal{S} of the matrix \mathcal{A} .

It returns $\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ where \mathcal{S} , \mathcal{P} , and \mathcal{P}^{-1} are such that $\mathcal{P}\mathcal{S}\mathcal{P}^{-1} = \mathcal{A}$.

\mathcal{A} is a rectangular matrix of univariate polynomials in x .

x is the variable name.

field extensions

Calculations are performed in \mathbb{Q} . To extend this field the ARNUM package can be used. For details see *section 8*.

synopsis

- The Smith normal form \mathcal{S} of an n by m matrix \mathcal{A} with univariate polynomial entries in x over a field F is computed. That is, the polynomials are then regarded as elements of the Euclidean domain $F(x)$.
- The Smith normal form is a diagonal matrix \mathcal{S} where:
 - $\text{rank}(\mathcal{A}) = \text{number of nonzero rows (columns) of } \mathcal{S}$.
 - $\mathcal{S}(i, i)$ is a monic polynomial for $0 < i \leq \text{rank}(\mathcal{A})$.
 - $\mathcal{S}(i, i)$ divides $\mathcal{S}(i+1, i+1)$ for $0 < i < \text{rank}(\mathcal{A})$.
 - $\mathcal{S}(i, i)$ is the greatest common divisor of all i by i minors of \mathcal{A} .

Hence, if we have the case that $n = m$, as well as $\text{rank}(\mathcal{A}) = n$, then product $(\mathcal{S}(i, i), i = 1 \dots n) = \det(\mathcal{A}) / \text{lcoeff}(\det(\mathcal{A}), x)$.

- The Smith normal form is obtained by doing elementary row and column operations. This includes interchanging rows (columns), multiplying through a row (column) by -1 , and adding integral multiples of one row (column) to another.
- Although the rank and determinant can be easily obtained from \mathcal{S} , this is not an efficient method for computing these quantities except that this may yield a partial factorization of $\det(\mathcal{A})$ without doing any explicit factorizations.

example

```
load_package normform;
```

$$\mathcal{A} = \begin{pmatrix} x & x+1 \\ 0 & 3 * x^2 \end{pmatrix}$$

$$\text{smithex}(\mathcal{A}, x) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & x^3 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 3 * x^2 & 1 \end{pmatrix}, \begin{pmatrix} x & x+1 \\ -3 & -3 \end{pmatrix} \right\}$$

16.38.3 smithex_int

function

Given an n by m rectangular matrix \mathcal{A} that contains *only* integer entries, `smithex_int(\mathcal{A})` computes the Smith normal form \mathcal{S} of \mathcal{A} .

It returns $\{\mathcal{S}, \mathcal{P}, \mathcal{P}^{-1}\}$ where \mathcal{S}, \mathcal{P} , and \mathcal{P}^{-1} are such that $\mathcal{P}\mathcal{S}\mathcal{P}^{-1} = \mathcal{A}$.

synopsis

- The Smith normal form \mathcal{S} of an n by m matrix \mathcal{A} with integer entries is computed.
- The Smith normal form is a diagonal matrix \mathcal{S} where:
 - $\text{rank}(\mathcal{A}) = \text{number of nonzero rows (columns) of } \mathcal{S}$.
 - $\text{sign}(\mathcal{S}(i, i)) = 1$ for $0 < i \leq \text{rank}(\mathcal{A})$.
 - $\mathcal{S}(i, i)$ divides $\mathcal{S}(i+1, i+1)$ for $0 < i < \text{rank}(\mathcal{A})$.
 - $\mathcal{S}(i, i)$ is the greatest common divisor of all i by i minors of \mathcal{A} .

Hence, if we have the case that $n = m$, as well as $\text{rank}(\mathcal{A}) = n$, then $\text{abs}(\det(\mathcal{A})) = \text{product}(\mathcal{S}(i, i), i = 1 \dots n)$.

- The Smith normal form is obtained by doing elementary row and column operations. This includes interchanging rows (columns), multiplying through a row (column) by -1 , and adding integral multiples of one row (column) to another.

example

```
load_package normform;
```

$$\mathcal{A} = \begin{pmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{pmatrix}$$

```
smithex_int( $\mathcal{A}$ ) =
```

$$\left\{ \begin{pmatrix} 3 & 0 & 0 \\ 0 & 12 & 0 \\ 0 & 0 & 60 \end{pmatrix}, \begin{pmatrix} -17 & -5 & -4 \\ 64 & 19 & 15 \\ -50 & -15 & -12 \end{pmatrix}, \begin{pmatrix} 1 & -24 & 30 \\ -1 & 25 & -30 \\ 0 & -1 & 1 \end{pmatrix} \right\}$$

16.38.4 frobenius

function

`frobenius(\mathcal{A})` computes the Frobenius normal form \mathcal{F} of the matrix \mathcal{A} .

It returns $\{\mathcal{F}, \mathcal{P}, \mathcal{P}^{-1}\}$ where \mathcal{F} , \mathcal{P} , and \mathcal{P}^{-1} are such that $\mathcal{P}\mathcal{F}\mathcal{P}^{-1} = \mathcal{A}$.

\mathcal{A} is a square matrix.

field extensions

Calculations are performed in \mathcal{Q} . To extend this field the ARNUM package can be used. For details see *section 8*.

modular arithmetic

`frobenius` can be calculated in a modular base. For details see *section 9*.

synopsis

- \mathcal{F} has the following structure:

$$\mathcal{F} = \begin{pmatrix} \mathcal{C}p_1 & & & \\ & \mathcal{C}p_2 & & \\ & & \ddots & \\ & & & \mathcal{C}p_k \end{pmatrix}$$

where the $\mathcal{C}(p_i)$'s are companion matrices associated with polynomials p_1, p_2, \dots, p_k , with the property that p_i divides p_{i+1} for $i = 1 \dots k-1$. All unmarked entries are zero.

- The Frobenius normal form defined in this way is unique (ie: if we require that p_i divides p_{i+1} as above).

example

```
load_package normform;
```

$$\mathcal{A} = \begin{pmatrix} \frac{-x^2+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \\ \frac{-x^2-x+y^2+y}{y} & \frac{-x^2+x+y^2-y}{y} \end{pmatrix}$$

```
frobenius( $\mathcal{A}$ ) =
```

$$\left\{ \begin{pmatrix} 0 & \frac{x*(x^2-x-y^2+y)}{y} \\ 1 & \frac{-2*x^2+x+2*y^2}{y} \end{pmatrix}, \begin{pmatrix} 1 & \frac{-x^2+y^2+y}{y} \\ 0 & \frac{-x^2-x+y^2+y}{y} \end{pmatrix}, \begin{pmatrix} 1 & \frac{-x^2+y^2+y}{x^2+x-y^2-y} \\ 0 & \frac{-y}{x^2+x-y^2-y} \end{pmatrix} \right\}$$

16.38.5 ratjordan

function

`ratjordan(\mathcal{A})` computes the rational Jordan normal form \mathcal{R} of the matrix \mathcal{A} .

It returns $\{\mathcal{R}, \mathcal{P}, \mathcal{P}^{-1}\}$ where \mathcal{R}, \mathcal{P} , and \mathcal{P}^{-1} are such that $\mathcal{P}\mathcal{R}\mathcal{P}^{-1} = \mathcal{A}$.

\mathcal{A} is a square matrix.

field extensions

Calculations are performed in \mathcal{Q} . To extend this field the ARNUM package can be used. For details see *section 8*.

modular arithmetic

`ratjordan` can be calculated in a modular base. For details see *section 9*.

synopsis

- \mathcal{R} has the following structure:

$$\mathcal{R} = \begin{pmatrix} r_{11} & & & & \\ & r_{12} & & & \\ & & \ddots & & \\ & & & r_{21} & \\ & & & & r_{22} \\ & & & & & \ddots \end{pmatrix}$$

The r_{ij} 's have the following shape:

$$r_{ij} = \begin{pmatrix} \mathcal{C}(p) & \mathcal{I} & & & \\ & \mathcal{C}(p) & \mathcal{I} & & \\ & & \ddots & \ddots & \\ & & & \mathcal{C}(p) & \mathcal{I} \\ & & & & \mathcal{C}(p) \end{pmatrix}$$

where there are e_{ij} times $\mathcal{C}(p)$ blocks along the diagonal and $\mathcal{C}(p)$ is the companion matrix associated with the irreducible polynomial p . All unmarked entries are zero.

example

```
load_package normform;
```

$$\mathcal{A} = \begin{pmatrix} x+y & 5 \\ y & x^2 \end{pmatrix}$$

```
ratjordan( $\mathcal{A}$ ) =
```

$$\left\{ \begin{pmatrix} 0 & -x^3 - x^2 * y + 5 * y \\ 1 & x^2 + x + y \end{pmatrix}, \begin{pmatrix} 1 & x+y \\ 0 & y \end{pmatrix}, \begin{pmatrix} 1 & \frac{-(x+y)}{y} \frac{1}{y} \\ 0 & \frac{1}{y} \end{pmatrix} \right\}$$

16.38.6 jordanSymbolic

function

`jordanSymbolic(A)` computes the Jordan normal form \mathcal{J} of the matrix \mathcal{A} .

It returns $\{\mathcal{J}, \mathcal{L}, \mathcal{P}, \mathcal{P}^{-1}\}$, where \mathcal{J}, \mathcal{P} , and \mathcal{P}^{-1} are such that $\mathcal{P}\mathcal{J}\mathcal{P}^{-1} = \mathcal{A}$. $\mathcal{L} = \{ll, \xi\}$, where ξ is a name and ll is a list of irreducible factors of $p(\xi)$.

\mathcal{A} is a square matrix.

field extensions

Calculations are performed in \mathcal{Q} . To extend this field the ARNUM package can be used. For details see *section 8*.

modular arithmetic

`jordanSymbolic` can be calculated in a modular base. For details see *subsection 9*.

extras

If using `xr`, the X interface for REDUCE, the appearance of the output can be improved by switching on `looking_good;`. This converts all `lambda` to ξ and improves the indexing, eg: `lambda12` \Rightarrow ξ_{12} . The example (*section 6.6*) shows the output when this switch is on.

synopsis

- A *Jordan block* $j_k(\lambda)$ is a k by k upper triangular matrix of the form:

$$j_k(\lambda) = \begin{pmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda & 1 \\ & & & & \lambda \end{pmatrix}$$

There are $k - 1$ terms “+1” in the superdiagonal; the scalar λ appears k times on the main diagonal. All other matrix entries are zero, and $j_1(\lambda) = (\lambda)$.

- A Jordan matrix $\mathcal{J} \in M_n$ (the set of all n by n matrices) is a direct sum of *Jordan blocks*.

$$\mathcal{J} = \begin{pmatrix} J_{n_1}(\lambda_1) & & \\ & J_{n_2}(\lambda_2) & \\ & & \ddots \\ & & & J_{n_k}(\lambda_k) \end{pmatrix}, n_1 + n_2 + \cdots + n_k = n$$

in which the orders n_i may not be distinct and the values λ_i need not be distinct.

- Here λ is a zero of the characteristic polynomial p of \mathcal{A} . If p does not split completely, symbolic names are chosen for the missing zeroes of p . If, by some means, one knows such missing zeroes, they can be substituted for the symbolic names. For this, `jordansymbolic` actually returns $\{\mathcal{J}, \mathcal{L}, \mathcal{P}, \mathcal{P}^{-1}\}$. \mathcal{J} is the Jordan normal form of \mathcal{A} (using symbolic names if necessary). $\mathcal{L} = \{ll, \xi\}$, where ξ is a name and ll is a list of irreducible factors of $p(\xi)$. If symbolic names are used then ξ_{ij} is a zero of ll_i . \mathcal{P} and \mathcal{P}^{-1} are as above.

example

```
load_package normform;
on looking_good;
```

$$\mathcal{A} = \begin{pmatrix} 1 & y \\ y^2 & 3 \end{pmatrix}$$

```
jordansymbolic(A) =
```

$$\left\{ \begin{pmatrix} \xi_{11} & 0 \\ 0 & \xi_{12} \end{pmatrix}, \{ \{-y^3 + \xi^2 - 4 * \xi + 3\}, \xi \}, \right. \\ \left. \begin{pmatrix} \xi_{11} - 3 & \xi_{12} - 3 \\ y^2 & y^2 \end{pmatrix}, \begin{pmatrix} \frac{\xi_{11}-2}{2*(y^3-1)} & \frac{\xi_{11}+y^3-1}{2*y^2*(y^3+1)} \\ \frac{\xi_{12}-2}{2*(y^3-1)} & \frac{\xi_{12}+y^3-1}{2*y^2*(y^3+1)} \end{pmatrix} \right\}$$

```
solve(-y^3 + xi^2 - 4 * xi + 3, xi);
```

$$\{\xi = \sqrt{y^3 + 1} + 2, \xi = -\sqrt{y^3 + 1} + 2\}$$

```
J = sub({xi(1,1) = sqrt(y^3 + 1) + 2, xi(1,2) = -sqrt(y^3 + 1) + 2},
first jordansymbolic (A));
```

$$\mathcal{J} = \begin{pmatrix} \sqrt{y^3+1}+2 & 0 \\ 0 & -\sqrt{y^3+1}+2 \end{pmatrix}$$

For a similar example of this in standard REDUCE (ie: not using `xr`), see the *normform.log* file.

16.38.7 jordan

function

`jordan(A)` computes the Jordan normal form \mathcal{J} of the matrix \mathcal{A} .

It returns $\{\mathcal{J}, \mathcal{P}, \mathcal{P}^{-1}\}$, where \mathcal{J} , \mathcal{P} , and \mathcal{P}^{-1} are such that $\mathcal{P}\mathcal{J}\mathcal{P}^{-1} = \mathcal{A}$.

\mathcal{A} is a square matrix.

field extensions

Calculations are performed in \mathcal{Q} . To extend this field the ARNUM package can be used. For details see *section 8*.

note

In certain polynomial cases `fullroots` is turned on to compute the zeroes. This can lead to the calculation taking a long time, as well as the output being very large. In this case a message `***** WARNING: fullroots turned on. May take a while. will be printed.` will be printed. It may be better to kill the calculation and compute `jordansymbolic` instead.

synopsis

- The Jordan normal form \mathcal{J} with entries in an algebraic extension of \mathcal{Q} is computed.
- A *Jordan block* $j_k(\lambda)$ is a k by k upper triangular matrix of the form:

$$j_k(\lambda) = \begin{pmatrix} \lambda & 1 & & & \\ & \lambda & 1 & & \\ & & \ddots & \ddots & \\ & & & \lambda & 1 \\ & & & & \lambda \end{pmatrix}$$

There are $k-1$ terms “+1” in the superdiagonal; the scalar λ appears k times on the main diagonal. All other matrix entries are zero, and $j_1(\lambda) = (\lambda)$.

- A Jordan matrix $\mathcal{J} \in M_n$ (the set of all n by n matrices) is a direct sum of *jordan blocks*.

$$\mathcal{J} = \begin{pmatrix} j_{n_1}(\lambda_1) & & & \\ & j_{n_2}(\lambda_2) & & \\ & & \ddots & \\ & & & j_{n_k}(\lambda_k) \end{pmatrix}, n_1 + n_2 + \cdots + n_k = n$$

in which the orders n_i may not be distinct and the values λ_i need not be distinct.

- Here λ is a zero of the characteristic polynomial p of \mathcal{A} . The zeroes of the characteristic polynomial are computed exactly, if possible. Otherwise they are approximated by floating point numbers.

example

```
load_package normform;
```

$$\mathcal{A} = \begin{pmatrix} -9 & -21 & -15 & 4 & 2 & 0 \\ -10 & 21 & -14 & 4 & 2 & 0 \\ -8 & 16 & -11 & 4 & 2 & 0 \\ -6 & 12 & -9 & 3 & 3 & 0 \\ -4 & 8 & -6 & 0 & 5 & 0 \\ -2 & 4 & -3 & 0 & 1 & 3 \end{pmatrix}$$

```
 $\mathcal{J}$  = first jordan( $\mathcal{A}$ );
```

$$\mathcal{J} = \begin{pmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & i+2 & 0 \\ 0 & 0 & 0 & 0 & 0 & -i+2 \end{pmatrix}$$

16.38.8 arnum

The package is loaded by `load_package arnum;`. The algebraic field \mathcal{Q} can now be extended. For example, `defpoly sqrt2**2-2;` will extend it to include $\sqrt{2}$ (defined here by `sqrt2`). The ARNUM package was written by Eberhard Schröder and is described in the *arnum.tex* file.

example

```
load_package normform;
load_package arnum;
defpoly sqrt2**2-2;
(sqrt2 now changed to  $\sqrt{2}$  for looks!)
```

$$\mathcal{A} = \begin{pmatrix} 4 * \sqrt{2} - 6 & -4 * \sqrt{2} + 7 & -3 * \sqrt{2} + 6 \\ 3 * \sqrt{2} - 6 & -3 * \sqrt{2} + 7 & -3 * \sqrt{2} + 6 \\ 3 * \sqrt{2} & 1 - 3 * \sqrt{2} & -2 * \sqrt{2} \end{pmatrix}$$

$$\text{ratjordan}(\mathcal{A}) = \left\{ \begin{pmatrix} \sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 \\ 0 & 0 & -3 * \sqrt{2} + 1 \end{pmatrix}, \begin{pmatrix} 7 * \sqrt{2} - 6 & \frac{2 * \sqrt{2} - 49}{31} & \frac{-21 * \sqrt{2} + 18}{31} \\ 3 * \sqrt{2} - 6 & \frac{21 * \sqrt{2} - 18}{31} & \frac{-21 * \sqrt{2} + 18}{31} \\ 3 * \sqrt{2} + 1 & \frac{-3 * \sqrt{2} + 24}{31} & \frac{3 * \sqrt{2} - 24}{31} \end{pmatrix}, \begin{pmatrix} 0 & \sqrt{2} + 1 & 1 \\ -1 & 4 * \sqrt{2} + 9 & 4 * \sqrt{2} \\ -1 & -\frac{1}{6} * \sqrt{2} + 1 & 1 \end{pmatrix} \right\}$$

16.38.9 modular

Calculations can be performed in a modular base by switching on `modular;`. The base can then be set by `setmod p;` (p a prime). The normal form will then have entries in $\mathbb{Z}/p\mathbb{Z}$.

By also switching on `balanced_mod;` the output will be shown using a symmetric modular representation.

Information on this modular manipulation can be found in *chapter 9* (Polynomials and Rationals) of the REDUCE User's Manual [5].

example

```
load_package normform;
on modular;
setmod 23;
```

$$\mathcal{A} = \begin{pmatrix} 10 & 18 \\ 17 & 20 \end{pmatrix}$$

```
jordansymbolic(A) =
```

$$\left\{ \begin{pmatrix} 18 & 0 \\ 0 & 12 \end{pmatrix}, \{\{\lambda + 5, \lambda + 11\}, \lambda\}, \begin{pmatrix} 15 & 9 \\ 22 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 14 \\ 1 & 15 \end{pmatrix} \right\}$$

```
on balanced_mod;
```

```
jordansymbolic(A) =
```

$$\left\{ \begin{pmatrix} -5 & 0 \\ 0 & -11 \end{pmatrix}, \{\{\lambda + 5, \lambda + 11\}, \lambda\}, \begin{pmatrix} -8 & 9 \\ -1 & 1 \end{pmatrix}, \begin{pmatrix} 1 & -9 \\ 1 & -8 \end{pmatrix} \right\}$$

Bibliography

- [1] T.M.L.Mulders and A.H.M. Levelt: *The Maple normform and Normform packages*. (1993)
- [2] T.M.L.Mulders: *Algoritmen in De Algebra, A Seminar on Algebraic Algorithms*, Nijmegen. (1993)
- [3] Roger A. Horn and Charles A. Johnson: *Matrix Analysis*. Cambridge University Press (1990)
- [4] Bruce W. Chat. . .[et al.]: *Maple (Computer Program)*. Springer-Verlag (1991)
- [5] Anthony C. Hearn: *REDUCE User's Manual 3.6*. RAND (1995)

16.39 NUMERIC: Solving numerical problems

This package implements basic algorithms of numerical analysis. These include:

- solution of algebraic equations by Newton's method

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2})
```

- solution of ordinary differential equations

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5)
```

- bounds of a function over an interval

```
bounds(sin x+x,x=(1 .. 2));
```

- minimizing a function (Fletcher Reeves steepest descent)

```
num_min(sin(x)+x/5, x);
```

- Chebyshev curve fitting

```
chebyshev_fit(sin x/x,x=(1 .. 3),5);
```

- numerical quadrature

```
num_int(sin x,x=(0 .. pi));
```

Author: Herbert Melenk.

The NUMERIC package implements some numerical (approximative) algorithms for REDUCE, based on the REDUCE rounded mode arithmetic. These algorithms are implemented for standard cases. They should not be called for ill-conditioned problems; please use standard mathematical libraries for these.

16.39.1 Syntax

Intervals, Starting Points

Intervals are generally coded as lower bound and upper bound connected by the operator `'..'`, usually associated to a variable in an equation. E.g.

```
x= (2.5 .. 3.5)
```

means that the variable x is taken in the range from 2.5 up to 3.5. Note, that the bounds can be algebraic expressions, which, however, must evaluate to numeric results. In cases where an interval is returned as the result, the lower and upper bounds can be extracted by the `PART` operator as the first and second part respectively. A starting point is specified by an equation with a numeric righthand side, e.g.

```
x=3.0
```

If for multivariate applications several coordinates must be specified by intervals or as a starting point, these specifications can be collected in one parameter (which is then a list) or they can be given as separate parameters alternatively. The list form is more appropriate when the parameters are built from other `REDUCE` calculations in an automatic style, while the flat form is more convenient for direct interactive input.

Accuracy Control

The keyword parameters $accuracy = a$ and $iterations = i$, where a and i must be positive integer numbers, control the iterative algorithms: the iteration is continued until the local error is below 10^{-a} ; if that is impossible within i steps, the iteration is terminated with an error message. The values reached so far are then returned as the result.

tracing

Normally the algorithms produce only a minimum of printed output during their operation. In cases of an unsuccessful or unexpected long operation a trace of the iteration can be printed by setting

```
on trnumeric;
```

16.39.2 Minima

The Fletcher Reeves version of the *steepest descent* algorithms is used to find the minimum of a function of one or more variables. The function must have continuous partial derivatives with respect to all variables. The starting point of the search can be specified; if not, random values are taken instead. The steepest descent algorithms in general find only local minima.

Syntax:

NUM_MIN (*exp*, *var*₁[= *val*₁][, *var*₂[= *val*₂]...]
[, *accuracy* = *a*][, *iterations* = *i*])

or

NUM_MIN (*exp*, {*var*₁[= *val*₁][, *var*₂[= *val*₂]...]}
[, *accuracy* = *a*][, *iterations* = *i*])

where *exp* is a function expression,

*var*₁, *var*₂, ... are the variables in *exp* and *val*₁, *val*₂, ... are the (optional) start values.

NUM_MIN tries to find the next local minimum along the descending path starting at the given point. The result is a list with the minimum function value as first element followed by a list of equations, where the variables are equated to the coordinates of the result point.

Examples:

```
num_min(sin(x)+x/5, x);

{4.9489585606, {X=29.643767785}}
```

```
num_min(sin(x)+x/5, x=0);

{ - 1.3342267466, {X= - 1.7721582671}}
```

```
% Rosenbrock function (well known as hard to minimize).
fktn := 100*(x1**2-x2)**2 + (1-x1)**2;
num_min(fktn, x1=-1.2, x2=1, iterations=200);

{0.00000021870228295, {X1=0.99953284494, X2=0.99906807238}}
```

16.39.3 Roots of Functions/ Solutions of Equations

An adaptively damped Newton iteration is used to find an approximative zero of a function, a function vector or the solution of an equation or an equation system. Equations are internally converted to a difference of lhs and rhs such that the Newton method (=zero detection) can be applied. The expressions must have continuous derivatives for all variables. A starting point for the iteration can be given. If not given, random values are taken instead. If the number of forms is not equal to the number of variables, the Newton method cannot be applied. Then the minimum of the sum of absolute squares is located instead.

With ON COMPLEX solutions with imaginary parts can be found, if either the expression(s) or the starting point contain a nonzero imaginary part.

Syntax:

NUM_SOLVE ($exp_1, var_1 [= val_1][, accuracy = a][, iterations = i]$)

or

NUM_SOLVE ($\{exp_1, \dots, exp_n\}, var_1 [= val_1], \dots, var_n [= val_n]$
 $[, accuracy = a][, iterations = i]$)

or

NUM_SOLVE ($\{exp_1, \dots, exp_n\}, \{var_1 [= val_1], \dots, var_n [= val_n]\}$
 $[, accuracy = a][, iterations = i]$)

where exp_1, \dots, exp_n are function expressions,

var_1, \dots, var_n are the variables,

val_1, \dots, val_n are optional start values.

NUM_SOLVE tries to find a zero/solution of the expression(s). Result is a list of equations, where the variables are equated to the coordinates of the result point.

The Jacobian matrix is stored as a side effect in the shared variable JACOBIAN.

Example:

```
num_solve({sin x=cos y, x + y = 1},{x=1,y=2});
```

```
{X= - 1.8561957251,Y=2.856195584}
```

```
jacobian;
```

```
[COS (X)   SIN (Y) ]
[          ]
[  1        1      ]
```

16.39.4 Integrals

For the numerical evaluation of univariate integrals over a finite interval the following strategy is used:

1. If the function has an antiderivative in close form which is bounded in the integration interval, this is used.
2. Otherwise a Chebyshev approximation is computed, starting with order 20, eventually up to order 80. If that is recognized as sufficiently convergent it is used for computing the integral by directly integrating the coefficient sequence.
3. If none of these methods is successful, an adaptive multilevel quadrature algorithm is used.

For multivariate integrals only the adaptive quadrature is used. This algorithm tolerates isolated singularities. The value *iterations* here limits the number of local interval intersection levels. *Accuracy* is a measure for the relative total discretization error (comparison of order 1 and order 2 approximations).

Syntax:

NUM_INT (*exp*, *var*₁ = (*l*₁..*u*₁)[, *var*₂ = (*l*₂..*u*₂) ...]

[, *accuracy* = *a*][, *iterations* = *i*])

where *exp* is the function to be integrated,

*var*₁, *var*₂, ... are the integration variables,

*l*₁, *l*₂, ... are the lower bounds,

*u*₁, *u*₂, ... are the upper bounds.

Result is the value of the integral.

Example:

```
num_int(sin x,x=(0 .. pi));
```

```
2.0000010334
```

16.39.5 Ordinary Differential Equations

A Runge-Kutta method of order 3 finds an approximate graph for the solution of a ordinary differential equation real initial value problem.

Syntax:

NUM_ODESOLVE (*exp*, *depvar* = *dv*, *indepvar*=(*from*..*to*)

[, *accuracy* = *a*][, *iterations* = *i*])

where

exp is the differential expression/equation,

depvar is an identifier representing the dependent variable (function to be found),

indepvar is an identifier representing the independent variable,

exp is an equation (or an expression implicitly set to zero) which contains the first derivative of *depvar* wrt *indepvar*,

from is the starting point of integration,

to is the endpoint of integration (allowed to be below *from*),

dv is the initial value of *depvar* in the point *indepvar* = *from*.

The ODE *exp* is converted into an explicit form, which then is used for a Runge Kutta iteration over the given range. The number of steps is controlled by the value of *i* (default: 20). If the steps are too coarse to reach the desired accuracy in the neighborhood of the starting point, the number is increased automatically.

Result is a list of pairs, each representing a point of the approximate solution of the ODE problem.

Example:

```
num_odesolve(df(y,x)=y,y=1,x=(0 .. 1), iterations=5);

{{0.0,1.0},{0.2,1.2214},{0.4,1.49181796},{0.6,1.8221064563},
{0.8,2.2255208258},{1.0,2.7182511366}}
```

Remarks:

- If in *exp* the differential is not isolated on the lefthand side, please ensure that the dependent variable is explicitly declared using a `DEPEND` statement, e.g.

```
depend y,x;
```

otherwise the formal derivative will be computed to zero by `REDUCE`.

- The `REDUCE` package `SOLVE` is used to convert the form into an explicit ODE. If that process fails or has no unique result, the evaluation is stopped with an error message.

16.39.6 Bounds of a Function

Upper and lower bounds of a real valued function over an interval or a rectangular multivariate domain are computed by the operator BOUNDS. The algorithmic basis is the computation with inequalities: starting from the interval(s) of the variables, the bounds are propagated in the expression using the rules for inequality computation. Some knowledge about the behavior of special functions like ABS, SIN, COS, EXP, LOG, fractional exponentials etc. is integrated and can be evaluated if the operator BOUNDS is called with rounded mode on (otherwise only algebraic evaluation rules are available).

If BOUNDS finds a singularity within an interval, the evaluation is stopped with an error message indicating the problem part of the expression.

Syntax:

BOUNDS (*exp*, *var*₁ = (*l*₁..*u*₁)[, *var*₂ = (*l*₂..*u*₂) ...])

BOUNDS (*exp*, {*var*₁ = (*l*₁..*u*₁)[, *var*₂ = (*l*₂..*u*₂) ...]})

where *exp* is the function to be investigated,

*var*₁, *var*₂, ... are the variables of *exp*,

*l*₁, *l*₂, ... and *u*₁, *u*₂, ... specify the area (intervals).

BOUNDS computes upper and lower bounds for the expression in the given area. An interval is returned.

Example:

```
bounds(sin x, x=(1 .. 2));

{-1, 1}

on rounded;
bounds(sin x, x=(1 .. 2));

0.84147098481 .. 1

bounds(x**2+x, x=(-0.5 .. 0.5));

- 0.25 .. 0.75
```

16.39.7 Chebyshev Curve Fitting

The operator family *Chebyshev_...* implements approximation and evaluation of functions by the Chebyshev method. Let $T_n^{(a,b)}(x)$ be the Chebyshev polynomial of order n transformed to the interval (a, b) . Then a function $f(x)$ can be approximated in (a, b) by a series

$$f(x) \approx \sum_{i=0}^N c_i T_i^{(a,b)}(x)$$

The operator *Chebyshev_fit* computes this approximation and returns a list, which has as first element the sum expressed as a polynomial and as second element the sequence of Chebyshev coefficients c_i . *Chebyshev_df* and *Chebyshev_int* transform a Chebyshev coefficient list into the coefficients of the corresponding derivative or integral respectively. For evaluating a Chebyshev approximation at a given point in the basic interval the operator *Chebyshev_eval* can be used. Note that *Chebyshev_eval* is based on a recurrence relation which is in general more stable than a direct evaluation of the complete polynomial.

CHEBYSHEV_FIT (*fcn*, *var* = (*lo*..*hi*), *n*)

CHEBYSHEV_EVAL (*coeffs*, *var* = (*lo*..*hi*), *var* = *pt*)

CHEBYSHEV_DF (*coeffs*, *var* = (*lo*..*hi*))

CHEBYSHEV_INT (*coeffs*, *var* = (*lo*..*hi*))

where *fcn* is an algebraic expression (the function to be fitted), *var* is the variable of *fcn*, *lo* and *hi* are numerical real values which describe an interval ($lo < hi$), *n* is the approximation order, an integer > 0 , set to 20 if missing, *pt* is a numerical value in the interval and *coeffs* is a series of Chebyshev coefficients, computed by one of *CHEBYSHEV_COEFF*, *_DF* or *_INT*.

Example:

```
on rounded;
```

```
w:=chebyshev_fit(sin x/x,x=(1 .. 3),5);
```

```

           3           2
w := {0.03824*x  - 0.2398*x  + 0.06514*x + 0.9778,
```

```
      {0.8991,-0.4066,-0.005198,0.009464,-0.00009511}}
```

```
chebyshev_eval(second w, x=(1 .. 3), x=2.1);
```

0.4111

16.39.8 General Curve Fitting

The operator *NUM_FIT* finds for a set of points the linear combination of a given set of functions (function basis) which approximates the points best under the objective of the least squares criterion (minimum of the sum of the squares of the deviation). The solution is found as zero of the gradient vector of the sum of squared errors.

Syntax:

NUM_FIT (*vals, basis, var = pts*)

where *vals* is a list of numeric values,

var is a variable used for the approximation,

pts is a list of coordinate values which correspond to *var*,

basis is a set of functions varying in *var* which is used for the approximation.

The result is a list containing as first element the function which approximates the given values, and as second element a list of coefficients which were used to build this function from the basis.

Example:

```
% approximate a set of factorials by a polynomial
pts:=for i:=1 step 1 until 5 collect i$
vals:=for i:=1 step 1 until 5 collect
      for j:=1:i product j$

num_fit(vals, {1, x, x**2}, x=pts);

      2
{14.571428571*X  - 61.428571429*X + 54.6, {54.6,
      - 61.428571429, 14.571428571}}

num_fit(vals, {1, x, x**2, x**3, x**4}, x=pts);
```

```

4
{2.2083333234*X  - 20.249999879*X
3
+ 67.791666154*X  - 93.749999133*X
2
+ 44.999999525,
{44.999999525, - 93.749999133, 67.791666154,
- 20.249999879, 2.2083333234}}

```

16.39.9 Function Bases

The following procedures compute sets of functions e.g. to be used for approximation. All procedures have two parameters, the expression to be used as *variable* (an identifier in most cases) and the order of the desired system. The functions are not scaled to a specific interval, but the *variable* can be accompanied by a scale factor and/or a translation in order to map the generic interval of orthogonality to another (e.g. $(x - 1/2) * 2\pi i$). The result is a function list with ascending order, such that the first element is the function of order zero and (for the polynomial systems) the function of order n is the $n + 1$ -th element.

<code>monomial_base(x,n)</code>	<code>{1,x,...,x**n}</code>
<code>trigonometric_base(x,n)</code>	<code>{1,sin x,cos x,sin(2x),cos(2x) ...}</code>
<code>Bernstein_base(x,n)</code>	Bernstein polynomials
<code>Legendre_base(x,n)</code>	Legendre polynomials
<code>Laguerre_base(x,n)</code>	Laguerre polynomials
<code>Hermite_base(x,n)</code>	Hermite polynomials
<code>Chebyshev_base_T(x,n)</code>	Chebyshev polynomials first kind
<code>Chebyshev_base_U(x,n)</code>	Chebyshev polynomials second kind

Example:

```
Bernstein_base(x,5);
```

```

5
{ - X  + 5*X  - 10*X  + 10*X  - 5*X + 1,
4
3
2

```

$$5 * X * (X^4 - 4 * X^3 + 6 * X^2 - 4 * X + 1),$$

$$10 * X^2 * (-X^3 + 3 * X^2 - 3 * X + 1),$$

$$10 * X^3 * (X^2 - 2 * X + 1),$$

$$5 * X^4 * (-X + 1),$$

$$X^5 \}$$

16.40 ODESOLVE: Ordinary differential equations solver

The ODESOLVE package is a solver for ordinary differential equations. At the present time it has very limited capabilities. It can handle only a single scalar equation presented as an algebraic expression or equation, and it can solve only first-order equations of simple types, linear equations with constant coefficients and Euler equations. These solvable types are exactly those for which Lie symmetry techniques give no useful information. For example, the evaluation of

```
depend (y, x) ;
odesolve (df (y, x) = x**2 + e**x, y, x) ;
```

yields the result

$$\{Y = \frac{X^3 + 3 \cdot E^X + 3 \cdot \text{ARBCONST}(1)}{3}\}$$

Main Author: Malcolm A.H. MacCallum.

Other contributors: Francis Wright, Alan Barnes.

16.40.1 Introduction

ODESolve 1+ is an experimental project to update and enhance the ordinary differential equation (ODE) solver (`odesolve`) that has been distributed as a standard component of REDUCE [2, 4, 3] for about 10 years. ODESolve 1+ is intended to provide a strict superset of the facilities provided by `odesolve`. This document describes a substantial re-implementation of previous versions of ODESolve 1+ that now includes almost none of the original `odesolve` code. This version is targeted at REDUCE 3.7 or later, and will not run in earlier versions. This project is being conducted partly under the auspices of the European CATHODE project [1]. Various test files, including three versions based on a published review of ODE solvers [7], are included in the ODESolve 1+ distribution. For further background see [10], which describes version 1.03. See also [11].

ODESolve 1+ is intended to implement some solution techniques itself (i.e. most of the simple and well known techniques [12]) and to provide an automatic interface to other more sophisticated solvers, such as PSODE [5, 6, 8] and CRACK [9], to handle cases where simple techniques fail. It is also intended to provide a unified interface to other special solvers, such as Laplace transforms, series solutions and numerical methods, under user request. Although none of these extensions

is explicitly implemented yet, a general extension interface is implemented (see §16.40.6).

The main motivation behind `ODESolve1+` is pragmatic. It is intended to meet user expectations, to have an easy user interface that normally does the right thing automatically, and to return solutions in the form that the user wants and expects. Quite a lot of development effort has been directed toward this aim. Hence, `ODESolve1+` solves common text-book special cases in preference to esoteric pathological special cases, and it endeavours to simplify solutions into convenient forms.

16.40.2 Installation

The file `odesolve.in` inputs the full set of source files that are required to implement `ODESolve1+` *assuming that the current directory is the `ODESolve1+` source directory*. Hence, `ODESolve1+` can be run without compiling it in any implementation of REDUCE 3.7 by starting REDUCE in the `ODESolve1+` source directory and entering the statement

```
1: in "odesolve.in"$
```

However, the recommended procedure is to compile it by starting REDUCE in the `ODESolve1+` source directory and entering the statements

```
1: faslout odesolve;
2: in "odesolve.in"$
3: faslend;
```

In CSL-REDUCE, this will work only if you have write access to the REDUCE image file (`reduce.img`), so you may need to set up a private copy first. In PSL-REDUCE, you may need to move the compiled image file `odesolve.b` to a directory in your PSL load path, such as the main fasl directory. Please refer to the documentation for your implementation of REDUCE for details. Once a compiled version of `ODESolve1+` has been correctly installed, it can be loaded by entering the REDUCE statement

```
1: load_package odesolve;
```

A string describing the current version of `ODESolve1+` is assigned to the algebraic-mode variable `odesolve_version`, which can be evaluated to check what version is actually in use.

In versions of REDUCE derived from the development source after 22 September 2000, use of the normal algebraic-mode `odesolve` operator causes the package to

autoload. However, the `ODESolve1+` global switches are not declared, and the symbolic mode interface provided for backward compatibility with the previous version is not defined, until after the package has loaded. The former is not a huge problem because all `ODESolve` switches can be accessed as optional arguments, and the backward compatibility interface should probably not be used in new code anyway.

16.40.3 User interface

The principal interface is via the operator `odesolve`. (It also has a synonym called `dsolve` to make porting of examples from Maple easier, but it does not accept general Maple syntax!) For purposes of description I will refer to the dependent variable as “ y ” and the independent variable as “ x ”, but of course the names are arbitrary. The general input syntax is

```
odesolve(ode, y, x, conditions, options);
```

All arguments except the first are optional. This is possible because, if necessary, `ODESolve1+` attempts to deduce the dependent and independent variables used and to make any necessary `DEPEND` declarations. Messages are output to indicate any assumptions or dependence declarations that are made. Here is an example of what is probably the shortest possible valid input:

```
odesolve(df(y,x));

*** Dependent var(s) assumed to be y

*** Independent var assumed to be x

*** depend y , x

{y=arbconst(1)}
```

Output of `ODESolve1+` messages is controlled by the standard `REDUCE` switch `msg`.

Specifying the ODE and its variables

The first argument (`ode`) is *required*, and must be either an ODE or a variable (or expression) that evaluates to an ODE. Automatic dependence declaration works *only* when the ODE is input *directly* as an argument to the `odesolve` operator. Here, “ODE” means an equation or expression containing one or more derivatives of y with respect to x . Derivatives of y with respect to other variables are not

allowed because `ODESolve1+` does not solve *partial* differential equations, and symbolic derivatives of variables other than y are treated as symbolic constants. An expression is implicitly equated to zero, as is usual in equation solvers.

The independent variable may be either an operator that explicitly depends on the independent variable, e.g. $y(x)$ (as required in Maple), or a simple variable that is declared (by the user or automatically by `ODESolve1+`) to depend on the independent variable. If the independent variable is an operator then it may depend on parameters as well as the independent variable. Variables may be simple identifiers or, more generally, REDUCE “kernels”, e.g.

```
operator x, y;
odesolve(df(y(x(a),b),x(a)) = 0);

*** Dependent var(s) assumed to be y(x(a),b)

*** Independent var assumed to be x(a)

{y(x(a),b)=arbconst(1)}
```

The order in which arguments are given must be preserved, but arguments may be omitted, except that if x is specified then y must also be specified, although an empty list `{ }` can be used as a “place-holder” to represent “no specified argument”. Variables are distinguished from options by requiring that if a variable is specified then it must appear in the ODE, otherwise it is assumed to be an option.

Generally in REDUCE it is not recommended to use the identifier `t` as a variable, since it is reserved in Lisp. However, it is very common practice in applied mathematics to use it as a variable to represent time, and for that reason `ODESolve1+` provides special support to allow it as either the independent or a dependent variable. But, of course, its use may still cause trouble in other parts of REDUCE!

Specifying conditions

If specified, the “conditions” argument must take the form of an (unordered) list of (unordered lists of) equations with either y , x , or a derivative of y on the left. A single list of conditions need not be contained within an outer list. Combinations of conditions are allowed. Conditions within one (inner) list all relate to the same x value. For example:

Boundary conditions:

```
{ {y=y0, x=x0}, {y=y1, x=x1}, ... }
```

Initial conditions:

```
{x=x0, y=y0, df(y,x)=dy0, ... }
```

Combined conditions:

$$\{\{y=y_0, x=x_0\}, \{df(y,x)=dy_1, x=x_1\}, \{df(y,x)=dy_2, y=y_2, x=x_2\}, \dots\}$$

Here is an example of boundary conditions:

```
odesolve(df(y,x,2) = y, y, x, {{x = 0, y = A}, {x = 1, y = B}});
```

$$\{y = \frac{-e^{2x} * a + e^{2x} * b * e^x + a * e^{2x} - b * e^{2x}}{e^x * e^{2x} - e^x}\}$$

Here is an example of initial conditions:

```
odesolve(df(y,x,2) = y, y, x, {x = 0, y = A, df(y,x) = B});
```

$$\{y = \frac{e^{2x} * a + e^{2x} * b + a - b}{2 * e^x}\}$$

Here is an example of combined conditions:

```
odesolve(df(y,x,2) = y, y, x, {{x=0, y=A}, {x=1, df(y,x)=B}});
```

$$\{y = \frac{e^{2x} * a + e^{2x} * b * e^x + a * e^{2x} - b * e^{2x}}{e^x * e^{2x} + e^x}\}$$

Boundary conditions on the values of y at various values of x may also be specified by replacing the variables by equations with single values or matching lists of values on the right, of the form

$$y = y_0, x = x_0$$

or

$$y = \{y_0, y_1, \dots\}, x = \{x_0, x_2, \dots\}$$

For example

```
odesolve(df(y,x) = y, y = A, x = 0);

      x
{y=e  *a}

odesolve(df(y,x,2) = y, y = {A, B}, x = {0, 1});

      2*x      2*x      2
      - e      *a + e      *b*e + a*e      - b*e
{y=-----}
      x  2      x
      e  *e      - e
```

Specifying options and defaults

The final arguments may be one or more of the option identifiers listed in the table below, which take precedence over the default settings. All options can also be specified on the right of equations with the identifier “output” on the left, e.g. “output = basis”. This facility is provided mainly for compatibility with other systems such as Maple, although it also allows options to be distinguished from variables in case of ambiguity. Some options can be specified on the left of equations that assign special values to the option. Currently, only “trode” and its synonyms can be assigned the value 1 to give an increased level of tracing.

The following switches set default options – they are all off by default. Options set locally using option arguments override the defaults set by switches.

Switch	Option	Effect on solution
odesolve_explicit	explicit	fully explicit
odesolve_expand	expand	expand roots of unity
odesolve_full	full	fully explicit and expanded
odesolve_implicit	implicit	implicit instead of parametric
	algint	turn on algint
odesolve_noint	noint	turn off selected integrations
odesolve_verbose	verbose	display ODE and conditions
odesolve_basis	basis	output basis solution for linear ODE
	trode	
trode	trace	turn on algorithm tracing
	tracing	
odesolve_fast	fast	turn off heuristics
odesolve_check	check	turn on solution checking

An “explicit” solution is an equation with *y* isolated on the left whereas an “implicit” solution is an equation that determines *y* as one or more of its solutions. A

“parametric” solution expresses both x and y in terms of some additional parameter. Some solution techniques naturally produce an explicit solution, but some produce either an implicit or a parametric solution. The “explicit” option causes `ODESolve1+` to attempt to convert solutions to explicit form, whereas the “implicit” option causes `ODESolve1+` to attempt to convert parametric solutions (only) to implicit form (by eliminating the parameter). These solution conversions may be slow or may fail in complicated cases.

`ODESolve1+` introduces two operators used in solutions: `root_of_unity` and `plus_or_minus`, the latter being a special case of the former, i.e. a second root of unity. These operators carry a tag that associates the same root of unity when it appears in more than one place in a solution (cf. the standard `root_of` operator). The “expand” option expands a single solution expressed in terms of these operators into a set of solutions that do not involve them. `ODESolve1+` also introduces two operators `expand_roots_of_unity` [which should perhaps be named `expand_root_of_unity`] and `expand_plus_or_minus`, that are used internally to perform the expansion described above, and can be used explicitly.

The “algint” option turns on “algebraic integration” locally only within `ODESolve1+`. It also loads the `algint` package if necessary. `Algint` allows `ODESolve1+` to solve some ODEs for which the standard `REDUCE` integrator hangs (i.e. takes an extremely long time to return). If the resulting solution contains unevaluated integrals then the `algint` switch should be turned on outside `ODESolve1+` before the solution is re-evaluated, otherwise the standard integrator may well hang again! For some ODEs, the `algint` option leads to better solutions than the standard `REDUCE` integrator.

Alternatively, the “noint” option prevents `REDUCE` from attempting to evaluate the integrals that arise in some solution techniques. If `ODESolve1+` takes too long to return a result then you might try adding this option to see if it helps solve this particular ODE, as illustrated in the test files. This option is provided to speed up the computation of solutions that contain integrals that cannot be evaluated, because in some cases `REDUCE` can spend a long time trying to evaluate such integrals before returning them unevaluated. This only affects integrals evaluated *within* the `ODESolve1+` operator. If a solution containing an unevaluated integral that was returned using the “noint” option is re-evaluated, it may again take `REDUCE` a very long time to fail to evaluate the integral, so considerable caution is recommended! (A global switch called “noint” is also installed when `ODESolve1+` is loaded, and can be turned on to prevent `REDUCE` from attempting to evaluate *any* integrals. But this effect may be very confusing, so this switch should be used only with extreme care. If you turn it on and then forget, you may wonder why `REDUCE` seems unable to evaluate even trivial integrals!)

The “verbose” option causes `ODESolve1+` to display the ODE, variables and conditions as it sees them internally, after pre-processing. This is intended for use

in demonstrations and possibly for debugging, and not really for general users.

The “basis” option causes `ODESolve1+` to output the general solutions of linear ODEs in basis format (explained below). Special solutions (of ODEs with conditions) and solutions of nonlinear ODEs are not affected.

The “trode” (or “trace” or “tracing”) option turns on tracing of the algorithms used by `ODESolve1+`. It reports its classification of the ODE and any intermediate results that it computes, such as a chain of progressively simpler (in some sense) ODEs that finally leads to a solution. Tracing can produce a lot of output, e.g. see the test log file “`zimmer.rlg`”. The option “`trode = 1`” or the global assignment “`!*trode := 1`” causes `ODESolve1+` to report every test that it tries in its classification process, producing even more tracing output. This is probably most useful for debugging, but it may give the curious user further insight into the operation of `ODESolve1+`.

The “fast” option disables all non-deterministic solution techniques (including most of those for nonlinear ODEs of order > 1). It may be most useful if `ODESolve1+` is used as a subroutine, including calling it recursively in a hook. It makes `ODESolve1+` behave like the `odesolve` distributed with REDUCE versions up to and including 3.7, and so does not affect the `odesolve.tst` file. The “fast” option causes `ODESolve1+` to return no solution fast in cases where, by default, it would return either a solution or no solution (perhaps much) more slowly. Solution of sufficiently simple “deterministically-solvable” ODEs is unaffected.

The “check” option turns on checking of the solution. This checking is performed by code that is largely independent of the solver, so as to perform a genuinely independent check. It is not turned on by default so as to avoid the computational overhead, which is currently of the order of 30%. A check is made that each component solution satisfies the ODE and that a general solution contains at least enough arbitrary constants, or equivalently that a basis solution contains enough basis functions. Otherwise, warning messages are output. It is possible that `ODESolve1+` may fail to verify a solution because the automatic simplification fails, which indicates a failure in the checker rather than in the solver. This option is not yet well tested; please report any checking failures to me (FJW).

In some cases, in particular when an implicit solution contains an unevaluated integral, the checker may need to differentiate an integral with respect to a variable other than the integration variable. In order to do this, it turns on the differentiator switch “`allowdfint`” globally. [I hope that this setting will eventually become the default.] In some cases, in particular in symbolic solutions of Clairaut ODEs, the checker may need to differentiate a composition of operators using the chain rule. In order to do this, it turns on the differentiator switch “`expanddf`” locally only. Although the code to support both these differentiator facilities has been in REDUCE for a while, they both require patches that are currently only applied when

ODESolve 1+ is loaded. [I hope that these patches will eventually become part of REDUCE itself.]

16.40.4 Output syntax

If ODESolve 1+ is successful it outputs a list of sub-solutions that together represent the solution of the input ODE. Each sub-solution is either an equation that defines a branch of the solution, explicitly or implicitly, or it is a list of equations that define a branch of the solution parametrically in the form $\{y = G(p), x = F(p), p\}$. Here p is the parameter, which is actually represented in terms of an operator called `arbparam` which has an integer argument to distinguish it from other unrelated parameters, as usual for arbitrary values in REDUCE.

A general solution will contain a number of arbitrary constants represented by an operator called `arbconst` with an integer argument to distinguish it from other unrelated arbitrary constants. A special solution resulting from applying conditions will contain fewer (usually no) arbitrary constants.

The general solution of a linear ODE in basis format is a list consisting of a list of basis functions for the solution space of the reduced ODE followed by a particular solution if the input ODE had a y -independent “driver” term, i.e. was not reduced (which is sometimes ambiguously called “homogeneous”). The particular solution is normally omitted if it is zero. The dependent variable y does not appear in a basis solution. The linear solver uses basis solutions internally.

Currently, there are cases where ODESolve 1+ cannot solve a linear ODE using its linear solution techniques, in which case it will try nonlinear techniques. These may generate a solution that is not (obviously) a linear combination of basis solutions. In this case, if a basis solution has been requested, ODESolve 1+ will report that it cannot separate the nonlinear combination, which it will return as the default linear combination solution.

If ODESolve 1+ fails to solve the ODE then it will return a list containing the input ODE (always in the form of a differential expression equated to 0). At present, ODESolve 1+ does not return partial solutions. If it fails to solve any part of the problem then it regards this as complete failure. (You can probably see if this has happened by turning on algorithm tracing.)

16.40.5 Solution techniques

The ODESolve 1+ interface module pre-processes the problem and applies any conditions to the solution. The other modules deal with the actual solution.

ODESolve 1+ first classifies the input ODE according to whether it is linear or nonlinear and calls the appropriate solver. An ODE that consists of a product of

linear factors is regarded as nonlinear. The second main classification is based on whether the input ODE is of first or higher degree.

Solution proceeds essentially by trying to reduce nonlinear ODEs to linear ones, and to reduce higher order ODEs to first order ODEs. Only simple linear ODEs and simple first-order nonlinear ODEs can be solved directly. This approach involves considerable recursion within `ODESolve1+`.

If all solution techniques fail then `ODESolve1+` attempts to factorize the derivative of the whole ODE, which sometimes leads to a solution.

Linear solution techniques

`ODESolve1+` splits every linear ODE into a “reduced ODE” and a “driver” term. The driver is the component of the ODE that is independent of y , the reduced ODE is the component of the ODE that depends on y , and the sign convention is such that the ODE can be written in the form “reduced ODE = driver”. The reduced ODE is then split into a list of “ODE coefficients”.

The linear solver now determines the order of the ODE. If it is 1 then the ODE is immediately solved using an integrating factor (if necessary). For a higher order linear ODE, `ODESolve1+` considers a sequence of progressively more complicated solution techniques. For most purposes, the ODE is made “monic” by dividing through by the coefficient of the highest order derivative. This puts the ODE into a standard form and effectively deals with arbitrary overall algebraic factors that would otherwise confuse the solution process. (Hence, there is no need to perform explicit algebraic factorization on linear ODEs.) The only situation in which the original non-monic form of the ODE is considered is when checking for exactness, which may depend critically on otherwise irrelevant overall factors.

If the ODE has constant coefficients then it can (in principle) be solved using elementary “D-operator” techniques in terms of exponentials via an auxiliary equation. However, this works only if the polynomial auxiliary equation can be solved. Assuming that it can and there is a driver term, `ODESolve1+` tries to use a method based on inverse “D-operator” techniques that involves repeated integration of products of the solutions of the reduced ODE with the driver. Experience (by Malcolm MacCallum) suggests that this normally gives the most satisfactory form of solution if the integrals can be evaluated. If any integral fails to evaluate, the more general method of “variation of parameters”, based on the Wronskian of the solution set of the reduced ODE, is used instead. This involves only a single integral and so can never lead to nested unevaluated integrals.

If the ODE has non-constant coefficients then it may be of Euler (sometimes ambiguously called “homogeneous”) type, which can be trivially reduced to an ODE with constant coefficients. A shift in x is accommodated in this process. Next it is tested for exactness, which leads to a first integral that is an ODE of order one

lower. After that it is tested for the explicit absence of y and low order derivatives, which allows trivial order reduction. Then the monic ODE is tested for exactness, and if that fails and the original ODE was non-monic then the original form is tested for exactness.

Finally, pattern matching is used to seek a solution involving special functions, such as Bessel functions. Currently, this is implemented only for second-order ODEs satisfied by Bessel and Airy-integral functions. It could easily be extended to other orders and other special functions. Shifts in x could also be accommodated in the pattern matching. [Work to enhance this component of `ODESolve 1+` is currently in progress.]

If all linear techniques fail then `ODESolve 1+` currently calls the variable interchange routine (described below), which takes it into the nonlinear solver. Occasionally, this is successful in producing some, although not necessarily the best, solution of a linear ODE.

Nonlinear solution techniques

In order to handle trivial nonlinearity, `ODESolve 1+` first factorizes the ODE algebraically, solves each factor that depends on y and then merges the resulting solutions. Other factors are ignored, but a warning is output unless they are purely numerical.

If all attempts at solution fail then `ODESolve 1+` checks whether the original (unfactored) ODE was exact, because factorization could destroy exactness. Currently, `ODESolve 1+` handles only first and second order nonlinear exact ODEs.

A version of the main solver applied to each algebraic factor branches depending on whether the ODE factor is linear or nonlinear, and the nonlinear solver branches depending on whether the order is 1 or higher and calls one of the solvers described in the next two sections. If that solver fails, `ODESolve 1+` checks for exactness (of the factor). If that fails, it checks whether only a single order derivative is involved and tries to solve algebraically for that. If successful, this decomposes the ODE into components that are, in some sense, simpler and may be solvable. (However, in some cases these components are algebraically very complicated examples of simple types of ODE that the integrator cannot in practice handle, and it can take a very long time before returning an unevaluated integral.)

If all else fails, `ODESolve 1+` interchanges the dependent and independent variables and calls the top-level solver recursively. It keeps a list of all ODEs that have entered the top-level solver in order to break infinite loops that could arise if the solution of the variable-interchanged ODE fails.

First-order nonlinear solution techniques If the ODE is a first-degree polynomial in the derivative then `ODESolve1+` represents it in terms of the “gradient”, which is a function of x and y such that the ODE can be written as “ $dy/dx = \text{gradient}$ ”. It then checks *in sequence* for the following special types of ODE, each of which it can (in principle) solve:

Separable The gradient has the form $f(x)g(y)$, leading immediately to a solution by quadrature, i.e. the solution can be immediately written in terms of indefinite integrals. (This is considered to be a solution of the ODE, regardless of whether the integrals can be evaluated.) The solver recognises both explicit and implicit dependence when detecting separable form.

Quasi-separable The gradient has the form $f(y + kx)$, which is (trivially) separable after a linear transformation. It arises as a special case of the “quasi-homogeneous” case below, but is better treated earlier as a case in its own right.

Homogeneous The gradient has the form $f(y/x)$, which is algebraically homogeneous. A substitution of the form “ $y = vx$ ” leads to a first-order linear ODE that is (in principle) immediately solvable.

Quasi-homogeneous The gradient has the form $f(\frac{a_1x+b_1y+c_1}{a_2x+b_2y+c_2})$, which is homogeneous after a linear transformation.

Bernoulli The gradient has the form $P(x)y + Q(x)y^n$, in which case the ODE is a first-order linear ODE for y^{1-n} .

Riccati The gradient has the form $a(x)y^2 + b(x)y + c(x)$, in which case the ODE can be transformed into a *linear* second-order ODE that may be solvable.

If the ODE is not first-degree then it may be linear in either x or y . Solving by taking advantage of this leads to a parametric solution of the original ODE, in which the parameter corresponds to y' . It may then be possible to eliminate the parameter to give either an implicit or explicit solution.

An ODE is “solvable for y ” if it can be put into the form $y = f(x, y')$. Differentiating with respect to x leads to a first-order ODE for $y'(x)$, which may be easier to solve than the original ODE. The special case that $y = xF(y') + G(y')$ is called a Lagrange (or d’Alembert) ODE. Differentiating with respect to x leads to a first-order linear ODE for $x(y')$. The even more special case that $y = xy' + G(y')$, which may arise in the equivalent implicit form $F(xy' - y) = G(y')$, is called a Clairaut ODE. The general solution is given by replacing y' by an arbitrary constant, and it may be possible to obtain a singular solution by differentiating and solving the resulting factors simultaneously with the original ODE.

An ODE is “solvable for x ” if it can be put into the form $x = f(y, y')$. Differentiating with respect to y leads to a first-order ODE for $y'(y)$, which may be easier to solve than the original ODE.

Currently, `ODESolve 1+` recognises the above forms only if the ODE manifestly has the specified form and does not try very hard to actually solve for x or y , which perhaps it should!

Higher-order nonlinear solution techniques The techniques used here are all special cases of Lie symmetry analysis, which is not yet applied in any general way.

Higher-order nonlinear ODEs are passed through a number of “simplifier” filters that are applied in succession, regardless of whether the previous filter simplifies the ODE or not. Currently, the first filter tests for the explicit absence of y and low order derivatives, which allows trivial order reduction. The second filter tests whether the ODE manifestly depends on $x + k$ for some constant k , in which case it shifts x to remove k .

After that, `ODESolve 1+` tests for each of the following special forms in sequence. The sequence used here is important, because the classification is not unique, so it is important to try the most useful classification first.

Autonomous An ODE is autonomous if it does not depend explicitly on x , in which case it can be reduced to an ODE in y' of order one lower.

Scale invariant or equidimensional in x An ODE is scale invariant if it is invariant under the transformation $x \rightarrow ax, y \rightarrow a^p y$, where a is an arbitrary indeterminate and p is a constant to be determined. It can be reduced to an autonomous ODE, and thence to an ODE of order one lower. The special case $p = 0$ is called equidimensional in x . It is the nonlinear generalization of the (reduced) linear Euler ODE.

Equidimensional in y An ODE is equidimensional in y if it is invariant under the transformation $y \rightarrow ay$. An exponential transformation of y leads to an ODE of the same order that *may* be “more linear” and so easier to solve, but there is no guarantee of this. All (reduced) linear ODEs are trivially equidimensional in y .

The recursive nature of `ODESolve 1+`, especially the thread described in this section, can lead to complicated “arbitrary constant expressions”. Arbitrary constants must be included at the point where an ODE is solved by quadrature. Further processing of such a solution, as may happen when a recursive solution stack is unwound, can lead to arbitrary constant expressions that should be re-written as simple arbitrary constants. There is some simple code included to perform this arbitrary constant simplification, but it is rudimentary and not entirely successful.

16.40.6 Extension interface

The idea is that the ODEsolve extension interface allows any user to add solution techniques without needing to edit and recompile the main source code, and (in principle) without needing to be intimately familiar with the internal operation of ODEsolve 1+.

The extension interface consists of a number of “hooks” at various critical places within ODEsolve 1+. These hooks are modelled in part on the hook mechanism used to extend and customize the Emacs editor, which is a large Lisp-based system with a structure similar to that of REDUCE. Each ODEsolve 1+ hook is an identifier which can be defined to be a function (i.e. a procedure), or have assigned to it (in symbolic mode) a function name or a (symbolic mode) list of function names. The function should be written to accept the arguments specified for the particular hook, and it should return either a solution to the specified class of ODE in the specified form or nil.

If a hook returns a non-nil value then that value is used by ODEsolve 1+ as the solution of the ODE at that stage of the solution process. [If the ODE being solved was generated internally by ODEsolve 1+ or conditions are imposed then the solution will be re-processed before being finally returned by ODEsolve 1+.] If a hook returns nil then it is ignored and ODEsolve 1+ proceeds as if the hook function had not been called at all. This is the same mechanism that it used internally by ODEsolve 1+ to run sub-solvers. If a hook evaluates to a list of function names then they are applied in turn to the hook arguments until a non-nil value is returned and this is the value of the hook; otherwise the hook returns nil. The same code is used to run all hooks and it checks that an identifier is the name of a function before it tries to apply it; otherwise the identifier is ignored. However, the hook code does not perform any other checks, so errors within functions run by hooks will probably terminate ODEsolve 1+ and errors in the return value will probably cause fatal errors later in ODEsolve 1+. Such errors are user errors rather than ODEsolve 1+ errors!

Hooks are defined in pairs which are inserted before and after critical stages of the solver, which currently means the general ODE solver, the nonlinear ODE solver, and the solver for linear ODEs of order greater than one (on the grounds that solving first order linear ODEs is trivial and the standard ODEsolve 1+ code should always suffice). The precise interface definition is as follows.

A reference to an “algebraic expression” implies that the REDUCE representation is a prefix or pseudo-prefix form. A reference to a “variable” means an identifier (and never a more general kernel). The “order” of an ODE is always an explicit positive integer. The return value of a hook function must always be either nil or an algebraic-mode list (which must be represented as a prefix form). Since the input and output of hook functions uses prefix forms (and never standard quotient forms), hook functions can equally well be written in either algebraic or symbolic

mode, and in fact `ODESolve 1+` uses a mixture internally. (An algebraic-mode procedure can return nil by returning nothing. The integer zero is *not* equivalent to nil in the context of `ODESolve 1+` hooks.)

Hook names: `ODESolve_Before_Hook`, `ODESolve_After_Hook`.

Run before and after: The general ODE solver.

Arguments: 3

1. The ODE in the form of an algebraic expression with no denominator that must be made identically zero by the solution.
2. The dependent variable.
3. The independent variable.

Return value: A list of equations exactly as returned by `ODESolve 1+` itself.

Hook names: `ODESolve_Before_Non_Hook`, `ODESolve_After_Non_Hook`.

Run before and after: The nonlinear ODE solver.

Arguments: 4

1. The ODE in the form of an algebraic expression with no denominator that must be made identically zero by the solution.
2. The dependent variable.
3. The independent variable.
4. The order of the ODE.

Return value: A list of equations exactly as returned by `ODESolve 1+` itself.

Hook names: `ODESolve_Before_Lin_Hook`, `ODESolve_After_Lin_Hook`.

Run before and after: The general linear ODE solver.

Arguments: 6

1. A list of the coefficient functions of the “reduced ODE”, i.e. the coefficients of the different orders (including zero) of derivatives of the dependent variable, each in the form of an algebraic expression, in low to high derivative order. [In general the ODE will not be “monic” so the leading (i.e. last) coefficient function will not be 1. Hence, the ODE may contain an essentially irrelevant overall algebraic factor.]
2. The “driver” term, i.e. the term involving only the independent variable, in the form of an algebraic expression. The sign convention is such that “reduced ODE = driver”.
3. The dependent variable.
4. The independent variable.
5. The (maximum) order (> 1) of the ODE.
6. The minimum order derivative present.

Return value: A list consisting of a basis for the solution space of the reduced ODE and optionally a particular integral of the full ODE. This list does not contain any equations, and the dependent variable never appears in it. The particular integral may be omitted if it is zero. The basis is itself a list of algebraic expressions in the independent variable. (Hence the return value is always a list and its first element is also always a list.)

Hook names: `ODESolve_Before_NonlGrad_Hook`,
`ODESolve_After_NonlGrad_Hook`.

Run before and after: The solver for first-order first-degree nonlinear (“gradient”) ODEs, which can be expressed in the form $dy/dx = \text{gradient}(y, x)$.

Arguments: 3

1. The “gradient”, which is an algebraic expression involving (in general) the dependent and independent variables, to which the ODE equates the derivative.
2. The dependent variable.
3. The independent variable.

Return value: A list of equations exactly as returned by `ODESolve 1+` itself. (In this case the list should normally contain precisely one equation.)

The file `extend.tst` contains a very simple test and demonstration of the operation of the first three classes of hook.

This extension interface is experimental and subject to change. Please check the version of this document (or the source code) for the version of `ODESolve1+` you are actually running.

16.40.7 Change log

27 February 1999 Version 1.06 frozen.

13 July 2000 Version 1.061 added an extension interface.

8 August 2000 Version 1.062 added the “fast” option.

21 September 2000 Version 1.063 added the “trace”, “check” and “algint” options, the “NonlGrad” hooks, handling of implicit dependence in separable ODEs, and handling of the general class of quasi-homogeneous ODEs.

28 September 2000 Version 1.064 added support for using ‘t’ as a variable and replaced the version identification output by the `odesolve_version` variable.

14 August 2001 Version 1.065 fixed obscure bugs in the first-order nonlinear ODE handler and the arbitrary constant simplifier, and revised some tracing messages slightly.

16.40.8 Planned developments

- Extend special-function solutions and allow shifts in x .
- Improve solution of linear ODEs, by (a) using linearity more generally to solve as “CF + PI”, (b) finding at least polynomial solutions of ODEs with polynomial coefficients, (c) implementing non-trivial reduction of order.
- Improve recognition of exact ODEs, and add some support for more general use of integrating factors.
- Add a “classify” option, that turns on trode but avoids any actual solution, to report all possible (?) top-level classifications.
- Improve `arbconst` and `arbparam` simplification.
- Add more standard elementary techniques and more general techniques such as Lie symmetry, Prelle-Singer, etc.

- Improve integration support, preferably to remove the need for the “noint” option.
- Solve systems of ODEs, including under- and over-determined ODEs and systems. Link to CRACK (Wolf) and/or DiffGrob2 (Mansfield)?
- Move more of the implementation to symbolic-mode code.

Bibliography

- [1] CATHODE (Computer Algebra Tools for Handling Ordinary Differential Equations) <http://www-lmc.imag.fr/CATHODE/>, <http://www-lmc.imag.fr/CATHODE2/>.
- [2] A. C. Hearn and J. P. Fitch (ed.), *REDUCE User's Manual 3.6*, RAND Publication CP78 (Rev. 7/95), RAND, Santa Monica, CA 90407-2138, USA (1995).
- [3] M. A. H. MacCallum, An Ordinary Differential Equation Solver for REDUCE, *Proc. ISSAC '88*, ed. P. Gianni, *Lecture Notes in Computer Science* **358**, Springer-Verlag (1989), 196–205.
- [4] M. A. H. MacCallum, ODESOLVE, \LaTeX file `reduce/doc/odesolve.tex` distributed with REDUCE 3.6. The first part of this document is included in the printed REDUCE User's Manual 3.6 [2], 345–346.
- [5] Y.-K. Man, *Algorithmic Solution of ODEs and Symbolic Summation using Computer Algebra*, PhD Thesis, School of Mathematical Sciences, Queen Mary and Westfield College, University of London (July 1994).
- [6] Y.-K. Man and M. A. H. MacCallum, A Rational Approach to the Prelle-Singer Algorithm, *J. Symbolic Computation*, **24** (1997), 31–43.
- [7] F. Postel and P. Zimmermann, A Review of the ODE Solvers of AXIOM, DERIVE, MAPLE, MATHEMATICA, MACSYMA, MUPAD and REDUCE, *Proceedings of the 5th Rhine Workshop on Computer Algebra, April 1-3, 1996, Saint-Louis, France*. Specific references are to the version dated April 11, 1996. The latest version of this review, together with log files for each of the systems, is available from <http://www.loria.fr/~zimmerma/ComputerAlgebra/>.
- [8] M. J. Prell and M. F. Singer, Elementary First Integrals of Differential Equations, *Trans. AMS* **279** (1983), 215–229.

- [9] T. Wolf and A. Brand, The Computer Algebra Package CRACK for Investigating PDEs, \LaTeX file `reduce/doc/crack.tex` distributed with REDUCE 3.6. A shorter document is included in the printed REDUCE User's Manual 3.6 [2], 241–244.
- [10] F. J. Wright, An Enhanced ODE Solver for REDUCE. *Programmirovaniye* No 3 (1997), 5–22, in Russian, and *Programming and Computer Software* No 3 (1997), in English.
- [11] F. J. Wright, Design and Implementation of `ODESolve1+` : An Enhanced REDUCE ODE Solver. CATHODE Workshop Report, Marseilles, May 1999, CATHODE (1999).
<http://centaur.maths.qmw.ac.uk/Papers/Marseilles/>.
- [12] D. Zwillinger, *Handbook of Differential Equations*, Academic Press. (Second edition 1992.)

16.41 ORTHOVEC: Manipulation of scalars and vectors

ORTHOVEC is a collection of REDUCE procedures and operations which provide a simple-to-use environment for the manipulation of scalars and vectors. Operations include addition, subtraction, dot and cross products, division, modulus, div, grad, curl, laplacian, differentiation, integration, and Taylor expansion.

Author: James W. Eastwood.

The revised version of ORTHOVEC is a collection of REDUCE 3.4 procedures and operations which provide a simple to use environment for the manipulation of scalars and vectors. Operations include addition, subtraction, dot and cross products, division, modulus, div, grad, curl, laplacian, differentiation, integration, $\mathbf{a} \cdot \nabla$ and Taylor expansion. Version 2 is summarized in [?]. It differs from the original ([?]) in revised notation and extended capabilities.

16.41.1 Introduction

The revised version of ORTHOVEC[?] is, like the original[?], a collection of REDUCE procedures and operators designed to simplify the machine aided manipulation of vectors and vector expansions frequently met in many areas of applied mathematics. The revisions have been introduced for two reasons: firstly, to add extra capabilities missing from the original and secondly, to tidy up input and output to make the package easier to use.

The changes from Version 1 include:

1. merging of scalar and vector unary and binary operators, $+$, $-$, $*$, $/$
2. extensions of the definitions of division and exponentiation to vectors
3. new vector dependency procedures
4. application of l'Hôpital's rule in limits and Taylor expansions
5. a new component selector operator
6. algebraic mode output of LISP vector components

The LISP vector primitives are again used to store vectors, although with the introduction of LIST types in algebraic mode in REDUCE 3.4, the implementation may have been more simply achieved using lists to store vector components.

The philosophy used in Version 2 follows that used in the original: namely, algebraic mode is used wherever possible. The view is taken that some computational inefficiencies are acceptable if it allows coding to be intelligible to (and thence adaptable by) users other than LISP experts familiar with the internal workings of REDUCE.

Procedures and operators in ORTHOVEC fall into the five classes: initialisation, input-output, algebraic operations, differential operations and integral operations. Definitions are given in the following sections, and a summary of the procedure names and their meanings are given in Table 1. The final section discusses test examples.

16.41.2 Initialisation

The procedure VSTART initialises ORTHOVEC. It may be called after ORTHOVEC has been INPUTted (or LOADED if a fast load version has been made) to reset coordinates. VSTART provides a menu of standard coordinate systems:-

1. cartesian $(x, y, z) = (x, y, z)$
2. cylindrical $(r, \theta, z) = (r, \theta, z)$
3. spherical $(r, \theta, \phi) = (r, \theta, \phi)$
4. general $(u_1, u_2, u_3) = (u_1, u_2, u_3)$
5. others

which the user selects by number. Selecting options (1)-(4) automatically sets up the coordinates and scale factors. Selection option (5) shows the user how to select another coordinate system. If VSTART is not called, then the default cartesian coordinates are used. ORTHOVEC may be re-initialised to a new coordinate system at any time during a given REDUCE session by typing

```
VSTART $.
```

16.41.3 Input-Output

ORTHOVEC assumes all quantities are either scalars or 3 component vectors. To define a vector a with components (c_1, c_2, c_3) use the procedure SVEC as follows

```
a := svec(c1, c2, c3);
```

The standard REDUCE output for vectors when using the terminator “;” is to list the three components inside square brackets $[\dots]$, with each component in prefix form. A replacement for the standard REDUCE procedure MAPRIN is included in the package to change the output of LISP vector components to algebraic notation. The procedure VOUT (which returns the value of its argument) can be used to give labelled output of components in algebraic form: e.g.,

```
b := svec (sin(x)**2, y**2, z)$
vout (b) $
```

The operator `_` can be used to select a particular component (1, 2 or 3) for output e.g.

```
b_1 ;
```

16.41.4 Algebraic Operations

Six infix operators, sum, difference, quotient, times, exponentiation and cross product, and four prefix operators, plus, minus, reciprocal and modulus are defined in ORTHOVEC. These operators can take suitable combinations of scalar and vector arguments, and in the case of scalar arguments reduce to the usual definitions of $+$, $-$, $*$, $/$, etc.

The operators are represented by symbols

```
+, -, /, *, ^, ><
```

The composite `><` is an attempt to represent the cross product symbol \times in ASCII

characters. If we let \mathbf{v} be a vector and s be a scalar, then valid combinations of arguments of the procedures and operators and the type of the result are as summarised below. The notation used is

result := procedure(left argument, right argument) or

result := (left operand) operator (right operand) .

Vector Addition

$\mathbf{v} := \text{VECTORPLUS}(\mathbf{v})$ or $\mathbf{v} := + \mathbf{v}$
 $s := \text{VECTORPLUS}(s)$ or $s := + s$
 $\mathbf{v} := \text{VECTORADD}(\mathbf{v}, \mathbf{v})$ or $\mathbf{v} := \mathbf{v} + \mathbf{v}$
 $s := \text{VECTORADD}(s, s)$ or $s := s + s$

Vector Subtraction

$\mathbf{v} := \text{VECTORMINUS}(\mathbf{v})$ or $\mathbf{v} := - \mathbf{v}$
 $s := \text{VECTORMINUS}(s)$ or $s := - s$
 $\mathbf{v} := \text{VECTORDIFFERENCE}(\mathbf{v}, \mathbf{v})$ or $\mathbf{v} := \mathbf{v} - \mathbf{v}$
 $s := \text{VECTORDIFFERENCE}(s, s)$ or $s := s - s$

Vector Division

$\mathbf{v} := \text{VECTORRECIP}(\mathbf{v})$ or $\mathbf{v} := / \mathbf{v}$
 $s := \text{VECTORRECIP}(s)$ or $s := / s$
 $\mathbf{v} := \text{VECTORQUOTIENT}(\mathbf{v}, \mathbf{v})$ or $\mathbf{v} := \mathbf{v} / \mathbf{v}$
 $\mathbf{v} := \text{VECTORQUOTIENT}(\mathbf{v}, s)$ or $\mathbf{v} := \mathbf{v} / s$
 $\mathbf{v} := \text{VECTORQUOTIENT}(s, \mathbf{v})$ or $\mathbf{v} := s / \mathbf{v}$
 $s := \text{VECTORQUOTIENT}(s, s)$ or $s := s / s$

Vector Multiplication

$\mathbf{v} := \text{VECTORTIMES}(s, \mathbf{v})$ or $\mathbf{v} := s * \mathbf{v}$
 $\mathbf{v} := \text{VECTORTIMES}(\mathbf{v}, s)$ or $\mathbf{v} := \mathbf{v} * s$
 $s := \text{VECTORTIMES}(\mathbf{v}, \mathbf{v})$ or $s := \mathbf{v} * \mathbf{v}$
 $s := \text{VECTORTIMES}(s, s)$ or $s := s * s$

Vector Cross Product

$\mathbf{v} := \text{VECTORCROSS}(\mathbf{v}, \mathbf{v})$ or $\mathbf{v} := \mathbf{v} \times \mathbf{v}$

Vector Exponentiation

$s := \text{VECTOREXPT}(\mathbf{v}, s)$ or $s := \mathbf{v} ^ s$
 $s := \text{VECTOREXPT}(s, s)$ or $s := s ^ s$

Vector Modulus

$s := \text{VMOD}(s)$
 $s := \text{VMOD}(\mathbf{v})$

All other combinations of operands for these operators lead to error messages being issued. The first two instances of vector multiplication are scalar multiplication of vectors, the third is the product of two scalars and the last is the inner (dot) product. The prefix operators $+$, $-$, $/$ can take either scalar or vector arguments and return results of the same type as their arguments. VMOD returns a scalar.

In compound expressions, parentheses may be used to specify the order of combi-

```

s  := div (v)
v  := grad(s)
v  := curl(v)
v  := delsq(v)
s  := delsq(s)
v  := v dotgrad v
s  := v dotgrad s

```

Table 16.12: ORTHOVEC valid combinations of operator and argument

nation. If parentheses are omitted the ordering of the operators, in increasing order of precedence is

+ | - | dotgrad | * | >< | ^ | _

and these are placed in the precedence list defined in REDUCE after <. The differential operator DOTGRAD is defined in the following section, and the component selector _ was introduced in section 3.

Vector divisions are defined as follows: If **a** and **b** are vectors and *c* is a scalar, then

$$\begin{aligned} \mathbf{a}/\mathbf{b} &= \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{b}|^2} \\ c/\mathbf{a} &= \frac{c\mathbf{a}}{|\mathbf{a}|^2} \end{aligned}$$

Both scalar multiplication and dot products are given by the same symbol, braces are advisable to ensure the correct precedences in expressions such as $(\mathbf{a} \cdot \mathbf{b})(\mathbf{c} \cdot \mathbf{d})$.

Vector exponentiation is defined as the power of the modulus:

$$\mathbf{a}^n \equiv \text{VMOD}(a)^n = |\mathbf{a}|^n$$

16.41.5 Differential Operations

Differential operators provided are div, grad, curl, delsq, and dotgrad. All but the last of these are prefix operators having a single vector or scalar argument as appropriate. Valid combinations of operator and argument, and the type of the result are shown in table 16.12.

All other combinations of operator and argument type cause error messages to be issued. The differential operators have their usual meanings [?]. The coordinate system used by these operators is set by invoking VSTART (cf. Sec. 16.41.2). The names h1, h2 and h3 are reserved for the scale factors, and u1, u2 and u3 are used for the coordinates.

VEX	VX	VPT	VORDER
v	v	v	v
v	v	v	s
v	s	s	s
s	v	v	v
s	v	v	s
s	s	s	s

Table 16.13: ORTHOVEC valid combination of argument types.

A vector extension, VDF, of the REDUCE procedure DF allows the differentiation of a vector (scalar) with respect to a scalar to be performed. Allowed forms are $\text{VDF}(\mathbf{v}, s) \rightarrow \mathbf{v}$ and $\text{VDF}(s, s) \rightarrow s$, where, for example

$$\text{vdf}(\mathbf{B}, x) \equiv \frac{\partial \mathbf{B}}{\partial x}$$

The standard REDUCE procedures DEPEND and NODEPEND have been redefined to allow dependences of vectors to be compactly defined. For example

```
a := svec(a1,a2,a3)$;
depend a,x,y;
```

causes all three components a1,a2 and a3 of a to be treated as functions of x and y. Individual component dependences can still be defined if desired.

```
depend a3,z;
```

The procedure VTAYLOR gives truncated Taylor series expansions of scalar or vector functions:-

```
vtaylor(vex,vx,vpt,vorder);
```

returns the series expansion of the expression VEX with respect to variable VX about point VPT to order VORDER. Valid combinations of argument types are shown in table 16.13.

Any other combinations cause error messages to be issued. Elements of VORDER must be non-negative integers, otherwise error messages are issued. If scalar VORDER is given for a vector expansion, expansions in each component are truncated at the same order, VORDER.

The new version of Taylor expansion applies l'Hôpital's rule in evaluating coef-

ficients, so handle cases such as $\sin(x)/(x)$, etc. which the original version of ORTHOVEC could not. The procedure used for this is LIMIT, which can be used directly to find the limit of a scalar function ex of variable x at point pt :-

```
ans := limit(ex,x,pt);
```

16.41.6 Integral Operations

Definite and indefinite vector, volume and scalar line integration procedures are included in ORTHOVEC. They are defined as follows:

$$\begin{aligned} \text{VINT}(\mathbf{v}, x) &= \int \mathbf{v}(x) dx \\ \text{DVINT}(\mathbf{v}, x, a, b) &= \int_a^b \mathbf{v}(x) dx \\ \text{VOLINT}(\mathbf{v}) &= \int \mathbf{v} h_1 h_2 h_3 du_1 du_2 du_3 \\ \text{DVOLINT}(\mathbf{v}, \mathbf{l}, \mathbf{u}, n) &= \int_1^{\mathbf{u}} \mathbf{v} h_1 h_2 h_3 du_1 du_2 du_3 \\ \text{LINEINT}(\mathbf{v}, \omega, t) &= \int \mathbf{v} \cdot d\mathbf{r} \equiv \int v_i h_i \frac{\partial \omega_i}{\partial t} dt \\ \text{DLINEINT}(\mathbf{v}, \omega t, a, b) &= \int_a^b v_i h_i \frac{\partial \omega_i}{\partial t} dt \end{aligned}$$

In the vector and volume integrals, \mathbf{v} are vector or scalar, a, b, x and n are scalar. Vectors \mathbf{l} and \mathbf{u} contain expressions for lower and upper bounds to the integrals. The integer index n defines the order in which the integrals over u_1, u_2 and u_3 are performed in order to allow for functional dependencies in the integral bounds:

n	order
1	$u_1 \ u_2 \ u_3$
2	$u_3 \ u_1 \ u_2$
3	$u_2 \ u_3 \ u_1$
4	$u_1 \ u_3 \ u_2$
5	$u_2 \ u_1 \ u_3$
otherwise	$u_3 \ u_2 \ u_1$

The vector ω in the line integral's arguments contain explicit paramterisation of the coordinates u_1, u_2, u_3 of the line $\mathbf{u}(t)$ along which the integral is taken.

Procedures	Description
VSTART	select coordinate system
SVEC	set up a vector
VOUT	output a vector
VECTORCOMPONENT	- extract a vector component (1-3)
VECTORADD	+ add two vectors or scalars
VECTORPLUS	+ unary vector or scalar plus
VECTORMINUS	- unary vector or scalar minus
VECTORDIFFERENCE	- subtract two vectors or scalars
VECTORQUOTIENT	/ vector divided by scalar
VECTORRECIP	/ unary vector or scalar division (reciprocal)
VECTORTIMES	* multiply vector or scalar by vector/scalar
VECTORCROSS	>< cross product of two vectors
VECTOREXPT	^ exponentiate vector modulus or scalar
VMOD	length of vector or scalar

Table 16.14: Procedures names and operators used in ORTHOVEC (part 1)

16.41.7 Test Cases

To use the REDUCE source version of ORTHOVEC, initiate a REDUCE session and then IN the file *orthovec.red* containing ORTHOVEC. However, it is recommended that for efficiency a compiled fast loading version be made and LOADED when required (see Sec. 18 of the REDUCE manual). If coordinate dependent differential and integral operators other than cartesian are needed, then VSTART must be used to reset coordinates and scale factors.

Six simple examples are given in the Test Run Output file *orthovectest.log* to illustrate the working of ORTHOVEC. The input lines were taken from the file *orthovectest.red* (the Test Run Input), but could equally well be typed in at the Terminal.

Example 47

Show that

$$(\mathbf{a} \times \mathbf{b}) \cdot (\mathbf{c} \times \mathbf{d}) - (\mathbf{a} \cdot \mathbf{c})(\mathbf{b} \cdot \mathbf{d}) + (\mathbf{a} \cdot \mathbf{d})(\mathbf{b} \cdot \mathbf{c}) \equiv 0$$

Example 48

Procedures	Description
DIV	divergence of vector
GRAD	gradient of scalar
CURL	curl of vector
DELSQ	laplacian of scalar or vector
DOTGRAD	(vector).grad(scalar or vector)
VTAYLOR	vector or scalar Taylor series of vector or scalar
VPTAYLOR	vector or scalar Taylor series of scalar
TAYLOR	scalar Taylor series of scalar
LIMIT	limit of quotient using l'Hôpital's rule
VINT	vector integral
DVINT	definite vector integral
VOLINT	volume integral
DVOLINT	definite volume integral
LINEINT	line integral
DLINEINT	definite line integral
MAPRIN	vector extension of REDUCE MAPRIN
DEPEND	vector extension of REDUCE DEPEND
NODEPEND	vector extension of REDUCE NODEPEND

Table 16.15: Procedures names and operators used in ORTHOVEC (part 2)

Write the equation of motion

$$\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} + \nabla p - \text{curl}(\mathbf{B}) \times \mathbf{B}$$

in cylindrical coordinates.

Example 49

Taylor expand

- $\sin(x) \cos(y) + e^z$ about the point $(0, 0, 0)$ to third order in x , fourth order in y and fifth order in z .
- $\sin(x)/x$ about x to fifth order.
- \mathbf{v} about $\mathbf{x} = (x, y, z)$ to fifth order, where $\mathbf{v} = (x/\sin(x), (e^y - 1)/y, (1 + z)^{10})$.

Example 50

Obtain the second component of the equation of motion in example 48, and the first component of the final vector Taylor series in example 49.

Example 51

Evaluate the line integral

$$\int_{\mathbf{r}_1}^{\mathbf{r}_2} \mathbf{A} \cdot d\mathbf{r}$$

from point $\mathbf{r}_1 = (1, 1, 1)$ to point $\mathbf{r}_2 = (2, 4, 8)$ along the path $(x, y, z) = (s, s^2, s^3)$ where

$$\mathbf{A} = (3x^2 + 5y)\mathbf{i} - 12xy\mathbf{j} + 2xyz^2\mathbf{k}$$

and $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ are unit vectors in the (x, y, z) directions.

Example 52

Find the volume V common to the intersecting cylinders $x^2 + y^2 = r^2$ and $x^2 + z^2 = r^2$ i.e. evaluate

$$V = 8 \int_0^r dx \int_0^{ub} dy \int_0^{ub} dz$$

where $ub = \sqrt{r^2 - x^2}$

16.42 PHYSOP: Operator calculus in quantum theory

This package has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly of the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space.

Author: Mathias Warns.

16.42.1 Introduction

The package PHYSOP has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly in the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space. Since the capabilities of the current REDUCE release to deal with complex expressions containing noncommutative operators are rather restricted, the first step was to enhance these possibilities in order to achieve a better usability of REDUCE for these kind of calculations. This has led to the development of a first package called NONCOM2 which is described in section 2. For more complicated expressions involving both scalar quantities and operators the need for an additional data type has emerged in order to make a clear separation between the various objects present in the calculation. The implementation of this new REDUCE data type is realized by the PHYSOP (for PHYSical OPERator) package described in section 3.

16.42.2 The NONCOM2 Package

The package NONCOM2 redefines some standard REDUCE routines in order to modify the way noncommutative operators are handled by the system. In standard REDUCE declaring an operator to be noncommutative using the NONCOM statement puts a global flag on the operator. This flag is checked when the system has to decide whether or not two operators commute during the manipulation of an expression.

The NONCOM2 package redefines the NONCOM statement in a way more suitable for calculations in physics. Operators have now to be declared noncommutative pairwise, i.e. coding:

```
NONCOM A,B;
```

declares the operators A and B to be noncommutative but allows them to commute with any other (noncommutative or not) operator present in the expression. In a similar way if one wants e.g. $A(X)$ and $A(Y)$ not to commute, one has now to code:

```
NONCOM A, A;
```

Each operator gets a new property list containing the operators with which it does not commute. A final example should make the use of the redefined NONCOM statement clear:

```
NONCOM A, B, C;
```

declares A to be noncommutative with B and C, B to be noncommutative with A and C and C to be noncommutative with A and B. Note that after these declaration e.g. $A(X)$ and $A(Y)$ are still commuting kernels.

Finally to keep the compatibility with standard REDUCE declaring a single identifier using the NONCOM statement has the same effect as in standard REDUCE i.e., the identifier is flagged with the NONCOM tag.

From the user's point of view there are no other new commands implemented by the package. Commutation relations have to be declared in the standard way as described in the manual i.e. using LET statements. The package itself consists of several redefined standard REDUCE routines to handle the new definition of noncommutativity in multiplications and pattern matching processes.

CAVEAT: Due to its nature, the package is highly version dependent. The current version has been designed for the 3.3 and 3.4 releases of REDUCE and may not work with previous versions. Some different (but still correct) results may occur by using this package in conjunction with LET statements since part of the pattern matching routines have been redesigned. The package has been designed to bridge a deficiency of the current REDUCE version concerning the notion of noncommutativity and it is the author's hope that it will be made obsolete by a future release of REDUCE.

16.42.3 The PHYSOP package

The package PHYSOP implements a new REDUCE data type to perform calculations with physical operators. The noncommutativity of operators is implemented using the NONCOM2 package so this file should be loaded prior to the use of

PHYSOP³⁷. In the following the new commands implemented by the package are described. Beside these additional commands, the full set of standard REDUCE instructions remains available for performing any other calculation.

Type declaration commands

The new REDUCE data type PHYSOP implemented by the package allows the definition of a new kind of operators (i.e. kernels carrying an arbitrary number of arguments). Throughout this manual, the name “operator” will refer, unless explicitly stated otherwise, to this new data type. This data type is in turn divided into 5 subtypes. For each of this subtype, a declaration command has been defined:

`SCALOP A;` declares A to be a scalar operator. This operator may carry an arbitrary number of arguments i.e. after the declaration: `SCALOP A;` all kernels of the form e.g. `A(J)`, `A(1,N)`, `A(N,L,M)` are recognized by the system as being scalar operators.

`VECOPI V;` declares V to be a vector operator. As for scalar operators, the vector operators may carry an arbitrary number of arguments. For example `V(3)` can be used to represent the vector operator \vec{V}_3 . Note that the dimension of space in which this operator lives is arbitrary. One can however address a specific component of the vector operator by using a special index declared as `PHYSINDEX` (see below). This index must then be the first in the argument list of the vector operator.

`TENSOP C(3);` declares C to be a tensor operator of rank 3. Tensor operators of any fixed integer rank larger than 1 can be declared. Again this operator may carry an arbitrary number of arguments and the space dimension is not fixed. The tensor components can be addressed by using special `PHYSINDEX` indices (see below) which have to be placed in front of all other arguments in the argument list.

`STATE U;` declares U to be a state, i.e. an object on which operators have a certain action. The state U can also carry an arbitrary number of arguments.

`PHYSINDEX X;` declares X to be a special index which will be used to address components of vector and tensor operators.

It is very important to understand precisely the way how the type declaration commands work in order to avoid type mismatch errors when using the PHYSOP package. The following examples should illustrate the way the program interprets type declarations. Assume that the declarations listed above have been typed in by the user, then:

³⁷To build a fast loading version of PHYSOP the NONCOM2 source code should be read in prior to the PHYSOP code

- $A, A(1, N), A(N, M, K)$ are SCALAR operators.
- $V, V(3), V(N, M)$ are VECTOR operators.
- $C, C(5), C(Y, Z)$ are TENSOR operators of rank 3.
- $U, U(P), U(N, L, M)$ are STATES.

BUT: $V(X), V(X, 3), V(X, N, M)$ are all scalar operators since the special index X addresses a specific component of the vector operator (which is a scalar operator). Accordingly, $C(X, X, X)$ is also a scalar operator because the diagonal component C_{xxx} of the tensor operator C is meant here (C has rank 3 so 3 special indices must be used for the components).

In view of these examples, every time the following text refers to scalar operators, it should be understood that this means not only operators defined by the SCALOP statement but also components of vector and tensor operators. Depending on the situation, in some case when dealing only with the components of vector or tensor operators it may be preferable to use an operator declared with SCALOP rather than addressing the components using several special indices (throughout the manual, indices declared with the PHYSINDEX command are referred to as special indices).

Another important feature of the system is that for each operator declared using the statements described above, the system generates 2 additional operators of the same type: the adjoint and the inverse operator. These operators are accessible to the user for subsequent calculations without any new declaration. The syntax is as following:

If A has been declared to be an operator (scalar, vector or tensor) the adjoint operator is denoted $A!+$ and the inverse operator is denoted $A!-1$ (an inverse adjoint operator $A!+!-1$ is also generated). The exclamation marks do not appear when these operators are printed out by REDUCE (except when the switch NAT is set to off) but have to be typed in when these operators are used in an input expression. An adjoint (but no inverse) state is also generated for every state defined by the user. One may consider these generated operators as "placeholders" which means that these operators are considered by default as being completely independent of the original operator. Especially if some value is assigned to the original operator, this value is not automatically assigned to the generated operators. The user must code additional assignment statements in order to get the corresponding values.

Exceptions from these rules are (i) that inverse operators are always ordered at the same place as the original operators and (ii) that the expressions $A!-1 * A$ and $A * A!-1$ are replaced³⁸ by the unit operator UNIT. This operator is defined as a scalar operator during the initialization of the PHYSOP package. It should be used to indicate the type of an operator expression whenever no other PHYSOP occur

³⁸This may not always occur in intermediate steps of a calculation due to efficiency reasons.

in it. For example, the following sequence:

```
SCALOP A;  
A:= 5;
```

leads to a type mismatch error and should be replaced by:

```
SCALOP A;  
A:=5*UNIT;
```

The operator `UNIT` is a reserved variable of the system and should not be used for other purposes.

All other kernels (including standard `REDUCE` operators) occurring in expressions are treated as ordinary scalar variables without any `PHYSOP` type (referred to as scalars in the following). Assignment statements are checked to ensure correct operator type assignment on both sides leading to an error if a type mismatch occurs. However an assignment statement of the form `A:= 0` or `LET A = 0` is always valid regardless of the type of `A`.

Finally a command `CLEARPHYSOP` has been defined to remove the `PHYSOP` type from an identifier in order to use it for subsequent calculations (e.g. as an ordinary `REDUCE` operator). However it should be remembered that no substitution rule is cleared by this function. It is therefore left to the user's responsibility to clear previously all substitution rules involving the identifier from which the `PHYSOP` type is removed.

Users should be very careful when defining procedures or statements of the type `FOR ALL ... LET ...` that the `PHYSOP` type of all identifiers occurring in such expressions is unambiguously fixed. The type analysing procedure is rather restrictive and will print out a "PHYSOP type conflict" error message if such ambiguities occur.

Ordering of operators in an expression

The ordering of kernels in an expression is performed according to the following rules:

1. Scalars are always ordered ahead of PHYSOP operators in an expression. The `REDUCE` statement `KORDER` can be used to control the ordering of scalars but has no effect on the ordering of operators.
2. The default ordering of operators follows the order in which they have been declared (and not the alphabetical one). This ordering scheme can be changed

using the command `OPORDER`. Its syntax is similar to the `KORDER` statement, i.e. coding: `OPORDER A, V, F;` means that all occurrences of the operator `A` are ordered ahead of those of `V` etc. It is also possible to include operators carrying indices (both normal and special ones) in the argument list of `OPORDER`. However including objects not defined as operators (i.e. scalars or indices) in the argument list of the `OPORDER` command leads to an error.

3. Adjoint operators are placed by the declaration commands just after the original operators on the `OPORDER` list. Changing the place of an operator on this list means not that the adjoint operator is moved accordingly. This adjoint operator can be moved freely by including it in the argument list of the `OPORDER` command.

Arithmetic operations on operators

The following arithmetic operations are possible with operator expressions:

1. Multiplication or division of an operator by a scalar.
2. Addition and subtraction of operators of the same type.
3. Multiplication of operators is only defined between two scalar operators.
4. The scalar product of two `VECTOR` operators is implemented with a new function `DOT`. The system expands the product of two vector operators into an ordinary product of the components of these operators by inserting a special index generated by the program. To give an example, if one codes:

```
VECOPI V, W;
V DOT W;
```

the system will transform the product into:

```
V (IDX1) * W (IDX1)
```

where `IDX1` is a `PHYSINDEX` generated by the system (called a `DUMMY INDEX` in the following) to express the summation over the components. The identifiers `IDXn` (`n` is a nonzero integer) are reserved variables for this purpose and should not be used for other applications. The arithmetic operator `DOT` can be used both in infix and prefix form with two arguments.

5. Operators (but not states) can only be raised to an integer power. The system expands this power expression into a product of the corresponding number of terms inserting dummy indices if necessary. The following examples explain the

transformations occurring on power expressions (system output is indicated with an `-->`):

```
SCALOP A; A**2;
- --> A*A
VECOPI V; V**4;
- --> V (IDX1) *V (IDX1) *V (IDX2) *V (IDX2)
TENSOP C (2); C**2;
- --> C (IDX3, IDX4) *C (IDX3, IDX4)
```

Note in particular the way how the system interprets powers of tensor operators which is different from the notation used in matrix algebra.

6. Quotients of operators are only defined between scalar operator expressions. The system transforms the quotient of 2 scalar operators into the product of the first operator times the inverse of the second one. Example³⁹:

```
SCALOP A, B; A / B;
-1
--> (B ) *A
```

7. Combining the last 2 rules explains the way how the system handles negative powers of operators:

```
SCALOP B;
B**(-3);
-1 -1 -1
--> (B ) * (B ) * (B )
```

The method of inserting dummy indices and expanding powers of operators has been chosen to facilitate the handling of complicated operator expressions and particularly their application on states (see section 3.4.3). However it may be useful to get rid of these dummy indices in order to enhance the readability of the system's final output. For this purpose the switch `CONTRACT` has to be turned on (`CONTRACT` is normally set to `OFF`). The system in this case contracts over dummy indices reinserting the `DOT` operator and reassembling the expanded powers. However due to the predefined operator ordering the system may not remove all the dummy indices introduced previously.

³⁹This shows how inverse operators are printed out when the switch `NAT` is on

Special functions

Commutation relations If 2 PHYSOPs have been declared noncommutative using the (redefined) NONCOM statement, it is possible to introduce in the environment elementary (anti-) commutation relations between them. For this purpose, 2 scalar operators COMM and ANTICOMM are available. These operators are used in conjunction with LET statements. Example:

```
SCALOP A,B,C,D;
LET COMM(A,B)=C;
FOR ALL N,M LET ANTICOMM(A(N),B(M))=D;
VECOPI U,V,W; PHYSINDEX X,Y,Z;
FOR ALL X,Y LET COMM(V(X),W(Y))=U(Z);
```

Note that if special indices are used as dummy variables in FOR ALL ... LET constructs then these indices should have been declared previously using the PHYSINDEX command.

Every time the system encounters a product term involving 2 noncommutative operators which have to be reordered on account of the given operator ordering, the list of available (anti-) commutators is checked in the following way: First the system looks for a commutation relation which matches the product term. If it fails then the defined anticommutation relations are checked. If there is no successful match the product term $A*B$ is replaced by:

```
A*B;
--> COMM(A,B) + B*A
```

so that the user may introduce the commutation relation later on.

The user may want to force the system to look for anticommutators only; for this purpose a switch ANTICOM is defined which has to be turned on (ANTICOM is normally set to OFF). In this case, the above example is replaced by:

```
ON ANTICOM;
A*B;
--> ANTICOMM(A,B) - B*A
```

Once the operator ordering has been fixed (in the example above B has to be ordered ahead of A), there is no way to prevent the system from introducing (anti-)commutators every time it encounters a product whose terms are not in the right

order. On the other hand, simply by changing the `OPORDER` statement and reevaluating the expression one can change the operator ordering without the need to introduce new commutation relations. Consider the following example:

```
SCALOP A,B,C;   NONCOM A,B;   OPORDER B,A;
LET COMM(A,B)=C;
A*B;
- --> B*A + C;
OPORDER A,B;
  B*A;
- --> A*B - C;
```

The functions `COMM` and `ANTICOMM` should only be used to define elementary (anti-) commutation relations between single operators. For the calculation of (anti-) commutators between complex operator expressions, the functions `COMMUTE` and `ANTICOMMUTE` have been defined. Example (is included as example 1 in the test file):

```
VE COP P,A,K;
PHYSINDEX X,Y;
FOR ALL X,Y LET COMM(P(X),A(Y))=K(X)*A(Y);
COMMUTE(P**2,P DOT A);
```

Adjoint expressions As has been already mentioned, for each operator and state defined using the declaration commands quoted in section 3.1, the system generates automatically the corresponding adjoint operator. For the calculation of the adjoint representation of a complicated operator expression, a function `ADJ` has been defined. Example⁴⁰:

```
SCALOP A,B;
ADJ(A*B);
      +      +
--> (B )*(A )
```

Application of operators on states For this purpose, a function `OPAPPLY` has been defined. It has 2 arguments and is used in the following combinations:

⁴⁰This shows how adjoint operators are printed out when the switch `NAT` is on

(i) `LET OPAPPLY (operator, state) = state`; This is to define a elementary action of an operator on a state in analogy to the way elementary commutation relations are introduced to the system. Example:

```
SCALOP A; STATE U;
FOR ALL N,P LET OPAPPLY ( (A(N) , U(P)) = EXP (I*N*P) *U(P) ;
```

(ii) `LET OPAPPLY (state, state) = scalar exp.`; This form is to define scalar products between states and normalization conditions. Example:

```
STATE U;
FOR ALL N,M LET OPAPPLY (U(N) , U(M) ) = IF N=M THEN 1 ELSE 0;
```

(iii) `state := OPAPPLY (operator expression, state)`; In this way, the action of an operator expression on a given state is calculated using elementary relations defined as explained in (i). The result may be assigned to a different state vector.

(iv) `OPAPPLY (state, OPAPPLY (operator expression, state))`; This is the way how to calculate matrix elements of operator expressions. The system proceeds in the following way: first the rightmost operator is applied on the right state, which means that the system tries to find an elementary relation which match the application of the operator on the state. If it fails the system tries to apply the leftmost operator of the expression on the left state using the adjoint representations. If this fails also, the system prints out a warning message and stops the evaluation. Otherwise the next operator occurring in the expression is taken and so on until the complete expression is applied. Then the system looks for a relation expressing the scalar product of the two resulting states and prints out the final result. An example of such a calculation is given in the test file.

The infix version of the `OPAPPLY` function is the vertical bar `|`. It is right associative and placed in the precedence list just above the minus (`-`) operator. Some of the `REDUCE` implementation may not work with this character, the prefix form should then be used instead⁴¹.

16.42.4 Known problems in the current release of PHYSOP

(i) Some spurious negative powers of operators may appear in the result of a calculation using the `PHYSOP` package. This is a purely "cosmetic" effect which is due to an additional factorization of the expression in the output printing routines of `REDUCE`. Setting off the `REDUCE` switch `ALLFAC` (`ALLFAC` is normally on)

⁴¹The source code can also be modified to choose another special character for the function

should make these terms disappear and print out the correct result (see example 1 in the test file).

(ii) The current release of the PHYSOP package is not optimized w.r.t. computation speed. Users should be aware that the evaluation of complicated expressions involving a lot of commutation relations requires a significant amount of CPU time and memory. Therefore the use of PHYSOP on small machines is rather limited. A minimal hardware configuration should include at least 4 MB of memory and a reasonably fast CPU (type Intel 80386 or equiv.).

(iii) Slightly different ordering of operators (especially with multiple occurrences of the same operator with different indices) may appear in some calculations due to the internal ordering of atoms in the underlying LISP system (see last example in the test file). This cannot be entirely avoided by the package but does not affect the correctness of the results.

16.42.5 Compilation of the packages

To build a fast loading module of the NONCOM2 package, enter the following commands after starting the REDUCE system:

```
faslout "noncom2";
  in "noncom2.red";
faslend;
```

To build a fast loading module of the PHYSOP package, enter the following commands after starting the REDUCE system:

```
faslout "physop";
  in "noncom2.red";
  in "physop.red";
faslend;
```

Input and output file specifications may change according to the underlying operating system.

On PSL-based systems, a spurious message:

```
*** unknown function PHYSOP!*SQ called from compiled code
```

may appear during the compilation of the PHYSOP package. This warning has no effect on the functionality of the package.

16.42.6 Final remarks

The package PHYSOP has been presented by the author at the IV inter. Conference on Computer Algebra in Physical Research, Dubna (USSR) 1990 (see M. Warns, *Software Extensions of REDUCE for Operator Calculus in Quantum Theory*, Proc. of the IV inter. Conf. on Computer Algebra in Physical Research, Dubna 1990, to appear). It has been developed with the aim in mind to perform calculations of the type exemplified in the test file included in the distribution of this package. However it should also be useful in some other domains like e.g. the calculations of complicated Feynman diagrams in QCD which could not be performed using the HEPHYS package. The author is therefore grateful for any suggestion to improve or extend the usability of the package. Users should not hesitate to contact the author for additional help and explanations on how to use this package. Some bugs may also appear which have not been discovered during the tests performed prior to the release of this version. Please send in this case to the author a short input and output listing displaying the encountered problem.

Acknowledgements

The main ideas for the implementation of a new data type in the REDUCE environment have been taken from the VECTOR package developed by Dr. David Harper (D. Harper, *Comp. Phys. Comm.* **54** (1989) 295). Useful discussions with Dr. Eberhard Schröder and Prof. John Fitch are also gratefully acknowledged.

16.42.7 Appendix: List of error and warning messages

In the following the error (E) and warning (W) messages specific to the PHYSOP package are listed.

`cannot declare x as data type (W):` An attempt has been made to declare an object x which cannot be used as a PHYSOP operator of the required type. The declaration command is ignored.

`x already defined as data type (W):` The object x has already been declared using a REDUCE type declaration command and can therefore not be used as a PHYSOP operator. The declaration command is ignored.

`x already declared as data type (W):` The object x has already been declared with a PHYSOP declaration command. The declaration command is ignored.

`x is not a PHYSOP (E):` An invalid argument has been included in an OPORDER command. Check the arguments.

`invalid argument(s) to function` (E): A function implemented by the PHYSOP package has been called with an invalid argument. Check type of arguments.

`Type conflict in operation` (E): A PHYSOP type conflict has occurred during an arithmetic operation. Check the arguments.

`invalid call of function with args: arguments` (E): A function of the PHYSOP package has been declared with invalid argument(s). Check the argument list.

`type mismatch in expression` (E): A type mismatch has been detected in an expression. Check the corresponding expression.

`type mismatch in assignement` (E): A type mismatch has been detected in an assignment or in a LET statement. Check the listed statement.

`PHYSOP type conflict in expr` (E): A ambiguity has been detected during the type analysis of the expression. Check the expression.

`operators in exponent cannot be handled` (E): An operator has occurred in the exponent of an expression.

`cannot raise a state to a power` (E): states cannot be exponentiated by the system.

`invalid quotient` (E): An invalid denominator has occurred in a quotient. Check the expression.

`physops of different types cannot be commuted` (E): An invalid operator has occurred in a call of the COMMUTE/ANTICOMMUTE function.

`commutators only implemented between scalar operators` (E): An invalid operator has occurred in the call of the COMMUTE/ANTICOMMUTE function.

`evaluation incomplete due to missing elementary relations` (W):
The system has not found all the elementary commutators or application relations necessary to calculate or reorder the input expression. The result may however be used for further calculations.

16.43 PM: A REDUCE pattern matcher

PM is a general pattern matcher similar in style to those found in systems such as SMP and Mathematica, and is based on the pattern matcher described in Kevin McIsaac, “Pattern Matching Algebraic Identities”, SIGSAM Bulletin, 19 (1985), 4-13.

Documentation for this package is in plain text.

Author: Kevin McIsaac.

16.44 RANDPOLY: A random polynomial generator

This package is based on a port of the Maple random polynomial generator together with some support facilities for the generation of random numbers and anonymous procedures.

Author: Francis J. Wright.

This package is based on a port of the Maple random polynomial generator together with some support facilities for the generation of random numbers and anonymous procedures.

16.44.1 Introduction

The operator `randpoly` is based on a port of the Maple random polynomial generator. In fact, although by default it generates a univariate or multivariate polynomial, in its most general form it generates a sum of products of arbitrary integer powers of the variables multiplied by arbitrary coefficient expressions, in which the variable powers and coefficient expressions are the results of calling user-supplied functions (with no arguments). Moreover, the “variables” can be arbitrary expressions, which are composed with the underlying polynomial-like function.

The user interface, code structure and algorithms used are essentially identical to those in the Maple version. The package also provides an analogue of the Maple `rand` random-number-generator generator, primarily for use by `randpoly`. There are principally two reasons for translating these facilities rather than designing comparable facilities anew: (1) the Maple design seems satisfactory and has already been “proven” within Maple, so there is no good reason to repeat the design effort; (2) the main use for these facilities is in testing the performance of other algebraic code, and there is an advantage in having essentially the same test data generator implemented in both Maple and REDUCE. Moreover, it is interesting to see the extent to which a facility can be translated without change between two systems. (This aspect will be described elsewhere.)

Sections 16.44.2 and 16.44.3 describe respectively basic and more advanced use of `randpoly`; §16.44.4 describes subsidiary functions provided to support advanced use of `randpoly`; §16.44.5 gives examples; an appendix gives some details of the only non-trivial algorithm, that used to compute random sparse polynomials. Additional examples of the use of `randpoly` are given in the test and demonstration file `randpoly.tst`.

16.44.2 Basic use of `randpoly`

The operator `randpoly` requires at least one argument corresponding to the polynomial variable or variables, which must be either a single expression or a list of expressions.⁴² In effect, `randpoly` replaces each input expression by an internal variable and then substitutes the input expression for the internal variable in the generated polynomial (and by default expands the result as usual), although in fact if the input expression is a REDUCE kernel then it is used directly. The rest of this document uses the term “variable” to refer to a general input expression or the internal variable used to represent it, and all references to the polynomial structure, such as its degree, are with respect to these internal variables. The actual degree of a generated polynomial might be different from its degree in the internal variables.

By default, the polynomial generated has degree 5 and contains 6 terms. Therefore, if it is univariate it is dense whereas if it is multivariate it is sparse.

Optional arguments

Other arguments can optionally be specified, in any order, after the first compulsory variable argument. All arguments receive full algebraic evaluation, subject to the current switch settings etc. The arguments are processed in the order given, so that if more than one argument relates to the same property then the last one specified takes effect. Optional arguments are either keywords or equations with keywords on the left.

In general, the polynomial is sparse by default, unless the keyword `dense` is specified as an optional argument. (The keyword `sparse` is also accepted, but is the default.) The default degree can be changed by specifying an optional argument of the form

`degree = natural number.`

In the multivariate case this is the total degree, i.e. the sum of the degrees with

⁴²If it is a single expression then the univariate code is invoked; if it is a list then the multivariate code is invoked, and in the special case of a list of one element the multivariate code is invoked to generate a univariate polynomial, but the result should be indistinguishable from that resulting from specifying a single expression not in a list.

respect to the individual variables. The keywords `deg` and `maxdeg` can also be used in place of `degree`. More complicated monomial degree bounds can be constructed by using the coefficient function described below to return a monomial or polynomial coefficient expression. Moreover, `randpoly` respects internally the REDUCE “asymptotic” commands `let`, `weight` etc. described in §10.4 of the REDUCE 3.6 manual, which can be used to exercise additional control over the polynomial generated.

In the sparse case (only), the default maximum number of terms generated can be changed by specifying an optional argument of the form

$$\text{terms} = \textit{natural number}.$$

The actual number of terms generated will be the minimum of the value of `terms` and the number of terms in a dense polynomial of the specified degree, number of variables, etc.

16.44.3 Advanced use of `randpoly`

The default order (or minimum or trailing degree) can be changed by specifying an optional argument of the form

$$\text{ord} = \textit{natural number}.$$

The keyword is `ord` rather than `order` because `order` is a reserved command name in REDUCE. The keyword `mindeg` can also be used in place of `ord`. In the multivariate case this is the total degree, i.e. the sum of the degrees with respect to the individual variables. The order normally defaults to 0.

However, the input expressions to `randpoly` can also be equations, in which case the order defaults to 1 rather than 0. Input equations are converted to the difference of their two sides before being substituted into the generated polynomial. The purpose of this facility is to easily generate polynomials with a specified zero – for example

$$\text{randpoly}(x = a);$$

generates a polynomial that is guaranteed to vanish at $x = a$, but is otherwise random.

Order specification and equation input are extensions of the current Maple version of `randpoly`.

The operator `randpoly` accepts two further optional arguments in the form of equations with the keywords `coeffs` and `expons` on the left. The right sides of each of these equations must evaluate to objects that can be applied as functions of

no variables. These functions should be normal algebraic procedures (or something equivalent); the `coeffs` procedure may return any algebraic expression, but the `expons` procedure must return an integer (otherwise `randpoly` reports an error). The values returned by the functions should normally be random, because it is the randomness of the coefficients and, in the sparse case, of the exponents that makes the constructed polynomial random.

A convenient special case is to use the function `rand` on the right of one or both of these equations; when called with a single argument `rand` returns an anonymous function of no variables that generates a random integer. The single argument of `rand` should normally be an integer range in the form $a .. b$, where a, b are integers such that $a < b$. The spaces around (or at least before) the infix operator “`..`” are necessary in some cases in REDUCE and generally recommended. For example, the `expons` argument might take the form

```
expons = rand(0 .. n)
```

where n will be the maximum degree with respect to each variable *independently*. In the case of `coeffs` the lower limit will often be the negative of the upper limit to give a balanced coefficient range, so that the `coeffs` argument might take the form

```
coeffs = rand(-n .. n)
```

which will generate random integer coefficients in the range $[-n, n]$.

16.44.4 Subsidiary functions: `rand`, `proc`, `random`

Rand: a random-number-generator generator

The first argument of `rand` must be either an integer range in the form $a .. b$, where a, b are integers such that $a < b$, or a positive integer n which is equivalent to the range $0 .. n - 1$. The operator `rand` constructs a function of no arguments that calls the REDUCE random number generator function `random` to return a random integer in the range specified; in the case that the first argument of `rand` is a single positive integer n the function constructed just calls `random(n)`, otherwise the call of `random` is scaled and shifted.

As an additional convenience, if `rand` is called with a second argument that is an identifier then the call of `rand` acts exactly like a procedure definition with the identifier as the procedure name. The procedure generated can then be called with an empty argument list by the algebraic processor.

[Note that `rand()` with no argument is an error in REDUCE and does not return directly a random number in a default range as it does in Maple – use instead the

REDUCE function `random` (see below).]

Proc: an anonymous procedure generator

The operator `proc` provides a generalization of `rand`, and is primarily intended to be used with expressions involving the `random` function (see below). Essentially, it provides a mechanism to prevent functions such as `random` being evaluated when the arguments to `randpoly` are evaluated, which is too early. `Proc` accepts a single argument which is converted into the body of an anonymous procedure, which is returned as the value of `proc`. (If a named procedure is required then the normal REDUCE `procedure` statement should be used instead.) Examples are given in the following sections, and in the file `randpoly.tst`.

Random: a generalized interface

As an additional convenience, this package extends the interface to the standard REDUCE `random` function so that it will directly accept either a natural number or an integer range as its argument, exactly as for the first argument of `rand`. Hence effectively

$$\text{rand}(X) = \text{proc } \text{random}(X)$$

although `rand` is marginally more efficient. However, `proc` and the generalized `random` interface allow expressions such as the following anonymous random fraction generator to be easily constructed:

```
proc(random(-99 .. 99)/random(1 .. 99))
```

Further support for procs

`Rand` is a special case of `proc`, and (for either) if the switch `comp` is on (and the compiler is available) then the generated procedure body is compiled.

`Rand` with a single argument and `proc` both return as their values anonymous procedures, which if they are not compiled are Lisp lambda expressions. However, if compilation is in effect then they return only an identifier that has no external significance⁴³ but which can be applied as a function in the same way as a lambda expression.

It is primarily intended that such “proc expressions” will be used immediately as input to `randpoly`. The algebraic processor is not intended to handle lambda expressions. However, they can be output or assigned to variables in algebraic mode,

⁴³It is not interned on the oblist.

although the output form looks a little strange and is probably best not displayed. But beware that lambda expressions cannot be evaluated by the algebraic processor (at least, not without declaring some internal Lisp functions to be algebraic operators). Therefore, for testing purposes or curious users, this package provides the operators `showproc` and `evalproc` respectively to display and evaluate “proc expressions” output by `rand` or `proc` (or in fact any lambda expression), in the case of `showproc` provided they are not compiled.

16.44.5 Examples

The file `randpoly.tst` gives a set of test and demonstration examples.

The following additional examples were taken from the Maple `randpoly` help file and converted to REDUCE syntax by replacing `[]` by `{ }` and making the other changes shown explicitly:

```
randpoly(x);
```

$$- 54x^5 - 92x^4 - 30x^3 + 73x^2 - 69x - 67$$

```
randpoly({x, y}, terms = 20);
```

$$\begin{aligned} & 31x^5 - 17x^4y - 48x^4 - 15x^3y^2 + 80x^3y + 92x^3 \\ & + 86x^2y^3 + 2x^2y^2 - 44x^2 + 83x^4y + 85x^3y^2 + 55x^2y^2 \\ & - 27x^5y + 33x^5 - 98xy^5 + 51y^4 - 2y^3 + 70y^2 - 60y - 10 \end{aligned}$$

```
randpoly({x, sin(x), cos(x)});
```

$$\begin{aligned} & \sin(x) * (- 4\cos(x)^4 - 85\cos(x)^3x + 50\sin(x)^3 \\ & - 20\sin(x)^2x + 76\sin(x)x + 96\sin(x)) \end{aligned}$$

```
% randpoly(z, expons = rand(-5..5)); % Maple
```

```
% A generalized random "polynomial"!
% Note that spaces are needed around .. in REDUCE.
on div; off allfac;
randpoly(z, expons = rand(-5 .. 5));
```

$$- 39z^4 + 14z^3 - 77z^{-3} - 37z^{-4} - 8z^{-5}$$

```
off div; on allfac;
% randpoly([x], coeffs = proc() randpoly(y) end); % Maple
randpoly({x}, coeffs = proc randpoly(y));
```

$$\begin{aligned} & 95x^5y^5 - 53x^5y^4 - 78x^5y^3 + 69x^5y^2 + 58x^5y - 58x^5 \\ & + 64x^4y^5 + 93x^4y^4 - 21x^4y^3 + 24x^4y^2 - 13x^4y \\ & - 28x^4y - 57x^3y^5 - 78x^3y^4 - 44x^3y^3 + 37x^3y^2 \\ & - 64x^3y - 95x^3y - 71x^2y^5 - 69x^2y^4 - x^2y^3 - 49x^2y^2 \\ & + 77x^2y + 48x^2y + 38x^5y + 93x^4y - 65x^3y - 83x^2y \\ & + 25x^5y + 51x^4y + 35x^3y - 18x^4y - 59x^3y + 73x^2y - y + 31 \end{aligned}$$

```
% A more conventional alternative is ...
% procedure r; randpoly(y)$ randpoly({x}, coeffs = r);
% or, in fact, equivalently ...
% randpoly({x}, coeffs = procedure r; randpoly(y));

randpoly({x, y}, dense);
```

$$\begin{aligned} & 85x^5 + 43x^4y + 68x^4y + 87x^3y^2 - 93x^3y - 20x^3 \\ & - 74x^2y^2 - 29x^2y^2 + 7x^2y^2 + 10x^4y + 62x^3y - 86x^2y \end{aligned}$$

$$+ 15*x*y - 97*x - 53*y + 71*y^5 - 46*y^4 - 28*y^3 + 79*y^2 + 44$$

16.44.6 Appendix: Algorithmic background

The only part of this package that involves any mathematics that is not completely trivial is the procedure to generate a sparse set of monomials of specified maximum and minimum total degrees in a specified set of variables. This involves some combinatorics, and the Maple implementation calls some procedures from the Maple Combinatorial Functions Package `combinat` (of which I have implemented restricted versions in `REDUCE`).

Given the maximum possible number N of terms (in a dense polynomial), the required number of terms (in the sparse polynomial) is selected as a random subset of the natural numbers up to N , where each number indexes a term. In the univariate case these indices are used directly as monomial exponents, but in the multivariate case they are converted to monomial exponent vectors using a lexicographic ordering.

Numbers of polynomial terms

By explicitly enumerating cases with 1, 2, etc. variables, as indicated by the inductive proof below, one deduces that:

Proposition 1 *In n variables, the number of distinct monomials having total degree precisely r is ${}^{r+n-1}C_{n-1}$, and the maximum number of distinct monomials in a polynomial of maximum total degree d is ${}^{d+n}C_n$.*

Proof Suppose the first part of the proposition is true, namely that there are at most

$$N_h(n, r) = {}^{r+n-1}C_{n-1}$$

distinct monomials in an n -variable *homogeneous* polynomial of total degree r . Then there are at most

$$N(d, r) = \sum_{r=0}^d {}^{r+n-1}C_{n-1} = {}^{d+n}C_n$$

distinct monomials in an n -variable polynomial of maximum total degree d .

The sum follows from the fact that

$${}^{r+n}C_n = \frac{(r+n)^n}{n!}$$

where $x^n = x(x-1)(x-2)\cdots(x-n+1)$ denotes a falling factorial, and

$$\sum_{a \leq x < b} x^n = \frac{x^{n+1}}{n+1} \Big|_a^b.$$

(See, for example, D. H. Greene & D. E. Knuth, *Mathematics for the Analysis of Algorithms*, Birkhäuser, Second Edn. 1982, equation (1.37)). Hence the second part of the proposition follows from the first.

The proposition holds for 1 variable ($n = 1$), because there is clearly 1 distinct monomial of each degree precisely r and hence at most $d+1$ distinct monomials in a polynomial of maximum degree d .

Suppose that the proposition holds for n variables, which are represented by the vector X . Then a homogeneous polynomial of degree r in the $n+1$ variables X together with the single variable x has the form

$$x^r P_0(X) + x^{r-1} P_1(X) + \cdots + x^0 P_r(X)$$

where $P_s(X)$ represents a polynomial of maximum total degree s in the n variables X , which therefore contains at most $s+n C_n$ distinct monomials. The homogeneous polynomial of degree r in $n+1$ terms therefore contains at most

$$\sum_{s=0}^r s+n C_n = r+n+1 C_{n+1}$$

distinct monomials. Hence the proposition holds for $n+1$ variables, and therefore by induction it holds for all n . \square

Mapping indices to exponent vectors

The previous proposition is also the basis of the algorithm to map term indices $m \in \mathbb{N}$ to exponent vectors $v \in \mathbb{N}^n$, where n is the number of variables.

Define a norm $\|\cdot\|$ on exponent vectors by $\|v\| = \sum_{i=1}^n v_i$, which corresponds to the total degree of the monomial. Then, from the previous proposition, the number of exponent vectors of length n with norm $\|v\| \leq d$ is $N(n, d) = d+n C_n$. The elements of the m^{th} exponent vector are constructed recursively by applying the algorithm to successive tail vectors, so let a subscript denote the length of the vector to which a symbol refers.

The aim is to compute the vector of length n with index $m = m_n$. If this vector has norm d_n then the index and norm must satisfy

$$N(n, d_n - 1) \leq m_n < N(n, d_n),$$

which can be used (as explained below) to compute d_n given n and m_n . Since there are $N(n, d_n - 1)$ vectors with norm less than d_n , the index of the $(n - 1)$ -element tail vector must be given by $m_{n-1} = m_n - N(n, d_n - 1)$, which can be used recursively to compute the norm d_{n-1} of the tail vector. From this, the first element of the exponent vector is given by $v_1 = d_n - d_{n-1}$.

The algorithm therefore has a natural recursive structure that computes the norm of each tail subvector as the recursion stack is built up, but can only compute the first term of each tail subvector as the recursion stack is unwound. Hence, it constructs the exponent vector from right to left, whilst being applied to the elements from left to right. The recursion is terminated by the observation that $v_1 = d_1 = m_1$ for an exponent vector of length $n = 1$.

The main sub-procedure, given the required length n and index m_n of an exponent vector, must return its norm d_n and the index of its tail subvector of length $n - 1$. Within this procedure, $N(n, d)$ can be efficiently computed for values of d increasing from 0, for which $N(n, 0) = {}^nC_n = 1$, until $N(n, d) > m$ by using the observation that

$$N(n, d) = {}^{d+n}C_n = \frac{(d+n)(d-1+n) \cdots (1+n)}{d!}.$$

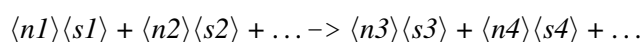
16.45 REACTEQN: Support for chemical reaction equation systems

This package allows a user to transform chemical reaction systems into ordinary differential equation systems (ODE) corresponding to the laws of pure mass action.

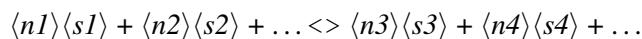
Documentation for this package is in plain text.

Author: Herbert Melenk.

A single reaction equation is an expression of the form



or



where the $\langle si \rangle$ are arbitrary names of species (REDUCE symbols) and the $\langle ni \rangle$ are positive integer numbers. The number 1 can be omitted. The connector \rightarrow describes a one way reaction, while $<>$ describes a forward and backward reaction.

A reaction system is a list of reaction equations, each of them optionally followed by one or two expressions for the rate constants. A rate constant can be a number, a symbol or an arbitrary REDUCE expression. If a rate constant is missing, an automatic constant of the form RATE(n) (where n is an integer counter) is generated. For double reactions the first constant is used for the forward direction, the second one for the backward direction.

The names of the species are collected in a list bound to the REDUCE variable SPECIES. This list is automatically filled during the processing of a reaction system. The species enter in an order corresponding to their appearance in the reaction system and the resulting ode's will be ordered in the same manner.

If a list of species is preassigned to the variable SPECIES either explicitly or from previous operations, the given order will be maintained and will dominate the formatting process. So the ordering of the result can be easily influenced by the user.

Syntax:

```
reac2ode {  $\langle reaction \rangle$  [,  $\langle rate \rangle$  [,  $\langle rate \rangle$ ]] [,  $\langle reaction \rangle$  [,  $\langle rate \rangle$  [,  $\langle rate \rangle$ ]]] .... };
```

where two rates are applicable only for $<>$ reactions.

Result is a system of explicit ordinary differential equations with polynomial right-hand sides. As side effect the following variables are set:

lists:

rates: list of the rates in the system

species: list of the species in the system

matrices:

inputmat: matrix of the input coefficients

outputmat: matrix of the output coefficients

In the matrices the row number corresponds to the input reaction number, while the column number corresponds to the species index. Note: if the rates are numerical values, it will be in most cases appropriate to switch on REDUCE rounded mode for floating point numbers. That is

on rounded;

Inputmat and outputmat can be used for linear algebra type investigations of the reaction system. The classical reaction matrix is the difference of these matrices; however, the two matrices contain more information than their differences because the appearance of a species on both sides is not reflected by the reaction matrix.

EXAMPLES: This input

```
% Example taken from Feinberg (Chemical Engineering):
```

```
species := {A1,A2,A3,A4,A5};

reac2ode { A1 + A4 <> 2A1, rho, beta,
           A1 + A2 <> A3, gamma, epsilon,
           A3      <> A2 + A5, theta, mue};
```

gives the output

```

                                2
{DF(A1,T)=RHO*A1*A4 - BETA*A1 - GAMMA*A1*A2 + EPSILON*A3,

DF(A2,T)= - GAMMA*A1*A2 + EPSILON*A3 + THETA*A3 - MUE*A2*A5,

DF(A3,T)=GAMMA*A1*A2 - EPSILON*A3 - THETA*A3 + MUE*A2*A5,

                                2
DF(A4,T)= - RHO*A1*A4 + BETA*A1 ,

DF(A5,T)=THETA*A3 - MUE*A2*A5}
```

The corresponding matrices are

```
inputmat;
```

```

[ 1 0 0 1 0 ]
[           ]
[ 1 1 0 0 0 ]
[           ]
[ 0 0 1 0 0 ]

outputmat;

[ 2 0 0 0 0 ]
[           ]
[ 0 0 1 0 0 ]
[           ]
[ 0 1 0 0 1 ]

% computation of the classical reaction matrix as difference
% of output and input matrix:

reactmat := outputmat-inputmat;

      [ 1  0  0  -1  0 ]
      [           ]
REACTMAT := [ -1 -1  1  0  0 ]
      [           ]
      [ 0  1  -1  0  1 ]

% Example with automatic generation of rate constants
% and automatic extraction of species

species := {};

reac2ode { A1 + A4 <> 2A1,
          A1 + A2 <> A3,
          a3 <> A2 + A5};

new species: A1
new species: A4
new species: A3
new species: A2
new species: A5

2
{DF(A1,T)= - A1 *RATE(2) + A1*A4*RATE(1) - A1*A2*RATE(3) +

```

```

A3*RATE(4),


$$DF(A4, T) = A1^2 * RATE(2) - A1 * A4 * RATE(1),$$

DF(A2, T) = - A1*A2*RATE(3) - A2*A5*RATE(6) + A3*RATE(5) + A3*RATE(4),
DF(A3, T) = A1*A2*RATE(3) + A2*A5*RATE(6) - A3*RATE(5) - A3*RATE(4),
DF(A5, T) = - A2*A5*RATE(6) + A3*RATE(5)\}

% Example with rates computed from numerical expressions

species := {};

reac2ode { A1 + A4 <> 2A1, 17.3* 22.4^1.5,
          0.04* 22.4^1.5 };

new species: A1
new species: A4


$$\{DF(A1, T) = - 4.24065 * A1^2 + 1834.08 * A1 * A4,$$



$$DF(A4, T) = 4.24065 * A1^2 - 1834.08 * A1 * A4\}$$


```

16.46 REDLOG: Extend REDUCE to a computer logic system

The name REDLOG stand for REDuce LOGic system. Redlog implements symbolic algorithms on first-order formulas with respect to user-chosen first-order languages and theories. The available domains include real numbers, integers, complex numbers, p-adic numbers, quantified propositional calculus, term algebras.

Documentation for this package can be found [online](#).

Authors: Andreas Dolzmann and Thomas Sturm

16.47 RESET: Code to reset REDUCE to its initial state

This package defines a command RESETREDUCE that works through the history of previous commands, and clears any values which have been assigned, plus any rules, arrays and the like. It also sets the various switches to their initial values. It is not complete, but does work for most things that cause a gradual loss of space. It would be relatively easy to make it interactive, so allowing for selective resetting.

There is no further documentation on this package.

Author: John Fitch.

16.48 RESIDUE: A residue package

This package supports the calculation of residues of arbitrary expressions.

Author: Wolfram Koepf.

This package supports the calculation of residues. The residue $\text{Res}_{z=a} f(z)$ of a function $f(z)$ at the point $a \in \mathbb{C}$ is defined as

$$\text{Res}_{z=a} f(z) = \frac{1}{2\pi i} \oint f(z) dz ,$$

with integration along a closed curve around $z = a$ with winding number 1.

If $f(z)$ is given by a Laurent series development at $z = a$

$$f(z) = \sum_{k=-\infty}^{\infty} a_k (z - a)^k ,$$

then

$$\text{Res}_{z=a} f(z) = a_{-1} . \quad (16.65)$$

If $a = \infty$, one defines on the other hand

$$\text{Res}_{z=\infty} f(z) = -a_{-1} \quad (16.66)$$

for given Laurent representation

$$f(z) = \sum_{k=-\infty}^{\infty} a_k \frac{1}{z^k} .$$

The package is loaded by the statement

```
1: load residue;
```

It contains two REDUCE operators:

- `residue(f, z, a)` determines the residue of f at the point $z = a$ if f is meromorphic at $z = a$. The calculation of residues at essential singularities of f is not supported.
- `poleorder(f, z, a)` determines the pole order of f at the point $z = a$ if f is meromorphic at $z = a$.

Note that both functions use the `taylor` package in connection with representations (16.65)–(16.66).

Here are some examples:


```
2: residue(x/(x^2-2),x,sqrt(2));
```

$$\frac{1}{2}$$

```
3: poleorder(x/(x^2-2),x,sqrt(2));
```

```
1
```

```
4: residue(sin(x)/(x^2-2),x,sqrt(2));
```

$$\frac{\sqrt{2} \sin(\sqrt{2})}{4}$$

```
5: poleorder(sin(x)/(x^2-2),x,sqrt(2));
```

```
1
```

```
6: residue(1/(x-1)^m/(x-2)^2,x,2);
```

$$-m$$

```
7: poleorder(1/(x-1)/(x-2)^2,x,2);
```

```
2
```

```
8: residue(sin(x)/x^2,x,0);
```

```
1
```

```
9: poleorder(sin(x)/x^2,x,0);
```

```
1
```

```
10: residue((1+x^2)/(1-x^2),x,1);
```

```
-1
```

```
11: poleorder((1+x^2)/(1-x^2),x,1);
```

```
1
```

```
12: residue((1+x^2)/(1-x^2), x, -1);
```

```
1
```

```
13: poleorder((1+x^2)/(1-x^2), x, -1);
```

```
1
```

```
14: residue(tan(x), x, pi/2);
```

```
-1
```

```
15: poleorder(tan(x), x, pi/2);
```

```
1
```

```
16: residue((x^n-y^n)/(x-y), x, y);
```

```
0
```

```
17: poleorder((x^n-y^n)/(x-y), x, y);
```

```
0
```

```
18: residue((x^n-y^n)/(x-y)^2, x, y);
```

$$\frac{y^n}{y}$$

```
19: poleorder((x^n-y^n)/(x-y)^2, x, y);
```

```
1
```

```
20: residue(tan(x)/sec(x-pi/2)+1/cos(x), x, pi/2);
```

```
-2
```

```
21: poleorder(tan(x)/sec(x-pi/2)+1/cos(x), x, pi/2);
```

```
1
```

```
22: for k:=1:2 sum residue((a+b*x+c*x^2)/(d+e*x+f*x^2), x,
```

```
part(part(solve(d+e*x+f*x^2,x),k),2));
```

$$\frac{b*f - c*e}{f^2}$$

```
23: residue(x^3/sin(1/x)^2,x,infinity);
```

$$-\frac{1}{15}$$

```
24: residue(x^3*sin(1/x)^2,x,infinity);
```

$$-1$$

Note that the residues of factorial and Γ function terms are not yet supported.

16.49 RLFI: REDUCE \LaTeX formula interface

This package adds \LaTeX syntax to REDUCE. Text generated by REDUCE in this mode can be directly used in \LaTeX source documents. Various mathematical constructions are supported by the interface including subscripts, superscripts, font changing, Greek letters, divide-bars, integral and sum signs, derivatives, and so on.

Author: Richard Liska.

High quality typesetting of mathematical formulas is a quite tedious task. One of the most sophisticated typesetting programs for mathematical text \TeX [?], together with its widely used macro package \LaTeX [?], has a strange syntax of mathematical formulas, especially of the complicated type. This is the main reason which lead us to designing the formula interface between the computer algebra system REDUCE and the document preparation system \LaTeX . The other reason is that all available syntaxes of the REDUCE formula output are line oriented and thus not suitable for typesetting in mathematical text. The idea of interfacing a computer algebra system to a typesetting program has already been used, eg. in [?] presenting the \TeX output of the MACSYMA computer algebra system.

The formula interface presented here adds to REDUCE the new syntax of formula output, namely \LaTeX syntax, and can also be named REDUCE - \LaTeX translator. Text generated by REDUCE in this syntax can be directly used in \LaTeX source documents. Various mathematical constructions are supported by the interface including subscripts, superscripts, font changing, Greek letters, divide-bars, integral and sum signs, derivatives etc.

The interface can be used in two ways:

- for typesetting of results of REDUCE algebraic calculations.
- for typesetting of users formulas.

The latter can even be used by users unfamiliar with the REDUCE system, because the REDUCE input syntax of formulas is almost the same as the syntax of the majority of programming languages. We aimed at speeding up the process of formula typesetting, because we are convinced, that the writing of correct complicated formulas in the REDUCE syntax is a much more simpler task than writing them in the \LaTeX syntax full of keywords and special characters `\`, `{`, `^` etc. It is clear, that not every formula produced by the interface is typeset in the best format from an aesthetic point of view. When a user is not satisfied with the result, he can add some \LaTeX commands to the REDUCE output - \LaTeX input.

The interface is connected to REDUCE by three new switches and several statements. To activate the \LaTeX output mode the switch `latex` must be set on. this switch, similar to the switch `fort` producing FORTRAN output, being on causes all outputs to be written in the \LaTeX syntax of formulas. The switch `VERBATIM`

is used for input printing control. If it is `on` input to REDUCE system is typeset in \LaTeX verbatim environment after the line containing the string `REDUCE Input :`.

The switch `lasimp` controls the algebraic evaluation of input formulas. If it is `on` every formula is evaluated, simplified and written in the form given by ordinary REDUCE statements and switches such as `factor`, `order`, `rat` etc. In the case when the `lasimp` switch is `off` evaluation, simplification or reordering of formulas is not performed and REDUCE acts only as a formula parser and the form of the formula output is exactly the same as that of the input, the only difference remains in the syntax. The mode `off lasimp` is designed especially for typesetting of formulas for which the user needs preservation of their structure. This switch has no meaning if the switch `Latex` is `off` and thus is working only for \LaTeX output.

For every identifier used in the typeset REDUCE formula the following properties can be defined by the statement `defid`:

- its printing symbol (Greek letters can be used).
- the font in which the symbol will be typeset.
- accent which will be typeset above the symbol.

Symbols with indexes are treated in REDUCE as operators. Each index corresponds to an argument of the operator. The meaning of operator arguments (where one wants to typeset them) is declared by the statement `defindex`. This statement causes the arguments to be typeset as subscripts or superscripts (on left or right-hand side of the operator) or as arguments of the operator.

The statement `mathstyle` defines the style of formula typesetting. The variable `laline!*` defines the length of output lines.

The fractions with horizontal divide bars are typeset by using the new REDUCE infix operator `\`. This operator is not algebraically simplified. During typesetting of powers the checking on the form of the power base and exponent is performed to determine the form of the typeset expression (eg. `sqrt` symbol, using parentheses).

Some special forms can be typeset by using REDUCE prefix operators. These are as follows:

- `int` - integral of an expression.
- `dint` - definite integral of an expression.
- `df` - derivative of an expression.
- `pdf` - partial derivative of an expression.
- `sum` - sum of expressions.

- `product` - product of expressions.
- `sqrt` - square root of expression.

There are still some problems unsolved in the present version of the interface as follows:

- breaking the formulas which do not fit on one line.
- automatic decision where to use divide bars in fractions.
- distinction of two- or more-character identifiers from the product of one-character symbols.
- typesetting of matrices.

Description of files

rlfi.red - REDUCE source file for this interface.

rlfi.tex - this document.

rlfi.bib - bibliography file for this document.

rlfi.tst - test file for this interface.

rlfi.log - \LaTeX output of the test session, can be directly used as \LaTeX input file.

Remark

After finishing presented interface, we have found another work [?], which solves the same problem. The RLFI package has been described in [?] too.

16.49.1 APPENDIX: Summary and syntax

Warning

The RLFI package can be used only on systems supporting lower case letters with `off raise` statement. The package distinguishes the upper and lower case letters, so be carefull in typing them. In REDUCE3.6 the REDUCEcommands have to be typed in lower-case while the switch `latex` is on, in previous versions the commands had to be typed in upper-case.

Switches

`latex` - If on output is in \LaTeX format. It turns off the `raise` switch if it is set on and on the `raise` switch if it is set off. By default is off.

lasimp - If on formulas are evaluated (simplified), REDUCE works as usually.
 If off no evaluation is performed and the structure of formulas is preserved.
 By default is on.

verbatim - If on the REDUCE input, while latex switch being on, is printed in \LaTeX verbatim environment. The actual REDUCE input is printed after the line containing the string "REDUCE Input : ". It turns on resp. off the echo switch when turned on resp. off. by default is off.

Operators

infix - \

prefix - int, dint, df, pdf, sum, product, sqrt and all REDUCE prefix operators defined in the REDUCE kernel and the SOLVE module.

```
<alg. expression> \ <alg. expression>
int(<function>, <variable>)
dint(<from>, <to>, <function>, <variable>)
df(<function>, <variables>)
<variables> ::= <o-variable> | <o-variable>, <variables>
<o-variable> ::= <variable> | <variable>, <order>
<variable> ::= <kernel>
<order> ::= <integer>
<function> ::= <alg. expression>
<from> ::= <alg. expression>
<to> ::= <alg. expression>
pdf(<function>, <variables>)
sum(<from>, <to>, <function>)
product(<from>, <to>, <function>)
sqrt(<alg. expression>)
```

<alg. expression> is any algebraic expression. Where appropriate, it can include also relational operators (e.g. argument <from> of sum or product operators is usually equation). <kernel> is identifier or prefix operator with arguments as described in [?]. Interface supports typesetting lists of algebraic expressions.

Statements

```
mathstyle <m-style>;
<m-style> ::= math | displaymath | equation
defid <identifier>, <d-equations>;
<d-equations> ::= <d-equation> | <d-equation>, <d-equations>
```

```

<d-equation> ::= <d-print symbol> | <d-font>|<d-accent>
<d-print symbol> ::= name = <print symbol>
<d-font> ::= font = <font>
<d-accent> ::= accent = <accent>
<print symbol> ::= <character> | <special symbol>
<special symbol> ::= alpha|beta|gamma|delta|epsilon|
    varepsilon|zeta|eta|theta|vartheta|iota|kappa|lambda|
    mu|nu|xi|pi|varpi|rho|varrho|sigma|varsigma|tau|
    upsilon|phi|varphi|chi|psi|omega|Gamma|Delta|Theta|
    Lambda|Xi|Pi|Sigma|Upsilon|Phi|Psi|Omega|infty|hbar
<font> ::= bold|roman
<accent> ::=hat|check|breve|acute|grave|tilde|bar|vec|
    dot|ddot

```

For special symbols and accents see [?], p. 43, 45, 51.

```

defindex <d-operators>;
<d-operators> ::= <d-operator> | <d-operator>,<d-operators>
<d-operator> ::= <prefix operator>(<descriptions>)
<prefix operator> ::= <identifier>
<descriptions> ::= <description> | <description>,<
    <descriptions>
<description> ::= arg | up | down | leftup | leftdown

```

The meaning of the statements is briefly described in the preceding text.

16.50 ROOTS: A REDUCE root finding package

This root finding package can be used to find some or all of the roots of a univariate polynomial with real or complex coefficients, to the accuracy specified by the user.

It is designed so that it can be used as an independent package, or it may be called from SOLVE if ROUNDED is on. For example, the evaluation of

```
on rounded, complex;
solve (x**3+x+5, x) ;
```

yields the result

```
{X= - 1.51598, X=0.75799 + 1.65035*I, X=0.75799 - 1.65035*I}
```

This package loads automatically.

Author: Stanley L. Kameny.

16.50.1 Introduction

The root finding package is designed so that it can be used as an independent package, or it can be integrated with and called by SOLVE. This document describes the package in its independent use. It can be used to find some or all of the roots of univariate polynomials with real or complex coefficients, to the accuracy specified by the user.

16.50.2 Root Finding Strategies

For all polynomials handled by the root finding package, strategies of factoring are employed where possible to reduce the amount of required work. These include square-free factoring and separation of complex polynomials into a product of a polynomial with real coefficients and one with complex coefficients. Whenever these succeed, the resulting smaller polynomials are solved separately, except that the root accuracy takes into account the possibility of close roots on different branches. One other strategy used where applicable is the powergcd method of reducing the powers of the initial polynomial by a common factor, and deriving the roots in two stages, as roots of the reduced power polynomial. Again here, the possibility of close roots on different branches is taken into account.

16.50.3 Top Level Functions

The top level functions can be called either as symbolic operators from algebraic mode, or they can be called directly from symbolic mode with symbolic mode arguments. Outputs are expressed in forms that print out correctly in algebraic mode.

Functions that refer to real roots only

Three top level functions refer only to real roots. Each of these functions can receive 1, 2 or 3 arguments.

The first argument is the polynomial p , that can be complex and can have multiple or zero roots. If arg2 and arg3 are not present, all real roots are found. If the additional arguments are present, they restrict the region of consideration.

- If arguments are $(p, \text{arg2})$ then Arg2 must be POSITIVE or NEGATIVE. If $\text{arg2}=\text{NEGATIVE}$ then only negative roots of p are included; if $\text{arg2}=\text{POSITIVE}$ then only positive roots of p are included. Zero roots are excluded.
- If arguments are $(p, \text{arg2}, \text{arg3})$ then Arg2 and Arg3 must be r (a real number) or EXCLUDE r , or a member of the list POSITIVE, NEGATIVE, INFINITY, -INFINITY. EXCLUDE r causes the value r to be excluded from the region. The order of the sequence $\text{arg2}, \text{arg3}$ is unimportant. Assuming that $\text{arg2} \leq \text{arg3}$ when both are numeric, then

$\{-\text{INFINITY}, \text{INFINITY}\}$	is equivalent to	$\{\}$	represents all roots;
$\{\text{arg2}, \text{NEGATIVE}\}$	represents	$-\infty < r < \text{arg2};$	
$\{\text{arg2}, \text{POSITIVE}\}$	represents	$\text{arg2} < r < \infty;$	

In each of the following, replacing an arg with EXCLUDE arg converts the corresponding inclusive \leq to the exclusive $<$

$\{\text{arg2}, -\text{INFINITY}\}$	represents	$-\infty < r \leq \text{arg2};$
$\{\text{arg2}, \text{INFINITY}\}$	represents	$\text{arg2} \leq r < \infty;$
$\{\text{arg2}, \text{arg3}\}$	represents	$\text{arg2} \leq r \leq \text{arg3};$

- If zero is in the interval the zero root is included.

REALROOTS This function finds the real roots of the polynomial p , using the REALROOT package to isolate real roots by the method of Sturm sequences, then polishing the root to the desired accuracy. Precision of computation is guaranteed to be sufficient to separate all real roots in the specified region. (cf. MULTIROOT for treatment of multiple roots.)

ISOLATER This function produces a list of rational intervals, each containing a single real root of the polynomial p , within the specified region, but does not find the roots.

RLROOTNO This function computes the number of real roots of p in the specified region, but does not find the roots.

Functions that return both real and complex roots

ROOTS p ; This is the main top level function of the roots package. It will find all roots, real and complex, of the polynomial p to an accuracy that is sufficient to separate them and which is a minimum of 6 decimal places. The value returned by ROOTS is a list of equations for all roots. In addition, ROOTS stores separate lists of real roots and complex roots in the global variables ROOTSREAL and ROOTSCOMPLEX.

The order of root discovery by ROOTS is highly variable from system to system, depending upon very subtle arithmetic differences during the computation. In order to make it easier to compare results obtained on different computers, the output of ROOTS is sorted into a standard order: a root with smaller real part precedes a root with larger real part; roots with identical real parts are sorted so that larger imaginary part precedes smaller imaginary part. (This is done so that for complex pairs, the positive imaginary part is seen first.)

However, when a polynomial has been factored (by square-free factoring or by separation into real and complex factors) then the root sorting is applied to each factor separately. This makes the final resulting order less obvious. However it is consistent from system to system.

ROOTS_AT_PREC p ; Same as ROOTS except that roots values are returned to a minimum of the number of decimal places equal to the current system precision.

ROOT_VAL p ; Same as ROOTS_AT_PREC, except that instead of returning a list of equations for the roots, a list of the root value is returned. This is the function that SOLVE calls.

NEARESTROOT(p,s); This top level function uses an iterative method to find the root to which the method converges given the initial starting origin s , which can be complex. If there are several roots in the vicinity of s and s is not significantly closer to one root than it is to all others, the convergence could arrive at a root that is not truly the nearest root. This function should therefore be used only when the user is certain that there is only one root in the immediate vicinity of the starting point s .

FIRSTROOT p; ROOTS is called, but only the first root determined by ROOTS is computed. Note that this is not in general the first root that would be listed in ROOTS output, since the ROOTS outputs are sorted into a canonical order. Also, in some difficult root finding cases, the first root computed might be incorrect.

Other top level functions

GETROOT(n,rr); If rr has the form of the output of ROOTS, REALROOTS, or NEARESTROOTS; GETROOT returns the rational, real, or complex value of the root equation. An error occurs if $n < 1$ or $n >$ the number of roots in rr.

MKPOLY rr; This function can be used to reconstruct a polynomial whose root equation list is rr and whose denominator is 1. Thus one can verify that if $rr := \text{ROOTS } p$, and $rr1 := \text{ROOTS MKPOLY } rr$, then $rr1 = rr$. (This will be true if MULTIROOT and RATROOT are ON, and ROUNDED is off.) However, $\text{MKPOLY } rr - \text{NUM } p = 0$ will be true if and only if all roots of p have been computed exactly.

Functions available for diagnostic or instructional use only

GFNEWT(p,r,cpx); This function will do a single pass through the function GFNEWTON for polynomial p and root r. If cpx=T, then any complex part of the root will be kept, no matter how small.

GFROOT(p,r,cpx); This function will do a single pass through the function GFROOTFIND for polynomial p and root r. If cpx=T, then any complex part of the root will be kept, no matter how small.

16.50.4 Switches Used in Input

The input of polynomials in algebraic mode is sensitive to the switches COMPLEX, ROUNDED, and ADJPREC. The correct choice of input method is important since incorrect choices will result in undesirable truncation or rounding of the input coefficients.

Truncation or rounding may occur if ROUNDED is on and one of the following is true:

1. a coefficient is entered in floating point form or rational form.
2. COMPLEX is on and a coefficient is imaginary or complex.

Therefore, to avoid undesirable truncation or rounding, then:

1. `ROUNDED` should be off and input should be in integer or rational form; or
2. `ROUNDED` can be on if it is acceptable to truncate or round input to the current value of system precision; or both `ROUNDED` and `ADJPREC` can be on, in which case system precision will be adjusted to accommodate the largest coefficient which is input; or
3. if the input contains complex coefficients with very different magnitude for the real and imaginary parts, then all three switches `ROUNDED`, `ADJPREC` and `COMPLEX` must be on.

integer and complex modes (off `ROUNDED`) any real polynomial can be input using integer coefficients of any size; integer or rational coefficients can be used to input any real or complex polynomial, independent of the setting of the switch `COMPLEX`. These are the most versatile input modes, since any real or complex polynomial can be input exactly.

modes rounded and complex-rounded (on `ROUNDED`) polynomials can be input using integer coefficients of any size. Floating point coefficients will be truncated or rounded, to a size dependent upon the system. If complex is on, real coefficients can be input to any precision using integer form, but coefficients of imaginary parts of complex coefficients will be rounded or truncated.

16.50.5 Internal and Output Use of Switches

The `REDUCE` arithmetic mode switches `ROUNDED` and `COMPLEX` control the behavior of the root finding package. These switches are returned in the same state in which they were set initially, (barring catastrophic error).

COMPLEX The root finding package controls the switch `COMPLEX` internally, turning the switch on if it is processing a complex polynomial. For a polynomial with real coefficients, the starting point argument for `NEARESTROOT` can be given in algebraic mode in complex form as $rl + im * I$ and will be handled correctly, independent of the setting of the switch `COMPLEX`. Complex roots will be computed and printed correctly regardless of the setting of the switch `COMPLEX`. However, if `COMPLEX` is off, the imaginary part will print out ahead of the real part, while the reverse order will be obtained if `COMPLEX` is on.

ROUNDED The root finding package performs computations using the arithmetic mode that is required at the time, which may be integer, Gaussian integer,

rounded, or complex rounded. The switch `BFTAG` is used internally to govern the mode of computation and precision is adjusted whenever necessary. The initial position of switches `ROUNDED` and `COMPLEX` are ignored. At output, these switches will emerge in their initial positions.

16.50.6 Root Package Switches

Note: switches `AUTOMODE`, `ISOROOT` and `ACCROOT`, present in earlier versions, have been eliminated.

RATROOT (Default OFF) If `RATROOT` is on all root equations are output in rational form. Assuming that the mode is `COMPLEX` (i.e. `ROUNDED` is off,) the root equations are guaranteed to be able to be input into `REDUCE` without truncation or rounding errors. (Cf. the function `MKPOLY` described above.)

MULTIROOT (Default ON) Whenever the polynomial has complex coefficients or has real coefficients and has multiple roots, as determined by the Sturm function, the function `SQFREF` is called automatically to factor the polynomial into square-free factors. If `MULTIROOT` is on, the multiplicity of the roots will be indicated in the output of `ROOTS` or `REALROOTS` by printing the root output repeatedly, according to its multiplicity. If `MULTIROOT` is off, each root will be printed once, and all roots should be normally be distinct. (Two identical roots should not appear. If the initial precision of the computation or the accuracy of the output was insufficient to separate two closely-spaced roots, the program attempts to increase accuracy and/or precision if it detects equal roots. If, however, the initial accuracy specified was too low, and it was not possible to separate the roots, the program will abort.)

TRROOT (Default OFF) If switch `TRROOT` is on, trace messages are printed out during the course of root determination, to show the progress of solution.

ROOTMSG (Default OFF) If switch `ROOTMSG` is on in addition to switch `TRROOT`, additional messages are printed out to aid in following the progress of Laguerre and Newton complex iteration. These messages are intended for debugging use primarily.

16.50.7 Operational Parameters and Parameter Setting.

ROOTACC# (Default 6) This parameter can be set using the function `ROOTACC n`; which causes `ROOTACC#` to be set to `MAX(n,6)`. If `ACCROOT` is on, roots will be determined to a minimum of `ROOTACC#` significant places. (If roots are closely spaced, a higher number of significant places is computed where needed.)

system precision The roots package, during its operation, will change the value of system precision but will restore the original value of system precision at termination except that the value of system precision is increased if necessary to allow the full roots output to be printed.

PRECISION *n*; If the user sets system precision, using the command **PRECISION *n*;** then the effect is to increase the system precision to *n*, and to have the same effect on **ROOTS** as **ROOTACC *n*;** ie. roots will now be printed with minimum accuracy *n*. The original conditions can then be restored by using the command **PRECISION RESET;** or **PRECISION NIL;**.

ROOTPREC *n*; The roots package normally sets the computation mode and precision automatically. However, if **ROOTPREC *n*;** is called and *n* is greater than the initial system precision then all root computation will be done initially using a minimum system precision *n*. Automatic operation can be restored by input of **ROOTPREC 0;**.

16.50.8 Avoiding truncation of polynomials on input

The roots package will not internally truncate polynomials. However, it is possible that a polynomial can be truncated by input reading functions of the embedding lisp system, particularly when input is given in floating point (rounded) format.

To avoid any difficulties, input can be done in integer or Gaussian integer format, or mixed, with integers or rationals used to represent quantities of high precision. There are many examples of this in the test package. It is usually best to let the roots package determine the precision needed to compute roots.

The number of digits that can be safely represented in floating point in the lisp system are contained in the global variable `!!NFPD`. Similarly, the maximum number of significant figures in floating point output are contained in the global variable `!!FLIM`. The roots package computes these values, which are needed to control the logic of the program.

The values of intermediate root iterations (that are printed when **TRROOT** is on) are given in bigfloat format even when the actual values are computed in floating point. This avoids intrusive rounding of root printout.

16.51 RSOLVE: Rational/integer polynomial solvers

This package provides operators that compute the exact rational zeros of a single univariate polynomial using fast modular methods. The algorithm used is that described by R. Loos (1983): Computing rational zeros of integral polynomials by p -adic expansion, *SIAM J. Computing*, **12**, 286–293.

Author: Francis J. Wright.

This package provides the operators `r_solve` and `i_solve` that compute respectively the exact rational or integer zeros of a single univariate polynomial using fast modular methods.

16.51.1 Introduction

This package provides operators that compute the exact rational zeros of a single univariate polynomial using fast modular methods. The algorithm used is that described by R. Loos (1983): Computing rational zeros of integral polynomials by p -adic expansion, *SIAM J. Computing*, **12**, 286–293. The operator `r_solve` computes all rational zeros whereas the operator `i_solve` computes only integer zeros in a way that is slightly more efficient than extracting them from the rational zeros. The `r_solve` and `i_solve` interfaces are almost identical, and are intended to be completely compatible with that of the general `solve` operator, although `r_solve` and `i_solve` give more convenient output when only rational or integer zeros respectively are required. The current implementation appears to be faster than `solve` by a factor that depends on the example, but is typically up to about 2.

I plan to extend this package to compute Gaussian integer and rational zeros and zeros of polynomial systems.

16.51.2 The user interface

The first argument is required and must simplify to either a univariate polynomial expression or equation with integer, rational or rounded coefficients. Symbolic coefficients are not allowed (and currently complex coefficients are not allowed either.) The argument is simplified to a quotient of integer polynomials and the denominator is silently ignored.

Subsequent arguments are optional. If the polynomial variable is to be specified then it must be the first optional argument, and if the first optional argument is not a valid option (see below) then it is (mis-)interpreted as the polynomial variable. However, since the variable in a non-constant univariate polynomial can be

deduced from the polynomial it is unnecessary to specify it separately, except in the degenerate case that the first argument simplifies to either 0 or $0 = 0$. In this case the result is returned by `i_solve` in terms of the operator `arbint` and by `r_solve` in terms of the (new) analogous operator `arbrat`. The operator `i_solve` will generally run slightly faster than `r_solve`.

The (rational or integer) zeros of the first argument are returned as a list and the default output format is the same as that used by `solve`. Each distinct zero is returned in the form of an equation with the variable on the left and the multiplicities of the zeros are assigned to the variable `root_multiplicities` as a list. However, if the switch `multiplicities` is turned on then each zero is explicitly included in the solution list the appropriate number of times (and `root_multiplicities` has no value).

Optional keyword arguments acting as local switches allow other output formats. They have the following meanings:

separate: assign the multiplicity list to the global variable `root_multiplicities` (the default);

expand or multiplicities: expand the solution list to include multiple zeros multiple times (the default if the `| multiplicities—` switch is on);

together: return each solution as a list whose second element is the multiplicity;

nomul: do not compute multiplicities (thereby saving some time);

noeqs: do not return univariate zeros as equations but just as values.

16.51.3 Examples

```
r_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\left\{x = \frac{-4}{3}, x = 3, x = -3, x = \frac{4}{3}\right\}$$

```
i_solve((9x^2 - 16)*(x^2 - 9), x);
```

$$\{x = 3, x = -3\}$$

See the test/demonstration file `rsolve.tst` for more examples.

16.51.4 Tracing

The switch `trsolve` turns on tracing of the algorithm. It is off by default.

16.52 SCOPE: REDUCE source code optimization package

SCOPE is a package for the production of an optimized form of a set of expressions. It applies an heuristic search for common (sub)expressions to almost any set of proper REDUCE assignment statements. The output is obtained as a sequence of assignment statements. GENTRAN is used to facilitate expression output.

Author: J.A. van Hulzen.

16.53 SETS: A basic set theory package

The SETS package provides algebraic-mode support for set operations on lists regarded as sets (or representing explicit sets) and on implicit sets represented by identifiers.

Author: Francis J. Wright.

The SETS package for REDUCE3.5 and later versions provides algebraic-mode support for set operations on lists regarded as sets (or representing explicit sets) and on implicit sets represented by identifiers. It provides the set-valued infix operators (with synonyms) `union`, `intersection` (`intersect`) and `setdiff` (`\`, `minus`) and the Boolean-valued infix operators (predicates) `member`, `subset_eq`, `subset`, `set_eq`. The union and intersection operators are n-ary and the rest are binary. A list can be explicitly converted to the canonical set representation by applying the operator `mkset`. (The package also provides an operator not specifically related to set theory called `evalb` that allows the value of any Boolean-valued expression to be displayed in algebraic mode.)

16.53.1 Introduction

REDUCE has no specific representation for a set, neither in algebraic mode nor internally, and any object that is mathematically a set is represented in REDUCE as a list. The difference between a set and a list is that in a set the ordering of elements is not significant and duplicate elements are not allowed (or are ignored). Hence a list provides a perfectly natural and satisfactory representation for a set (but not vice versa). Some languages, such as Maple, provide different internal representations for sets and lists, which may allow sets to be processed more efficiently, but this is not *necessary*.

This package supports set theoretic operations on lists and represents the results as normal algebraic-mode lists, so that all other REDUCE facilities that apply to lists can still be applied to lists that have been constructed by explicit set operations. The algebraic-mode set operations provided by this package have all been available in symbolic mode for a long time, and indeed are used internally by the rest of REDUCE, so in that sense set theory facilities in REDUCE are far from new. What this package does is make them available in algebraic mode, generalize their operation by extending the arity of union and intersection, and allow their arguments to be implicit sets represented by unbound identifiers. It performs some simplifications on such symbolic set-valued expressions, but this is currently rather *ad hoc* and is probably incomplete.

For examples of the operation of the SETS package see (or run) the test file `sets.tst`. This package is experimental and developments are under consideration; if you have suggestions for improvements (or corrections) then please send

them to me (FJW), preferably by email. The package is intended to be run under REDUCE3.5 and later versions; it may well run correctly under earlier versions although I cannot provide support for such use.

16.53.2 Infix operator precedence

The set operators are currently inserted into the standard REDUCE precedence list (see page 28, §2.7, of the REDUCE 3.6 manual) as follows:

```
or and not member memq = set_eq neq eq >= > <= < subset_eq
subset freeof + - setdiff union intersection * / ^ .
```

16.53.3 Explicit set representation and `mkset`

Explicit sets are represented by lists, and this package does not require any restrictions at all on the forms of lists that are regarded as sets. Nevertheless, duplicate elements in a set correspond by definition to the same element and it is conventional and convenient to represent them by a single element, i.e. to remove any duplicate elements. I will call this a normal representation. Since the order of elements in a set is irrelevant it is also conventional and may be convenient to sort them into some standard order, and an appropriate ordering of a normal representation gives a canonical representation. This means that two identical sets have identical representations, and therefore the standard REDUCE equality predicate (`=`) correctly determines set equality; without a canonical representation this is not the case.

Pre-processing of explicit set-valued arguments of the set-valued operators to remove duplicates is always done because of the obvious efficiency advantage if there were any duplicates, and hence explicit sets appearing in the values of such operators will never contain any duplicate elements. Such sets are also currently sorted, mainly because the result looks better. The ordering used satisfies the `ordp` predicate used for most sorting within REDUCE, except that explicit integers are sorted into increasing numerical order rather than the decreasing order that satisfies `ordp`.

Hence explicit sets appearing in the result of any set operator are currently returned in a canonical form. Any explicit set can also be put into this form by applying the operator `mkset` to the list representing it. For example

```
mkset {1,2,y,x*y,x+y};
```

```
{x + y, x*y, y, 1, 2}
```

The empty set is represented by the empty list `{ }`.

16.53.4 Union and intersection

The operator `intersection` (the name used internally) has the shorter synonym `intersect`. These operators will probably most commonly be used as binary infix operators applied to explicit sets, e.g.

```
{1,2,3} union {2,3,4};

{1,2,3,4}

{1,2,3} intersect {2,3,4};

{2,3}
```

They can also be used as n-ary operators with any number of arguments, in which case it saves typing to use them as prefix operators (which is possible with all REDUCE infix operators), e.g.

```
{1,2,3} union {2,3,4} union {3,4,5};

{1,2,3,4,5}

intersect ({1,2,3}, {2,3,4}, {3,4,5});

{3}
```

For completeness, they can currently also be used as unary operators, in which case they just return their arguments (in canonical form), and so act as slightly less efficient versions of `mkset` (but this may change), e.g.

```
union {1,5,3,5,1};

{1,3,5}
```

16.53.5 Symbolic set expressions

If one or more of the arguments evaluates to an unbound identifier then it is regarded as representing a symbolic implicit set, and the union or intersection will evaluate to an expression that still contains the union or intersection operator. These two operators are symmetric, and so if they remain symbolic their arguments will be sorted as for any symmetric operator. Such symbolic set expressions are simplified, but the simplification may not be complete in non-trivial cases. For example:

```
a union b union {} union b union {7,3};

{3,7} union a union b

a intersect {};

{}
```

In implementations of REDUCE that provide fancy display using mathematical notation, such as PSL-REDUCE 3.6 for MS-Windows, the empty set, union, intersection and set difference are all displayed using their conventional mathematical symbols, namely \emptyset , \cup , \cap , \setminus .

A symbolic set expression is a valid argument for any other set operator, e.g.

```
a union (b intersect c);

b intersection c union a
```

Intersection distributes over union, which is not applied by default but is implemented as a rule list assigned to the variable `set_distribution_rule`, e.g.

```
a intersect (b union c);

(b union c) intersection a

a intersect (b union c) where set_distribution_rule;

a intersection b union a intersection c
```

16.53.6 Set difference

The set difference operator is represented by the symbol \setminus and is always output using this symbol, although it can also be input using either of the two names `setdiff` (the name used internally) or `minus` (as used in Maple). It is a binary operator, its operands may be any combination of explicit or implicit sets, and it may be used in an argument of any other set operator. Here are some examples:

```
{1,2,3} \ {2,4};

{1,3}

{1,2,3} \ {};
```

```

{1,2,3}

a \ {1,2};

a\{1,2}

a \ a;

{}

a \ {};

a

{} \ a;

{}

```

16.53.7 Predicates on sets

These are all binary infix operators. Currently, like all REDUCE predicates, they can only be used within conditional statements (`if`, `while`, `repeat`) or within the argument of the `evalb` operator provided by this package, and they cannot remain symbolic – a predicate that cannot be evaluated to a Boolean value causes a normal REDUCE error.

The `evalb` operator provides a convenient shorthand for an `if` statement designed purely to display the value of any Boolean expression (not only predicates defined in this package). It has some similarity with the `evalb` function in Maple, except that the values returned by `evalb` in REDUCE (the identifiers `true` and `false`) have no significance to REDUCE itself. Hence, in REDUCE, use of `evalb` is *never* necessary.

```

if a = a then true else false;

true

evalb(a = a);

true

if a = b then true else false;

false

```

```
evalb(a = b);
```

```
false
```

```
evalb 1;
```

```
true
```

```
evalb 0;
```

```
false
```

I will use the `evalb` operator in preference to an explicit `if` statement for purposes of illustration.

Set membership

Set membership is tested by the predicate `member`. Its left operand is regarded as a potential set element and its right operand *must* evaluate to an explicit set. There is currently no sense in which the right operand could be an implicit set; this would require a mechanism for declaring implicit set membership (akin to implicit variable dependence) which is currently not implemented. Set membership testing works like this:

```
evalb(1 member {1,2,3});
```

```
true
```

```
evalb(2 member {1,2} intersect {2,3});
```

```
true
```

```
evalb(a member b);
```

```
***** b invalid as list
```

Set inclusion

Set inclusion is tested by the predicate `subset_eq` where `a subset_eq b` is true if the set *a* is either a subset of or equal to the set *b*; strict inclusion is tested by the predicate `subset` where `a subset b` is true if the set *a* is *strictly* a subset of the set *b* and is false if *a* is equal to *b*. These predicates provide some support

for symbolic set expressions, but this is not yet correct as indicated below. Here are some examples:

```
evalb({1,2} subset_eq {1,2,3});
true
evalb({1,2} subset_eq {1,2});
true
evalb({1,2} subset {1,2});
false

evalb(a subset a union b);
true
evalb(a\b subset a);
true
evalb(a intersect b subset a union b);   %%% BUG
false
```

An undecidable predicate causes a normal REDUCE error, e.g.

```
evalb(a subset_eq {b});

***** Cannot evaluate a subset_eq {b} as Boolean-valued set
expression

evalb(a subset_eq b);   %%% BUG

false
```

Set equality

As explained above, equality of two sets in canonical form can be reliably tested by the standard REDUCE equality predicate (=). This package also provides the

predicate `set_eq` to test equality of two sets not represented canonically. The two predicates behave identically for operands that are symbolic set expressions because these are always evaluated to canonical form (although currently this is probably strictly true only in simple cases). Here are some examples:

```
evalb({1,2,3} = {1,2,3});

true

evalb({2,1,3} = {1,3,2});

false

evalb(mkset{2,1,3} = mkset{1,3,2});

true

evalb({2,1,3} set_eq {1,3,2});

true

evalb(a union a = a\{\});

true
```

16.53.8 Installation

The source file `sets.red` can be read into REDUCE when required using `IN`. If the “professional” version is being used this should be done with `ON COMP set`, but it is much better to compile the code as a FASL file using `FASLOUT` and then load it with `LOAD_PACKAGE` (or `LOAD`). See the REDUCE manual and implementation-specific guide for further details.

This package has to redefine the REDUCE internal procedure `mk!*sq` and a warning about this can be expected and ignored. I believe (and hope!) that this redefinition is safe and will not have any unexpected consequences for the rest of REDUCE.

16.53.9 Possible future developments

- Unary union/intersection to implement repeated union/intersection on a set of sets.

- More symbolic set algebra, canonical forms for set expressions, more complete simplification.
- Better support for Boolean variables via a version (evalb10?) of `evalb` that returns 1/0 instead of `true/false`, or predicates that return 1/0 directly.

16.54 SPDE: Finding symmetry groups of PDE's

The package SPDE provides a set of functions which may be used to determine the symmetry group of Lie- or point-symmetries of a given system of partial differential equations. In many cases the determining system is solved completely automatically. In other cases the user has to provide additional input information for the solution algorithm to terminate.

Author: Fritz Schwarz.

The package SPDE provides a set of functions which may be applied to determine the symmetry group of Lie- or point-symmetries of a given system of partial differential equations. Preferably it is used interactively on a computer terminal. In many cases the determining system is solved completely automatically. In some other cases the user has to provide some additional input information for the solution algorithm to terminate. The package should only be used in compiled form.

For all theoretical questions, a description of the algorithm and numerous examples the following articles should be consulted: “Automatically Determining Symmetries of Partial Differential Equations”, Computing vol. 34, page 91-106(1985) and vol. 36, page 279-280(1986), “Symmetries of Differential Equations: From Sophus Lie to Computer Algebra”, SIAM Review, to appear, and Chapter 2 of the Lecture Notes “Computer Algebra and Differential Equations of Mathematical Physics”, to appear.

16.54.1 Description of the System Functions and Variables

The symmetry analysis of partial differential equations logically falls into three parts. Accordingly the most important functions provided by the package are:

Function name	Operation
CRESYS(< arguments >)	Constructs determining system
SIMPSYS()	Solves determining system
RESULT()	Prints infinitesimal generators and commutator table

Some other useful functions for obtaining various kinds of output are:

Function name	Operation
PRSYS()	Prints determining system
PRGEN()	Prints infinitesimal generators
COMM(U,V)	Prints commutator of generators U and V

There are several global variables defined by the system which should not be used for any other purpose than that given in Table ?? and ??. The three globals of the type integer are:

Variable name	Meaning
NN	Number of independent variables
MM	Number of dependent variables
PCLASS=0, 1 or 2	Controls amount of output

In addition there are the following global variables of type operator:

Variable name	Meaning
X(I)	Independent variable x_i
U(ALFA)	Dependent variable u^{alfa}
U(ALFA,I)	Derivative of u^{alfa} w.r.t. x_i
DEQ(I)	i-th differential equation
SDER(I)	Derivative w.r.t. which DEQ(I) is resolved
GL(I)	i-th equation of determining system
GEN(I)	i-th infinitesimal generator
XI(I), ETA(ALFA) ZETA(ALFA,I)	See definition given in the references quoted in the introduction.
C(I)	i-th function used for substitution

The differential equations of the system at issue have to be assigned as values to the operator deq i applying the notation which is defined in Table ??. The entries in the third and the last line of that Table have obvious extensions to higher derivatives.

The derivative w.r.t. which the i-th differential equation deq i is resolved has to be assigned to sder i. Exception: If there is a single differential equation and no assignment has been made by the user, the highest derivative is taken by default.

When the appropriate assignments are made to the variable deq, the values of NN and MM (Table ??) are determined automatically, i.e. they have not to be assigned by the user.

The function CRESYS may be called with any number of arguments, i.e.

`CRESYS(); or CRESYS(deq 1, deq 2, ...);`

are legal calls. If it is called without any argument, all current assignments to deq are taken into account. Example: If deq 1, deq 2 and deq 3 have been assigned a differential equation and the symmetry group of the full system comprising all three equations is desired, equivalent calls are

```
CRESYS();    or    CRESYS(deq 1,deq 2,deq 3);
```

The first alternative saves some typing. If later in the session the symmetry group of deq 1 alone has to be determined, the correct call is

```
CRESYS deq 1;
```

After the determining system has been created, SIMPSYS which has no arguments may be called for solving it. The amount of intermediate output produced by SIMPSYS is controlled by the global variable PCLASS with the default value 0. With PCLASS equal to 0, no intermediate steps are shown. With PCLASS equal to 1, all intermediate steps are displayed so that the solution algorithm may be followed through in detail. Each time the algorithm passes through the top of the main solution loop the message

```
Entering main loop
```

is written. PCLASS equal 2 produces a lot of LISP output and is of no interest for the normal user.

If with PCLASS=0 the procedure SIMPSYS terminates without any response, the determining system is completely solved. In some cases SIMPSYS does not solve the determining system completely in a single run. In general this is true if there are only genuine differential equations left which the algorithm cannot handle at present. If a case like this occurs, SIMPSYS returns the remaining equations of the determining system. To proceed with the solution algorithm, appropriate assignments have to be transmitted by the user, e.g. the explicit solution for one of the returned differential equations. Any new functions which are introduced thereby must be operators of the form $c(k)$ with the correct dependencies generated by a depend statement (see the "REDUCE User's Guide"). Its enumeration has to be chosen in agreement with the current number of functions which have already been introduced. This value is returned by SIMPSYS too.

After the determining system has been solved, the procedure RESULT, which has no arguments, may be called. It displays the infinitesimal generators and its non-vanishing commutators.

16.54.2 How to Use the Package

In this Section it is explained by way of several examples how the package SPDE is used interactively to determine the symmetry group of partial differential equations. Consider first the diffusion equation which in the notation given above may be written as

```
deq 1:=u(1,1)+u(1,2,2);
```

It has been assigned as the value of deq 1 by this statement. There is no need to assign a value to sder 1 here because the system comprises only a single equation.

The determining system is constructed by calling

```
CRESYS(); or CRESYS deq 1;
```

The latter call is compulsory if there are other assignments to the operator deq i than for i=1.

The error message

```
***** Differential equations not defined
```

appears if there are no differential equations assigned to any deq.

If the user wants the determining system displayed for inspection before starting the solution algorithm he may call

```
PRSYS();
```

and gets the answer

```
GL(1):=2*DF(ETA(1),U(1),X(2)) - DF(XI(2),X(2),2) -
      DF(XI(2),X(1))
```

```
GL(2):=DF(ETA(1),U(1),2) - 2*DF(XI(2),U(1),X(2))
```

```
GL(3):=DF(ETA(1),X(2),2) + DF(ETA(1),X(1))
```

```
GL(4):=DF(XI(2),U(1),2)
```

```
GL(5):=DF(XI(2),U(1)) - DF(XI(1),U(1),X(2))
```

```
GL(6):=2*DF(XI(2),X(2)) - DF(XI(1),X(2),2) - DF(XI(1),X(1))
```

```
GL(7):=DF(XI(1),U(1),2)
```

```
GL(8):=DF(XI(1),U(1))
```

```
GL(9):=DF(XI(1),X(2))
```

The remaining dependencies

$XI(2)$ depends on $U(1), X(2), X(1)$

$XI(1)$ depends on $U(1), X(2), X(1)$

$ETA(1)$ depends on $U(1), X(2), X(1)$

The last message means that all three functions $XI(1)$, $XI(2)$ and $ETA(1)$ depend on $X(1)$, $X(2)$ and $U(1)$. Without this information the nine equations $GL(1)$ to $GL(9)$ forming the determining system are meaningless. Now the solution algorithm may be activated by calling

```
SIMPSYS();
```

If the print flag `PCLASS` has its default value which is 0 no intermediate output is produced and the answer is

Determining system is not completely solved

The remaining equations are

$GL(1) := DF(C(1), X(2), 2) + DF(C(1), X(1))$

Number of functions is 16

The remaining dependencies

$C(1)$ depends on $X(2), X(1)$

With `PCLASS` equal to 1 about 6 pages of intermediate output are obtained. It allows the user to follow through each step of the solution algorithm.

In this example the algorithm did not solve the determining system completely as it is shown by the last message. This was to be expected because the diffusion equation is linear and therefore the symmetry group contains a generator depending on a function which solves the original differential equation. In cases like this the user has to provide some additional information to the system so that the solution algorithm may continue. In the example under consideration the appropriate input is

```
DF(C(1), X(1)) := - DF(C(1), X(2), 2);
```


If now the solution algorithm is activated again by

```
SIMPSYS();
```

the solution algorithm terminates without any further message, i.e. there are no equations of the determining system left unsolved. To obtain the symmetry generators one has to say finally

```
RESULT();
```

and obtains the answer

The differential equation

```
DEQ(1):=U(1,2,2) + U(1,1)
```

The symmetry generators are

```
GEN(1):= DX(1)
```

```
GEN(2):= DX(2)
```

```
GEN(3):= 2*DX(2)*X(1) + DU(1)*U(1)*X(2)
```

```
GEN(4):= DU(1)*U(1)
```

```
GEN(5):= 2*DX(1)*X(1) + DX(2)*X(2)
```

```
GEN(6):= 4*DX(1)*X(1)2
```

```
+ 4*DX(2)*X(2)*X(1)
```

```
+ DU(1)*U(1)*(X(2)2 - 2*X(1))
```

```
GEN(7):= DU(1)*C(1)
```

The remaining dependencies

```
C(1) depends on X(2),X(1)
```

Constraints

$$DF(C(1), X(1)) := - DF(C(1), X(2), 2)$$

The non-vanishing commutators of the finite subgroup

$$COMM(1, 3) := 2 * DX(2)$$

$$COMM(1, 5) := 2 * DX(1)$$

$$COMM(1, 6) := 8 * DX(1) * X(1) + 4 * DX(2) * X(2) - 2 * DU(1) * U(1)$$

$$COMM(2, 3) := DU(1) * U(1)$$

$$COMM(2, 5) := DX(2)$$

$$COMM(2, 6) := 4 * DX(2) * X(1) + 2 * DU(1) * U(1) * X(2)$$

$$COMM(3, 5) := - (2 * DX(2) * X(1) + DU(1) * U(1) * X(2))$$

$$\begin{aligned} COMM(5, 6) := & 8 * DX(1) * X(1)^2 \\ & + 8 * DX(2) * X(2) * X(1) \\ & + 2 * DU(1) * U(1) * (X(2)^2 - 2 * X(1)) \end{aligned}$$

The message “Constraints” which appears after the symmetry generators are displayed means that the function $c(1)$ depends on $x(1)$ and $x(2)$ and satisfies the diffusion equation.

More examples which may be used for test runs are given in the final section.

If the user wants to test a certain ansatz of a symmetry generator for given differential equations, the correct proceeding is as follows. Create the determining system as described above. Make the appropriate assignments for the generator and call PRSYS() after that. The determining system with this ansatz substituted is returned. Example: Assume again that the determining system for the diffusion equation has been created. To check the correctness for example of generator GEN 3 which has been obtained above, the assignments

```
XI (1) :=0;   XI (2) :=2*X (1) ;   ETA (1) :=X (2) *U (1) ;
```

have to be made. If now PRSYS() is called all GL(K) are zero proving the correctness of this generator.

Sometimes a user only wants to know some of the functions ZETA for various values of its possible arguments and given values of MM and NN. In these cases the user has to assign the desired values of MM and NN and may call the ZETAs after that. Example:

```
MM:=1;   NN:=2;

FACTOR U (1,2) , U (1,1) , U (1,1,2) , U (1,1,1) ;

ON LIST;

ZETA (1,1) ;

-U (1,2) *U (1,1) *DF (XI (2) , U (1) )

-U (1,2) *DF (XI (2) , X (1) )

      2
-U (1,1) *DF (XI (1) , U (1) )

+U (1,1) * (DF (ETA (1) , U (1) ) -DF (XI (1) , X (1) ) )

+DF (ETA (1) , X (1) )

ZETA (1,1,1) ;

-2*U (1,1,2) *U (1,1) *DF (XI (2) , U (1) )

-2*U (1,1,2) *DF (XI (2) , X (1) )

-U (1,1,1) *U (1,2) *DF (XI (2) , U (1) )

-3*U (1,1,1) *U (1,1) *DF (XI (1) , U (1) )

+U (1,1,1) * (DF (ETA (1) , U (1) ) -2*DF (XI (1) , X (1) ) )

      2
-U (1,2) *U (1,1) *DF (XI (2) , U (1) , 2)
```

```

-2*U(1,2)*U(1,1)*DF(XI(2),U(1),X(1))

-U(1,2)*DF(XI(2),X(1),2)

      3
-U(1,1)*DF(XI(1),U(1),2)

      2
+U(1,1)*(DF(ETA(1),U(1),2)-2*DF(XI(1),U(1),X(1)))

+U(1,1)*(2*DF(ETA(1),U(1),X(1))-DF(XI(1),X(1),2))

+DF(ETA(1),X(1),2)

```

If by error no values to MM or NN and have been assigned the message

```
***** Number of variables not defined
```

is returned. Often the functions ZETA are desired for special values of its arguments ETA(ALFA) and XI(K). To this end they have to be assigned first to some other variable. After that they may be evaluated for the special arguments. In the previous example this may be achieved by

```
Z11:=ZETA(1,1)$      Z111:=ZETA(1,1,1)$
```

Now assign the following values to XI 1, XI 2 and ETA 1:

```

XI 1:=4*X(1)**2; XI 2:=4*X(2)*X(1);

ETA 1:=U(1)*(X(2)**2-2*X(1));

```

They correspond to the generator GEN 6 of the diffusion equation which has been obtained above. Now the desired expressions are obtained by calling

```

Z11;

      2
-(4*U(1,2)*X(2)-U(1,1)*X(2))+10*U(1,1)*X(1)+2*U(1))

Z111;

```

$$- (8*U(1,1,2)*X(2) - U(1,1,1)*X(2) + 18*U(1,1,1)*X(1) + 12*U(1,1))$$

16.54.3 Test File

This appendix is a test file. The symmetry groups for various equations or systems of equations are determined. The variable PCLASS has the default value 0 and may be changed by the user before running it. The output may be compared with the results which are given in the references.

```
%The Burgers equations

deq 1:=u(1,1)+u 1*u(1,2)+u(1,2,2)$

cresys deq 1$ simpsys()$ result()$

%The Kadomtsev-Petviashvili equation

deq 1:=3*u(1,3,3)+u(1,2,2,2,2)+6*u(1,2,2)*u 1
      +6*u(1,2)**2+4*u(1,1,2)$

cresys deq 1$ simpsys()$ result()$

%The modified Kadomtsev-Petviashvili equation

deq 1:=u(1,1,2)-u(1,2,2,2,2)-3*u(1,3,3)
      +6*u(1,2)**2*u(1,2,2)+6*u(1,3)*u(1,2,2)$

cresys deq 1$ simpsys()$ result()$

%The real- and the imaginary part of the nonlinear
%Schroedinger equation

deq 1:= u(1,1)+u(2,2,2)+2*u 1**2*u 2+2*u 2**3$

deq 2:=-u(2,1)+u(1,2,2)+2*u 1*u 2**2+2*u 1**3$

%Because this is not a single equation the two assignments

sder 1:=u(2,2,2)$ sder 2:=u(1,2,2)$
```

```

%are necessary.

cresys()$ simpsys()$ result()$

%The symmetries of the system comprising the four equations
deq 1:=u(1,1)+u 1*u(1,2)+u(1,2,2)$
deq 2:=u(2,1)+u(2,2,2)$
deq 3:=u 1*u 2-2*u(2,2)$
deq 4:=4*u(2,1)+u 2*(u 1**2+2*u(1,2))$

sder 1:=u(1,2,2)$ sder 2:=u(2,2,2)$ sder 3:=u(2,2)$
sder 4:=u(2,1)$

%is obtained by calling

cresys()$ simpsys()$

df(c 5,x 1):=-df(c 5,x 2,2)$
df(c 5,x 2,x 1):=-df(c 5,x 2,3)$

simpsys()$ result()$

% The symmetries of the subsystem comprising equation 1
% and 3 are obtained by

cresys(deq 1,deq 3)$ simpsys()$ result()$

% The result for all possible subsystems is discussed in
% detail in ``Symmetries and Involution Systems: Some
% Experiments in Computer Algebra'', contribution to the
% Proceedings of the Oberwolfach Meeting on Nonlinear
% Evolution Equations, Summer 1986, to appear.

```

16.55 SPECFN: Package for special functions

This special function package is separated into two portions to make it easier to handle. The packages are called SPECFN and SPECFN2. The first one is more general in nature, whereas the second is devoted to special special functions. Documentation for the first package can be found in the file `specfn.tex` in the “doc” directory, and examples in `specfn.tst` and `specfmor.tst` in the examples directory.

The package SPECFN is designed to provide algebraic and numerical manipulations of several common special functions, namely:

- Bernoulli Numbers and Euler Numbers;
- Stirling Numbers;
- Binomial Coefficients;
- Pochhammer notation;
- The Gamma function;
- The Psi function and its derivatives;
- The Riemann Zeta function;
- The Bessel functions J and Y of the first and second kind;
- The modified Bessel functions I and K;
- The Hankel functions H1 and H2;
- The Kummer hypergeometric functions M and U;
- The Beta function, and Struve, Lommel and Whittaker functions;
- The Airy functions;
- The Exponential Integral, the Sine and Cosine Integrals;
- The Hyperbolic Sine and Cosine Integrals;
- The Fresnel Integrals and the Error function;
- The Dilog function;
- Hermite Polynomials;
- Jacobi Polynomials;
- Legendre Polynomials;

- Spherical and Solid Harmonics;
- Laguerre Polynomials;
- Chebyshev Polynomials;
- Gegenbauer Polynomials;
- Euler Polynomials;
- Bernoulli Polynomials.
- Jacobi Elliptic Functions and Integrals;
- 3j symbols, 6j symbols and Clebsch Gordan coefficients;

Author: Chris Cannam, with contributions from Winfried Neun, Herbert Melenk, Victor Adamchik, Francis Wright and several others.

16.56 SPECFN2: Package for special special functions

This package provides algebraic manipulations of generalized hypergeometric functions and Meijer's G function. Generalized hypergeometric functions are simplified towards special functions and Meijer's G function is simplified towards special functions or generalized hypergeometric functions.

Author: Victor Adamchik, with major updates by Winfried Neun.

The (generalised) hypergeometric functions

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right)$$

are defined in textbooks on special functions as

$${}_pF_q \left(\begin{matrix} a_1, \dots, a_p \\ b_1, \dots, b_q \end{matrix} \middle| z \right) = \sum_{n=0}^{\infty} \frac{(a_1)_n \dots (a_p)_n}{(b_1)_n \dots (b_q)_n} \frac{z^n}{n!}$$

where $(a)_n$ is the Pochhammer symbol

$$(a)_n = \prod_{k=0}^{n-1} (a + k)$$

The function

$$G_{pq}^{mn} \left(z \middle| \begin{matrix} (a_p) \\ (b_q) \end{matrix} \right)$$

has been studied by C. S. Meijer beginning in 1936 and has been called Meijer's G function later on. The complete definition of Meijer's G function can be found in [?]. Many well-known functions can be written as G functions, e.g. exponentials, logarithms, trigonometric functions, Bessel functions and hypergeometric functions.

Several hundreds of particular values can be found in [?].

16.56.1 REDUCE operator HYPERGEOMETRIC

The operator `hypergeometric` expects 3 arguments, namely the list of upper parameters (which may be empty), the list of lower parameters (which may be empty too), and the argument, e.g. the input:

```
hypergeometric ( {}, {}, z );
```

yields the output

```
z
e
```

and the input

```
hypergeometric ( {1/2, 1}, {3/2}, -x^2 );
```

gives

```
atan(abs(x))
-----
abs(x)
```

16.56.2 Extending the HYPERGEOMETRIC operator

Since hundreds of particular cases for the generalised hypergeometric functions can be found in the literature, one cannot expect that all cases are known to the `hypergeometric` operator. Nevertheless the set of special cases can be augmented by adding rules to the REDUCE system, e.g.

```
let {hypergeometric({1/2, 1/2}, {3/2}, -(~x)^2) => asinh(x)/x};
```

16.56.3 REDUCE operator `meijerg`

The operator `meijerg` expects 3 arguments, namely the list of upper parameters (which may be empty), the list of lower parameters (which may be empty too), and the argument.

The first element of the lists has to be the list of the first n or m respective parameters, e.g. to describe

$$G_{11}^{10} \left(x \left| \begin{matrix} 1 \\ 0 \end{matrix} \right. \right)$$

one has to write

`MeijerG({},{},1,{{0}},x);` % and the result is:

$$\frac{\text{sign}(-x+1) + \text{sign}(x+1)}{2}$$

and for

$$G_{02}^{10} \left(\frac{x^2}{4} \left| 1 + \frac{1}{4}, 1 - \frac{1}{4} \right. \right)$$

`MeijerG({{}},{1+1/4,1-1/4},{(x^2)/4} * sqrt pi;`

$$\frac{\sqrt{\pi} * \sqrt{\frac{2}{\text{abs}(x) * \pi}} * \sin(\text{abs}(x)) * x}{4}$$

16.57 SUM: A package for series summation

This package implements the Gosper algorithm for the summation of series. It defines operators `SUM` and `PROD`. The operator `SUM` returns the indefinite or definite summation of a given expression, and `PROD` returns the product of the given expression.

This package loads automatically.

Author: Fujio Kako.

This package implements the Gosper algorithm for the summation of series. It defines operators `SUM` and `PROD`. The operator `SUM` returns the indefinite or definite summation of a given expression, and the operator `PROD` returns the product of the given expression. These are used with the syntax:

```
SUM(EXPR:expression, K:kernel, [LOLIM:expression [, UPLIM:expression]])
PROD(EXPR:expression, K:kernel, [LOLIM:expression [, UPLIM:expression]])
```

If there is no closed form solution, these operators return the input unchanged. `UPLIM` and `LOLIM` are optional parameters specifying the lower limit and upper limit of the summation (or product), respectively. If `UPLIM` is not supplied, the upper limit is taken as `K` (the summation variable itself).

For example:

```
sum(n**3,n);

sum(a+k*r,k,0,n-1);

sum(1/((p+(k-1)*q)*(p+k*q)),k,1,n+1);

prod(k/(k-2),k);
```

Gosper's algorithm succeeds whenever the ratio

$$\frac{\sum_{k=n_0}^n f(k)}{\sum_{k=n_0}^{n-1} f(k)}$$

is a rational function of n . The function `SUM!-SQ` handles basic functions such as polynomials, rational functions and exponentials.

The trigonometric functions `sin`, `cos`, etc. are converted to exponentials and then Gosper's algorithm is applied. The result is converted back into `sin`, `cos`, `sinh` and `cosh`.

Summations of logarithms or products of exponentials are treated by the formula:

$$\sum_{k=n_0}^n \log f(k) = \log \prod_{k=n_0}^n f(k)$$

$$\prod_{k=n_0}^n \exp f(k) = \exp \sum_{k=n_0}^n f(k)$$

Other functions, as shown in the test file for the case of binomials and formal products, can be summed by providing LET rules which must relate the functions evaluated at k and $k - 1$ (k being the summation variable).

There is a switch TRSUM (default OFF). If this switch is on, trace messages are printed out during the course of Gosper's algorithm.

16.58 SYMMETRY: Operations on symmetric matrices

This package computes symmetry-adapted bases and block diagonal forms of matrices which have the symmetry of a group. The package is the implementation of the theory of linear representations for small finite groups such as the dihedral groups.

Author: Karin Gatermann.

This short note describes a package of REDUCE procedures that compute symmetry-adapted bases and block diagonal forms of matrices which have the symmetry of a group. The SYMMETRY package is the implementation of the theory of linear representations for small finite groups such as the dihedral groups.

16.58.1 Introduction

The exploitation of symmetry is a very important principle in mathematics, physics and engineering sciences. The aim of the SYMMETRY package is to give an easy access to the underlying theory of linear representations for small groups. For example the dihedral groups D_3, D_4, D_5, D_6 are included. For an introduction to the theory see SERRE [3] or STIEFEL and FÄSSLER [4]. For a given orthogonal (or unitarian) linear representation

$$\vartheta : G \longrightarrow GL(K^n), \quad K = R, C.$$

the character $\psi \rightarrow K$, the canonical decomposition or the bases of the isotypic components are computed. A matrix A having the symmetry of a linear representation, e.g.

$$\vartheta_t A = A \vartheta_t \quad \forall t \in G,$$

is transformed to block diagonal form by a coordinate transformation. The dependence of the algorithm on the field of real or complex numbers is controlled by the switch `complex`. An example for this is given in the testfile *symmetry.tst*.

As the algorithm needs information concerning the irreducible representations this information is stored for some groups (see the operators in Section 3). It is assumed that only orthogonal (unitar) representations are given.

The package is loaded by

```
load symmetry;
```

16.58.2 Operators for linear representations

First the data structure for a linear representation has to be explained. *representation* is a list consisting of the group identifier and equations which assign matrices to the generators of the group.

Example:

```

rr:=mat((0,1,0,0),
        (0,0,1,0),
        (0,0,0,1),
        (1,0,0,0));

sp:=mat((0,1,0,0),
        (1,0,0,0),
        (0,0,0,1),
        (0,0,1,0));

representation:={D4,rD4=rr,sD4=sp};

```

For orthogonal (unitarian) representations the following operators are available.

`canonicaldecomposition(representation);`

returns an equation giving the canonical decomposition of the linear representation.

`character(representation);`

computes the character of the linear representation. The result is a list of the group identifier and of lists consisting of a list of group elements in one equivalence class and a real or complex number.

`symmetrybasis(representation,nr);`

computes the basis of the isotypic component corresponding to the irreducible representation of type nr. If the nr-th irreducible representation is multidimensional, the basis is symmetry adapted. The output is a matrix.

`symmetrybasispart(representation,nr);`

is similar as `symmetrybasis`, but for multidimensional irreducible representations only the first part of the symmetry adapted basis is computed.

`allsymmetrybases(representation);`

is similar as `symmetrybasis` and `symmetrybasispart`, but the bases of all isotypic components are computed and thus a complete coordinate transformation is returned.

`diagonalize(matrix,representation);`

returns the block diagonal form of matrix which has the symmetry of the given linear representation. Otherwise an error message occurs.

`on complex;`

Of course the property of irreducibility depends on the field K of real or complex numbers. This is why the algorithm depends on K . The type of computation is set by the switch *complex*.

16.58.3 Display Operators

In this section the operators are described which give access to the stored information for a group. First the operators for the abstract groups are given. Then it is described how to get the irreducible representations for a group.

`availablegroups()` ;

returns the list of all groups for which the information such as irreducible representations is stored. In the following `group` is always one of these group identifiers.

`printgroup(group)` ;

returns the list of all group elements;

`generators(group)` ;

returns a list of group elements which generates the group. For the definition of a linear representation matrices for these generators have to be defined.

`characterable(group)` ;

returns a list of the characters corresponding to the irreducible representations of this group.

`charactern(group, nr)` ;

returns the character corresponding to the `nr`-th irreducible representation of this group as a list (see also `character`).

`irreduciblereptable(group)` ;

returns the list of irreducible representations of the group.

`irreduciblerepnr(group, nr)` ;

returns an irreducible representation of the group. The output is a list of the group identifier and equations assigning the representation matrices to group elements.

16.58.4 Storing a new group

If the user wants to do computations for a group for which information is not pre-defined, the package SYMMETRY offers the possibility to supply information for this group.

For this the following data structures are used.

elemlist = list of identifiers.

relationlist = list of equations with identifiers and operators @ and **.

grouptable = matrix with the (1,1)-entry grouptable.

filename = "myfilename.new".

The following operators have to be used in this order.

```
setgenerators(group,elemlist,relationlist);
```

Example:

```
setgenerators(K4,{s1K4,s2K4},
  {s1K4^2=id,s2K4^2=id,s1K4@s2K4=s2K4@s1K4});
```

setelements(group,relationlist);

The group elements except the neutral element are given as product of the defined generators. The neutral element is always called *id*.

Example:

```
setelements(K4,
  {s1K4=s1K4,s2K4=s2K4,rK4=s1K4@s2K4});
```

setgrouptable(group,grouptable);

installs the group table.

Example:

```
tab:=
  mat((grouptable,      id,      s1K4, s2K4, rK4),
      (id              ,      id,      s1K4, s2K4, rK4),
      (s1K4            ,      s1K4,      id, rK4,s2K4),
      (s2K4            ,      s2K4,      rK4,  id,s1K4),
      (rK4             ,      rK4,      s2K4, s1K4,  id));

setgrouptable(K4,tab);
```

Rsetrepresentation(representation,type);

is used to define the real irreducible representations of the group. The variable *type* is either *realttype* or *complexttype* which indicates the type of the real irre-

ducible representation.

Example:

```
eins:=mat((1));
mineins:=mat((-1));
rep3:={K4,s1K4=eins,s2K4=mineins};
Rsetrepresentation(rep3,realtype);
```

Csetrepresentation(representation);

This defines the complex irreducible representations.

setavailable(group);

terminates the installation of the group203. It checks some properties of the irreducible representations and makes the group available for the operators in Sections 2 and 3.

storegroup(group,filename);

writes the information concerning the group to the file with name *filename*.

loadgroups(filename);

loads a user defined group from the file *filename* into the system.

Bibliography

- [1] G. James, A. Kerber: *Representation Theory of the Symmetric Group*. Addison, Wesley (1981).
- [2] W. Ludwig, C. Falter: *Symmetries in Physics*. Springer, Berlin, Heidelberg, New York (1988).
- [3] J.-P. Serre, *Linear Representations of Finite Groups*. Springer, New York (1977).
- [4] E. Stiefel, A. Fässler, *Gruppentheoretische Methoden und ihre Anwendung*. Teubner, Stuttgart (1979). (English translation to appear by Birkhäuser (1992)).

16.59 TAYLOR: Manipulation of Taylor series

This package carries out the Taylor expansion of an expression in one or more variables and efficient manipulation of the resulting Taylor series. Capabilities include basic operations (addition, subtraction, multiplication and division) and also application of certain algebraic and transcendental functions.

Author: Rainer Schöpf.

16.59.1 Basic Use

The most important operator is 'TAYLOR'. It is used as follows:

```
TAYLOR(EXP:algebraic,
      VAR:kernel,VAR0:algebraic,ORDER:integer[,...])
      :algebraic.
```

where EXP is the expression to be expanded. It can be any REDUCE object, even an expression containing other Taylor kernels. VAR is the kernel with respect to which EXP is to be expanded. VAR0 denotes the point about which and ORDER the order up to which expansion is to take place. If more than one (VAR, VAR0, ORDER) triple is specified TAYLOR will expand its first argument independently with respect to each variable in turn. For example,

```
taylor(e^(x^2+y^2),x,0,2,y,0,2);
```

will calculate the Taylor expansion up to order $X^2 * Y^2$:

$$1 + y^2 + x^2 + y^2 * x^2 + O(x^3, y^3)$$

Note that once the expansion has been done it is not possible to calculate higher orders. Instead of a kernel, VAR may also be a list of kernels. In this case expansion will take place in a way so that the *sum* of the degrees of the kernels does not exceed ORDER. If VAR0 evaluates to the special identifier INFINITY, expansion is done in a series in 1/VAR instead of VAR.

The expansion is performed variable per variable, i.e. in the example above by first expanding $\exp(x^2 + y^2)$ with respect to x and then expanding every coefficient with respect to y .

There are two extra operators to compute the Taylor expansions of implicit and inverse functions:

```
IMPLICIT_TAYLOR(F:algebraic,
```

```

VAR:kernel,DEPVAR:kernel,
VAR0:algebraic,DEPVAR0:algebraic,
ORDER:integer)
:algebraic

```

takes a function F depending on two variables VAR and $DEPVAR$ and computes the Taylor series of the implicit function $DEPVAR(VAR)$ given by the equation $F(VAR,DEPVAR) = 0$, around the point $VAR0$. (Violation of the necessary condition $F(VAR0,DEPVAR0)=0$ causes an error.) For example,

```
implicit_taylor(x^2 + y^2 - 1,x,y,0,1,5);
```

gives the output

$$1 - \frac{1}{2}x^2 - \frac{1}{8}x^4 + O(x^6)$$

The operator

```

INVERSE_TAYLOR(F:algebraic,VAR:kernel,DEPVAR:kernel,
VAR0:algebraic,ORDER:integer)
: algebraic

```

takes a function F depending on $VAR1$ and computes the Taylor series of the inverse of F with respect to $VAR2$. For example,

```
inverse_taylor(exp(x)-1,x,y,0,8);
```

yields

$$y - \frac{1}{2}y^2 + \frac{1}{3}y^3 - \frac{1}{4}y^4 + \frac{1}{5}y^5 + (3 \text{ terms}) + O(y^9)$$

When a Taylor kernel is printed, only a certain number of (non-zero) coefficients are shown. If there are more, an expression of the form $(n \text{ terms})$ is printed to indicate how many non-zero terms have been suppressed. The number of terms printed is given by the value of the shared algebraic variable `TAYLORPRINTTERMS`. Allowed values are integers and the special identifier `ALL`. The latter setting specifies that all terms are to be printed. The default setting is 5.

The `PART` operator can be used to extract subexpressions of a Taylor expansion in the usual way. All terms can be accessed, irregardless of the value of the variable

TAYLORPRINTTERMS.

If the switch `TAYLORKEEPORIGINAL` is set to ON the original expression EXP is kept for later reference. It can be recovered by means of the operator

TAYLORORIGINAL(EXP:*exprn*):*exprn*

An error is signalled if EXP is not a Taylor kernel or if the original expression was not kept, i.e. if `TAYLORKEEPORIGINAL` was OFF during expansion. The template of a Taylor kernel, i.e. the list of all variables with respect to which expansion took place together with expansion point and order can be extracted using .

TAYLORTEMPLATE(EXP:*exprn*):*list*

This returns a list of lists with the three elements (VAR,VAR0,ORDER). As with TAYLORORIGINAL, an error is signalled if EXP is not a Taylor kernel.

The operator

TAYLORTOSTANDARD(EXP:*exprn*):*exprn*

converts all Taylor kernels in EXP into standard form and resimplifies the result.

The boolean operator

TAYLORSERIESP(EXP:*exprn*):*boolean*

may be used to determine if EXP is a Taylor kernel. (Note that this operator is subject to the same restrictions as, e.g., ORDP or NUMBERP, i.e. it may only be used in boolean expressions in IF or LET statements.

Finally there is

TAYLORCOMBINE(EXP:*exprn*):*exprn*

which tries to combine all Taylor kernels found in EXP into one. Operations currently possible are:

- Addition, subtraction, multiplication, and division.
- Roots, exponentials, and logarithms.
- Trigonometric and hyperbolic functions and their inverses.

Application of unary operators like LOG and ATAN will nearly always succeed. For binary operations their arguments have to be Taylor kernels with the same template. This means that the expansion variable and the expansion point must match. Expansion order is not so important, different order usually means that one of them is truncated before doing the operation.

If `TAYLORKEEPORIGINAL` is set to ON and if all Taylor kernels in `exp` have their original expressions kept `TAYLORCOMBINE` will also combine these and store the result as the original expression of the resulting Taylor kernel. There is also the switch `TAYLORAUTOEXPAND` (see below).

There are a few restrictions to avoid mathematically undefined expressions: it is not possible to take the logarithm of a Taylor kernel which has no terms (i.e. is zero), or to divide by such a beast. There are some provisions made to detect singularities during expansion: poles that arise because the denominator has zeros at the expansion point are detected and properly treated, i.e. the Taylor kernel will start with a negative power. (This is accomplished by expanding numerator and denominator separately and combining the results.) Essential singularities of the known functions (see above) are handled correctly.

Differentiation of a Taylor expression is possible. If you differentiate with respect to one of the Taylor variables the order will decrease by one.

Substitution is a bit restricted: Taylor variables can only be replaced by other kernels. There is one exception to this rule: you can always substitute a Taylor variable by an expression that evaluates to a constant. Note that REDUCE will not always be able to determine that an expression is constant.

Only simple Taylor kernels can be integrated. More complicated expressions that contain Taylor kernels as parts of themselves are automatically converted into a standard representation by means of the `TAYLORTOSTANDARD` operator. In this case a suitable warning is printed.

It is possible to revert a Taylor series of a function f , i.e., to compute the first terms of the expansion of the inverse of f from the expansion of f . This is done by the operator

`TAYLORREVERT(EXP:expn,OLDVAR:kernel, NEWVAR:kernel):expn`

EXP must evaluate to a Taylor kernel with OLDVAR being one of its expansion variables. Example:

```
taylor (u - u**2, u, 0, 5)$
taylorrevert (ws, u, x);
```

gives

$$x^2 + x^3 + 2x^4 + 5x^5 + 14x^6 + O(x^7)$$

This package introduces a number of new switches:

`TAYLORAUTOCOMBINE` causes Taylor expressions to be automatically combined during the simplification process. This is equivalent to applying `TAYLORCOMBINE` to every expression that contains Taylor kernels. Default is ON.

`TAYLORAUTOEXPAND` makes Taylor expressions “contagious” in the sense that `TAYLORCOMBINE` tries to Taylor expand all non-Taylor subexpressions and

to combine the result with the rest. Default is OFF.

TAYLORKEEPORIGINAL forces the package to keep the original expression, i.e. the expression that was Taylor expanded. All operations performed on the Taylor kernels are also applied to this expression which can be recovered using the operator TAYLORORIGINAL. Default is OFF.

TAYLORPRINTORDER causes the remainder to be printed in big- O notation. Otherwise, three dots are printed. Default is ON.

VERBOSELOAD will cause REDUCE to print some information when the Taylor package is loaded. This switch is already present in PSL systems. Default is OFF.

16.59.2 Caveats

TAYLOR should always detect non-analytical expressions in its first argument. As an example, consider the function $xy/(x+y)$ that is not analytical in the neighborhood of $(x, y) = (0, 0)$: Trying to calculate

```
taylor(x*y/(x+y), x, 0, 2, y, 0, 2);
```

causes an error

```
***** Not a unit in argument to QUOTTAYLOR
```

Note that it is not generally possible to apply the standard REDUCE operators to a Taylor kernel. For example, PART, COEFF, or COEFFN cannot be used. Instead, the expression at hand has to be converted to standard form first using the TAYLORTOSTANDARD operator.

16.59.3 Warning messages

```
*** Cannot expand further... truncation done
    You will get this warning if you try to expand a Taylor kernel to a higher
    order.
```

```
*** Converting Taylor kernels to standard representation
```

This warning appears if you try to integrate an expression containing Taylor kernels.

16.59.4 Error messages

***** Branch point detected in ...

This occurs if you take a rational power of a Taylor kernel and raising the lowest order term of the kernel to this power yields a non analytical term (i.e. a fractional power).

***** Cannot replace part ... in Taylor kernel

The PART operator can only be used to either replace the template of a Taylor kernel (part 2) or the original expression that is kept for reference (part 3).

***** Computation loops (recursive definition?): ...

Most probably the expression to be expanded contains an operator whose derivative involves the operator itself.

***** Error during expansion (possible singularity)

The expression you are trying to expand caused an error. As far as I know this can only happen if it contains a function with a pole or an essential singularity at the expansion point. (But one can never be sure.)

***** Essential singularity in ...

An essential singularity was detected while applying a special function to a Taylor kernel.

***** Expansion point lies on branch cut in ...

The only functions with branch cuts this package knows of are (natural) logarithm, inverse circular and hyperbolic tangent and cotangent. The branch cut of the logarithm is assumed to lie on the negative real axis. Those of the arc tangent and arc cotangent functions are chosen to be compatible with this: both have essential singularities at the points $\pm i$. The branch cut of arc tangent is the straight line along the imaginary axis connecting $+1$ to -1 going through ∞ whereas that of arc cotangent goes through the origin. Consequently, the branch cut of the inverse hyperbolic tangent resp. cotangent lies on the real axis and goes from -1 to $+1$, that of the latter across 0 , the other across ∞ .

The error message can currently only appear when you try to calculate the inverse tangent or cotangent of a Taylor kernel that starts with a negative degree. The case of a logarithm of a Taylor kernel whose constant term is a negative real number is not caught since it is difficult to detect this in general.

***** Input expression non-zero at given point

Violation of the necessary condition $F(\text{VAR0}, \text{DEPVAR0})=0$ for the arguments of IMPLICIT_TAYLOR.

***** Invalid substitution in Taylor kernel: ...

You tried to substitute a variable that is already present in the Taylor kernel or on which one of the Taylor variables depend.

```

***** Not a unit in ...
    This will happen if you try to divide by or take the logarithm of a Taylor
    series whose constant term vanishes.

***** Not implemented yet (...)
    Sorry, but I haven't had the time to implement this feature. Tell me if you
    really need it, maybe I have already an improved version of the package.

***** Reversion of Taylor series not possible: ...
    You tried to call the TAYLORREVERT operator with inappropriate argu-
    ments. The second half of this error message tells you why this operation is
    not possible.

***** Taylor kernel doesn't have an original part
    The Taylor kernel upon which you try to use TAYLORORIGINAL was cre-
    ated with the switch TAYLORKEEPORIGINAL set to OFF and does there-
    fore not keep the original expression.

***** Wrong number of arguments to TAYLOR
    You try to use the operator TAYLOR with a wrong number of arguments.

***** Zero divisor in TAYLOREXPAND
    A zero divisor was found while an expression was being expanded. This
    should not normally occur.

***** Zero divisor in Taylor substitution
    That's exactly what the message says. As an example consider the case of a
    Taylor kernel containing the term  $1/x$  and you try to substitute  $x$  by 0.

***** ... invalid as kernel
    You tried to expand with respect to an expression that is not a kernel.

***** ... invalid as order of Taylor expansion
    The order parameter you gave to TAYLOR is not an integer.

***** ... invalid as Taylor kernel
    You tried to apply TAYLORORIGINAL or TAYLORTEMPLATE to an ex-
    pression that is not a Taylor kernel.

***** ... invalid as Taylor Template element
    You tried to substitute the TAYLORTEMPLATE part of a Taylor kernel with
    a list a incorrect form. For the correct form see the description of the
    TAYLORTEMPLATE operator.

***** ... invalid as Taylor variable
    You tried to substitute a Taylor variable by an expression that is not a kernel.

```



```
***** ... invalid as value of TaylorPrintTerms
```

You have assigned an invalid value to `TAYLORPRINTTERMS`. Allowed values are: an integer or the special identifier `ALL`.

```
TAYLOR PACKAGE (...): this can't happen ...
```

This message shows that an internal inconsistency was detected. This is not your fault, at least as long as you did not try to work with the internal data structures of `REDUCE`. Send input and output to me, together with the version information that is printed out.

16.59.5 Comparison to other packages

At the moment there is only one `REDUCE` package that I know of: the truncated power series package by Alan Barnes and Julian Padgett. In my opinion there are two major differences:

- The interface. They use the domain mechanism for their power series, I decided to invent a special kind of kernel. Both approaches have advantages and disadvantages: with domain modes, it is easier to do certain things automatically, e.g., conversions.
- The concept of a truncated series. Their idea is to remember the original expression and to compute more coefficients when more of them are needed. My approach is to truncate at a certain order and forget how the unexpanded expression looked like. I think that their method is more widely usable, whereas mine is more efficient when you know in advance exactly how many terms you need.

16.60 TPS: A truncated power series package

This package implements formal Laurent series expansions in one variable using the domain mechanism of REDUCE. This means that power series objects can be added, multiplied, differentiated etc., like other first class objects in the system. A lazy evaluation scheme is used and thus terms of the series are not evaluated until they are required for printing or for use in calculating terms in other power series. The series are extendible giving the user the impression that the full infinite series is being manipulated. The errors that can sometimes occur using series that are truncated at some fixed depth (for example when a term in the required series depends on terms of an intermediate series beyond the truncation depth) are thus avoided.

Authors: Alan Barnes and Julian Padget.

16.60.1 Introduction

This package implements formal power series expansions in one variable using the domain mechanism of REDUCE. This means that power series objects can be added, multiplied, differentiated etc. like other first class objects in the system. A lazy evaluation scheme is used in the package and thus terms of the series are not evaluated until they are required for printing or for use in calculating terms in other power series. The series are extendible giving the user the impression that the full infinite series is being manipulated. The errors that can sometimes occur using series that are truncated at some fixed depth (for example when a term in the required series depends on terms of an intermediate series beyond the truncation depth) are thus avoided.

Below we give a brief description of the operators available in the power series package together with some examples of their use.

16.60.2 PS Operator

Syntax:

`PS(EXPRN:algebraic,DEPVAR:kernel,ABOUT:algebraic):ps object`

The `PS` operator returns a power series object (a tagged domain element) representing the univariate formal power series expansion of `EXPRN` with respect to the dependent variable `DEPVAR` about the expansion point `ABOUT`. `EXPRN` may itself contain power series objects.

The algebraic expression `ABOUT` should simplify to an expression which is independent of the dependent variable `DEPVAR`, otherwise an error will result. If

ABOUT is the identifier INFINITY then the power series expansion about $\text{DEPVAR} = \infty$ is obtained in ascending powers of $1/\text{DEPVAR}$.

If the command is terminated by a semi-colon, a power series object representing EXPRN is compiled and then a number of terms of the power series expansion are evaluated and printed. The expansion is carried out as far as the value specified by PSEXPLIM. If, subsequently, the value of PSEXPLIM is increased, sufficient information is stored in the power series object to enable the additional terms to be calculated without recalculating the terms already obtained.

If the command is terminated by a dollar symbol, a power series object is compiled, but at most one term is calculated at this stage.

If the function has a pole at the expansion point then the correct Laurent series expansion will be produced.

The following examples are valid uses of PS:

```
psexplim 6;
ps(log x,x,1);
ps(e**(sin x),x,0);
ps(x/(1+x),x,infinity);
ps(sin x/(1-cos x),x,0);
```

New user-defined functions may be expanded provided the user provides LET rules giving

1. the value of the function at the expansion point
2. a differentiation rule for the new function.

For example

```
operator sech;
forall x let df(sech x,x)= - sech x * tanh x;
let sech 0 = 1;
ps(sech(x**2),x,0);
```

The power series expansion of an integral may also be obtained (even if REDUCE cannot evaluate the integral in closed form). An example of this is

```
ps(int(e**x/x,x),x,1);
```

Note that if the integration variable is the same as the expansion variable then REDUCE's integration package is not called; if on the other hand the two variables are different then the integrator is called to integrate each of the coefficients in the

power series expansion of the integrand. The constant of integration is zero by default.

16.60.3 PSEXPLIM Operator

Syntax:

`PSEXPLIM(UPTO:integer):integer`

or

`PSEXPLIM():integer`

Calling this operator sets an internal variable of the TPS package to the value of UPTO (which should evaluate to an integer). The value returned is the previous value of this variable. The default value is six.

If PSEXPLIM is called with no argument, the current value for the expansion limit is returned.

16.60.4 PSORDLIM Operator

Syntax:

`PSORDLIM(UPTO:integer):integer`

or

`PSORDLIM():integer`

An internal variable is set to the value of UPTO (which should evaluate to an integer). The value returned is the previous value of the variable. The default value is 15.

If PSORDLIM is called with no argument, the current value is returned.

The significance of this control is that the system attempts to find the order of the power series required, that is the order is the degree of the first non-zero term in the power series. If the order is greater than the value of this variable an error message is given and the computation aborts. This prevents infinite loops in examples such as

```
ps(1 - (sin x)**2 - (cos x)**2, x, 0);
```

where the expression being expanded is identically zero, but is not recognized as such by REDUCE.

16.60.5 PSTERM Operator

Syntax:

`PSTERM(TPS:power series object, NTH:integer):algebraic`

The operator `PSTERM` returns the `NTH` term of the existing power series object `TPS`. If `NTH` does not evaluate to an integer or `TPS` to a power series object an error results. It should be noted that an integer is treated as a power series.

16.60.6 PSORDER Operator

Syntax:

`PSORDER(TPS:power series object):integer`

The operator `PSORDER` returns the order, that is the degree of the first non-zero term, of the power series object `TPS`. `TPS` should evaluate to a power series object or an error results. If `TPS` is zero, the identifier `UNDEFINED` is returned.

16.60.7 PSSETORDER Operator

Syntax:

`PSSETORDER(TPS:power series object, ORD:integer):integer`

The operator `PSSETORDER` sets the order of the power series `TPS` to the value `ORD`, which should evaluate to an integer. If `TPS` does not evaluate to a power series object, then an error occurs. The value returned by this operator is the previous order of `TPS`, or 0 if the order of `TPS` was undefined. This operator is useful for setting the order of the power series of a function defined by a differential equation in cases where the power series package is inadequate to determine the order automatically.

16.60.8 PSDEPVAR Operator

Syntax:

`PSDEPVAR(TPS:power series object) :identifier`

The operator `PSDEPVAR` returns the expansion variable of the power series object `TPS`. `TPS` should evaluate to a power series object or an integer, otherwise an error results. If `TPS` is an integer, the identifier `UNDEFINED` is returned.

16.60.9 PSEXPANSIONPT operator

Syntax:

`PSEXPANSIONPT(TPS:power series object):algebraic`

The operator `PSEXPANSIONPT` returns the expansion point of the power series object `TPS`. `TPS` should evaluate to a power series object or an integer, otherwise an error results. If `TPS` is integer, the identifier `UNDEFINED` is returned. If the expansion is about infinity, the identifier `INFINITY` is returned.

16.60.10 PSFUNCTION Operator

Syntax:

`PSFUNCTION(TPS:power series object):algebraic`

The operator `PSFUNCTION` returns the function whose expansion gave rise to the power series object `TPS`. `TPS` should evaluate to a power series object or an integer, otherwise an error results.

16.60.11 PSCHANGEVAR Operator

Syntax:

`PSCHANGEVAR(TPS:power series object, X:kernel):power series object`

The operator `PSCHANGEVAR` changes the dependent variable of the power series object `TPS` to the variable `X`. `TPS` should evaluate to a power series object and `X` to a kernel, otherwise an error results. Also `X` should not appear as a parameter in `TPS`. The power series with the new dependent variable is returned.

16.60.12 PSREVERSE Operator

Syntax:

`PSREVERSE(TPS:power series object):power series`

Power series reversion. The power series `TPS` is functionally inverted. Four cases arise:

1. If the order of the series is 1, then the expansion point of the inverted series is 0.
2. If the order is 0 *and* if the first order term in `TPS` is non-zero, then the expansion point of the inverted series is taken to be the coefficient of the zeroth order term in `TPS`.

3. If the order is -1 the expansion point of the inverted series is the point at infinity. In all other cases a REDUCE error is reported because the series cannot be inverted as a power series. Puiseux expansion would be required to handle these cases.
4. If the expansion point of TPS is finite it becomes the zeroth order term in the inverted series. For expansion about 0 or the point at infinity the order of the inverted series is one.

If TPS is not a power series object after evaluation an error results.

Here are some examples:

```
ps(sin x,x,0);
psreverse(ws); % produces series for asin x about x=0.
ps(exp x,x,0);
psreverse ws; % produces series for log x about x=1.
ps(sin(1/x),x,infinity);
psreverse(ws); % series for 1/asin(x) about x=0.
```

16.60.13 PSCOMPOSE Operator

Syntax:

`PSCOMPOSE(TPS1:power series, TPS2:power series):power series`

PSCOMPOSE performs power series composition. The power series TPS1 and TPS2 are functionally composed. That is to say that TPS2 is substituted for the expansion variable in TPS1 and the result expressed as a power series. The dependent variable and expansion point of the result coincide with those of TPS2. The following conditions apply to power series composition:

1. If the expansion point of TPS1 is 0 then the order of the TPS2 must be at least 1.
2. If the expansion point of TPS1 is finite, it should coincide with the coefficient of the zeroth order term in TPS2. The order of TPS2 should also be non-negative in this case.
3. If the expansion point of TPS1 is the point at infinity then the order of TPS2 must be less than or equal to -1.

If these conditions do not hold the series cannot be composed (with the current algorithm terms of the inverted series would involve infinite sums) and a REDUCE error occurs.

Examples of power series composition include the following.

```

a:=ps(exp y,y,0);  b:=ps(sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(sin x)
% about x=0.

a:=ps(exp z,z,1); b:=ps(cos x,x,0);
pscompose(a,b);
% Produces the power series expansion of exp(cos x)
% about x=0.

a:=ps(cos(1/x),x,infinity);  b:=ps(1/sin x,x,0);
pscompose(a,b);
% Produces the power series expansion of cos(sin x)
% about x=0.

```

16.60.14 PSSUM Operator

Syntax:

```

PSSUM(J:kernel = LOWLIM:integer, COEFF:algebraic, X:kernel,
      ABOUT:algebraic, POWER:algebraic):power series

```

The formal power series sum for J from LOWLIM to INFINITY of

$$\text{COEFF} * (\text{X} - \text{ABOUT}) ** \text{POWER}$$

or if ABOUT is given as INFINITY

$$\text{COEFF} * (1/\text{X}) ** \text{POWER}$$

is constructed and returned. This enables power series whose general term is known to be constructed and manipulated using the other procedures of the power series package.

J and X should be distinct simple kernels. The algebraics ABOUT, COEFF and POWER should not depend on the expansion variable X, similarly the algebraic ABOUT should not depend on the summation variable J. The algebraic POWER should be a strictly increasing integer valued function of J for J in the range LOWLIM to INFINITY.

```

pssum(n=0,1,x,0,n*n);
% Produces the power series summation for n=0 to
% infinity of x**(n*n).

```



```

pssum(m=1, (-1)**(m-1)/(2m-1), y, 1, 2m-1);
% Produces the power series expansion of atan(y-1)
% about y=1.

pssum(j=1, -1/j, x, infinity, j);
% Produces the power series expansion of log(1-1/x)
% about the point at infinity.

pssum(n=0, 1, x, 0, 2n**2+3n) + pssum(n=1, 1, x, 0, 2n**2-3n);
% Produces the power series summation for n=-infinity
% to +infinity of x**(2n**2+3n).

```

16.60.15 PSCOPY Operator

Syntax:

`PSCOPY(TPS:power series):power series`

This procedure returns a copy of the power series TPS. The copy has no shared sub-structures in common with the original series. This enables substitutions to be performed on the series without side-effects on previously computed objects. For example:

```

clear a;
b := ps(sin(a*x)), x, 0);
b where a => 1;

```

will result in a being set to 1 in each of the terms of the power series and the resulting expressions being simplified. Owing to the way power series objects are implemented using Lisp vectors, this has the side-effect that the value of b is changed. This may be avoided by copying the series with PSCOPY before applying the substitution, thus:

```

b := ps(sin(a*x)), x, 0);
pscopy b where a => 1;

```

16.60.16 PSTRUNCATE Operator

Syntax:

`PSTRUNCATE(TPS:power series POWER: integer):algebraic`

This procedure truncates the power series TPS discarding terms of order higher

than `POWER`. The series is extended automatically if the value of `POWER` is greater than the order of last term calculated to date.

```
b := ps(sin x, x, 0);
a := pstruncate(b, 11);
```

will result in `a` being set to the eleventh order polynomial resulting in truncating the series for $\sin x$ after the term involving x^{11} .

If `POWER` is less than the order of the series then 0 is returned. If `POWER` does not simplify to an integer or if `TPS` is not a power series object then Reduce errors result.

16.60.17 Arithmetic Operations

As power series objects are domain elements they may be combined together in algebraic expressions in algebraic mode of `REDUCE` in the normal way.

For example if `A` and `B` are power series objects then the commands such as:

```
a*b;
a/b;
a**2+b**2;
```

will produce power series objects representing the product, quotient and the sum of the squares of the power series objects `A` and `B` respectively.

16.60.18 Differentiation

If `A` is a power series object depending on `X` then the input `df(a, x);` will produce the power series expansion of the derivative of `A` with respect to `X`.

Note however that currently the input `int(a, x);` will not work as intended; instead one must input `ps(int(a, x), x, 0);` in order to obtain the power series expansion of the integral of `a`.

16.60.19 Restrictions and Known Bugs

If `A` is a power series object and `X` is a variable which evaluates to itself then currently expressions such as `a*x` do not evaluate to a single power series object (although the result is formally valid). Instead use `ps(a*x, x, 0)` etc..

Similarly expressions such as `sin(A)` where `A` is a PS object currently will not be expanded. For example:

```
a:=ps(1/(1+x),x,0);
b:=sin a;
```

will not expand $\sin(1/(1+x))$ as a power series. In fact

```
SIN(1 - X + X**2 - X**3 + .....)
```

will be returned. However,

```
b:=ps(sin(a),x,0);
```

or

```
b:=ps(sin(1/(1+x)),x,0);
```

should work as intended.

The handling of functions with essential singularities is currently erratic: usually an error message

```
***** Essential Singularity
```

or

```
***** Logarithmic Singularity
```

occurs but occasionally a division by zero error or some drastic error like (for PSL) binding stack overflow may occur.

There is no simple way to write the results of power series calculation to a file and read them back into REDUCE at a later stage.

16.61 TRI: TeX REDUCE interface

This package provides facilities written in REDUCE-Lisp for typesetting REDUCE formulas using \TeX . The \TeX -REDUCE-Interface incorporates three levels of \TeX output: without line breaking, with line breaking, and with line breaking plus indentation.

Author: Werner Antweiler.

16.62 TRIGSIMP: Simplification and factorization of trigonometric and hyperbolic functions

TRIGSIMP is a useful tool for all kinds of trigonometric and hyperbolic simplification and factorization. There are three procedures included in TRIGSIMP: `trigsimp`, `trigfactorize` and `triggcd`. The first is for finding simplifications of trigonometric or hyperbolic expressions with many options, the second for factorizing them and the third for finding the greatest common divisor of two trigonometric or hyperbolic polynomials.

Author: Wolfram Koepf.

16.62.1 Introduction

The REDUCE package TRIGSIMP is a useful tool for all kinds of problems related to trigonometric and hyperbolic simplification and factorization. There are three operators included in TRIGSIMP: `trigsimp`, `trigfactorize` and `triggcd`. The first is for simplifying trigonometric or hyperbolic expressions and has many options, the second is for factorizing them and the third is for finding the greatest common divisor of two trigonometric or hyperbolic polynomials. This package is automatically loaded when one of these operators is used.

16.62.2 Simplifying trigonometric expressions

As there is no normal form for trigonometric and hyperbolic expressions, the same function can convert in many different directions, e.g. $\sin(2x) \leftrightarrow 2 \sin(x) \cos(x)$. The user has the possibility to give several parameters to the operator `trigsimp` in order to influence the transformations. It is possible to decide whether or not a rational expression involving trigonometric and hyperbolic functions vanishes.

To simplify an expression `f`, one uses `trigsimp(f[,options])`. For example:

```
trigsimp(sin(x)^2+cos(x)^2);
```

```
1
```

The possible options (where * denotes the default) are:

1. `sin*` or `cos`;
2. `sinh*` or `cosh`;
3. `expand*`, `combine` or `compact`;

4. `hyp, trig or expon`;
5. `keepalltrig`;
6. `tan and/or tanh`;
7. target arguments of the form *variable / positive integer*.

From each of the first four groups one can use at most one option, otherwise an error message will occur. Options can be given in any order.

The first group fixes the preference used while transforming a trigonometric expression:

```
trigsimp(sin(x)^2);
```

$$\sin^2(x)$$

```
trigsimp(sin(x)^2, cos);
```

$$-\cos^2(x) + 1$$

The second group is the equivalent for the hyperbolic functions.

The third group determines the type of transformation. With the default, `expand`, an expression is transformed to use only simple variables as arguments:

```
trigsimp(sin(2x+y));
```

$$2\cos(x)\cos(y)\sin(x) - 2\sin(x)\sin(y)\sin(x) + \sin(y)$$

With `combine`, products of trigonometric functions are transformed to trigonometric functions involving sums of variables:

```
trigsimp(sin(x)*cos(y), combine);
```

$$\frac{\sin(x-y) + \sin(x+y)}{2}$$

With `compact`, the REDUCE operator `compact` [2] is applied to `f`. This often leads to a simple form, but in contrast to `expand` one does not get a normal form. For example:

```
trigsimp((1-sin(x)^2)^20*(1-cos(x)^2)^20, compact);
```

$$\cos(x)^{40} \sin(x)^{40}$$

With an option from the fourth group, the input expression is transformed to trigonometric, hyperbolic or exponential form respectively:

```
trigsimp(sin(x), hyp);
```

$$- \sinh(ix) \cdot i$$

```
trigsimp(sinh(x), expon);
```

$$\frac{e^{2x} - 1}{2e^x}$$

```
trigsimp(e^x, trig);
```

$$\cos(ix) - \sin(ix) \cdot i$$

Usually, \tan , \cot , \sec , \csc are expressed in terms of \sin and \cos . It can sometimes be useful to avoid this, which is handled by the option `keepalltrig`:

```
trigsimp(tan(x+y), keepalltrig);
```

$$\frac{-(\tan(x) + \tan(y))}{\tan(x) \tan(y) - 1}$$

Alternatively, the options `tan` and/or `tanh` can be given to convert the output to the specified form as far as possible:

```
trigsimp(tan(x+y), tan);
```

$$\frac{-(\tan(x) + \tan(y))}{\tan(x) \tan(y) - 1}$$

By default, the other functions used will be `cos` and/or `cosh`, unless the other desired functions are also specified in which case this choice will be respected.

The final possibility is to specify additional target arguments for the trigonometric or hyperbolic functions, each of which should have the form of a variable divided by a positive integer. These additional arguments are treated as if they had occurred within the expression to be simplified, and their denominators are used in determining the overall denominator to use for each variable in the simplified form:

```
trigsimp(csc x - cot x + csc y - cot y, x/2, y/2, tan);
```

$$\tan\left(\frac{x}{2}\right) + \tan\left(\frac{y}{2}\right)$$

It is possible to use the options of different groups simultaneously:

```
trigsimp(sin(x)^4, cos, combine);
```

$$\frac{\cos(4x) - 4\cos(2x) + 3}{8}$$

Sometimes, it is necessary to handle an expression in separate steps:

```
trigsimp((sinh(x)+cosh(x))^n+(cosh(x)-sinh(x))^n, expon);
```

$$\left(\frac{1}{e^x}\right)^n + e^{n \cdot x}$$

```
trigsimp(ws, hyp);
```

$$2 \cdot \cosh(n \cdot x)$$

```
trigsimp((cosh(a*n)*sinh(a)*sinh(p)+cosh(a)*sinh(a*n)*sinh(p)+
sinh(a - p)*sinh(a*n))/sinh(a));
```

$$\cosh(a \cdot n) \cdot \sinh(p) + \cosh(p) \cdot \sinh(a \cdot n)$$

```
trigsimp(ws, combine);
```

$$\sinh(a \cdot n + p)$$

The `trigsimp` operator can be applied to equations, lists and matrices (and compositions thereof) as well as scalar expressions, and automatically maps itself re-

cursively over such non-scalar data structures:

```
trigsimp( { sin(2x) = cos(2x) } );
```

$$\{2*\cos(x)*\sin(x) = -2*\sin(x)^2 + 1\}$$

16.62.3 Factorizing trigonometric expressions

With `trigfactorize(p, x)` one can factorize the trigonometric or hyperbolic polynomial `p` in terms of trigonometric functions of the argument `x`. The output has the same format as that from the standard REDUCE operator `factorize`. For example:

```
trigfactorize(sin(x), x/2);
```

$$\{\{2, 1\}, \{\sin(\frac{x}{2}), 1\}, \{\cos(\frac{x}{2}), 1\}\}$$

If the polynomial is not coordinated or balanced [1], the output will equal the input. In this case, changing the value for `x` can help to find a factorization, e.g.

```
trigfactorize(1+cos(x), x);
```

```
\{\cos(x) + 1, 1\}
```

```
trigfactorize(1+cos(x), x/2);
```

$$\{\{2, 1\}, \{\cos(\frac{x}{2}), 2\}\}$$

The polynomial can consist of both trigonometric and hyperbolic functions:

```
trigfactorize(sin(2x)*sinh(2x), x);
```

```
\{\{4, 1\}, \{\sinh(x), 1\}, \{\cosh(x), 1\}, \{\sin(x), 1\}, \{\cos(x), 1\}\}
```

The `trigfactorize` operator respects the standard REDUCE `factorize` switch `nopowers` – see the REDUCE manual for details. Turning it on gives the behaviour that was standard before REDUCE 3.7:

```
on nopowers;

trigfactorize(1+cos(x), x/2);
```

$$\{2, \cos\left(\frac{x}{2}\right), \cos\left(\frac{x}{2}\right)\}$$

16.62.4 GCDs of trigonometric expressions

The operator `triggcd` is essentially an application of the algorithm behind `trigfactorize`. With its help the user can find the greatest common divisor of two trigonometric or hyperbolic polynomials. It uses the method described in [1]. The syntax is `triggcd(p, q, x)`, where `p` and `q` are the trigonometric polynomials and `x` is the argument to use. For example:

```
triggcd(sin(x), 1+cos(x), x/2);
```

$$\cos\left(\frac{x}{2}\right)$$

```
triggcd(sin(x), 1+cos(x), x);
```

```
1
```

The polynomials p and q can consist of both trigonometric and hyperbolic functions:

```
triggcd(sin(2x)*sinh(2x), (1-cos(2x))*(1+cosh(2x)), x);
```

```
cosh(x)*sin(x)
```

16.62.5 Further Examples

With the help of this package the user can create identities:

```
trigsimp(tan(x)*tan(y));
```

$$\frac{\sin(x)*\sin(y)}{\cos(x)*\cos(y)}$$

```
trigsimp(ws, combine);
```

$$\frac{\cos(x - y) - \cos(x + y)}{\cos(x - y) + \cos(x + y)}$$

```
trigsimp((sin(x-a)+sin(x+a))/(cos(x-a)+cos(x+a)));
```

$$\frac{\sin(x)}{\cos(x)}$$

```
trigsimp(cosh(n*acosh(x))-cos(n*acos(x)), trig);
```

```
0
```

```
trigsimp(sec(a-b), keepalltrig);
```

$$\frac{\csc(a) \csc(b) \sec(a) \sec(b)}{\csc(a) \csc(b) + \sec(a) \sec(b)}$$

```
trigsimp(tan(a+b), keepalltrig);
```

$$\frac{-(\tan(a) + \tan(b))}{\tan(a) \tan(b) - 1}$$

```
trigsimp(ws, keepalltrig, combine);
```

```
tan(a + b)
```

Some difficult expressions can be simplified:

```
df(sqrt(1+cos(x)), x, 4);
```

$$\begin{aligned} & (-4\cos^5(x) - 4\cos^4(x) - 20\cos^3(x)\sin(x) + 12\cos^3(x) \\ & - 24\cos^2(x)\sin^2(x) + 20\cos^2(x) - 15\cos(x)\sin^4(x) \\ & \qquad \qquad \qquad 2 \qquad \qquad \qquad 4 \qquad \qquad \qquad 2 \end{aligned}$$

```

+ 12*cos(x)*sin(x) + 8*cos(x) - 15*sin(x) + 16*sin(x) )/
(16*sqrt(cos(x) + 1)

      4      3      2
*(cos(x)  + 4*cos(x)  + 6*cos(x)  + 4*cos(x) + 1))

on rationalize;
trigsimp(ws);

      sqrt(cos(x) + 1)
-----
      16

off rationalize;
load_package taylor;

taylor(sin(x+a)*cos(x+b), x, 0, 4);

cos(b)*sin(a) + (cos(a)*cos(b) - sin(a)*sin(b))*x

      2
- (cos(a)*sin(b) + cos(b)*sin(a))*x

      2*( - cos(a)*cos(b) + sin(a)*sin(b))      3
+ -----*x
      3

      cos(a)*sin(b) + cos(b)*sin(a)      4      5
+ -----*x + O(x )
      3

trigsimp(ws, combine);

      sin(a - b) + sin(a + b)
----- + cos(a + b)*x - sin(a + b)*x      2
      2

      2*cos(a + b)      3      sin(a + b)      4      5
- -----*x + -----*x + O(x )
      3      3

```

Certain integrals whose evaluation was not possible in REDUCE (without preprocessing) are now computable:

```
int(trigsimp(sin(x+y)*cos(x-y)*tan(x)), x);
```

$$\frac{(\cos(x)^2 x - \cos(x) \sin(x) - 2 \cos(y) \log(\cos(x)) \sin(y) + \sin(x)^2 x)/2}{2}$$

```
int(trigsimp(sin(x+y)*cos(x-y)/tan(x)), x);
```

$$\frac{(\cos(x) \sin(x) - 2 \cos(y) \log(\tan(\frac{x}{2}) + 1) \sin(y) + 2 \cos(y) \log(\tan(\frac{x}{2})) \sin(y) + x)/2}{2}$$

Without the package, the integration fails, and in the second case one does not receive an answer for many hours.

```
trigfactorize(sin(2x)*cos(y)^2, y/2);
```

```
{{2*cos(x)*sin(x), 1},
```

$$\{\cos(\frac{y}{2}) - \sin(\frac{y}{2}), 2\},$$

$$\{\cos(\frac{y}{2}) + \sin(\frac{y}{2}), 2\}$$

```
trigfactorize(sin(y)^4-x^2, y);
```

$$\{\{\sin(y)^2 + x, 1\}, \{\sin(y)^2 - x, 1\}\}$$

```
trigfactorize(sin(x)*sinh(x), x/2);
```

```
{{4, 1},
```

```
 x
```

$$\{\sinh(\frac{x}{2}), 1\},$$

$$\{\cosh(\frac{x}{2}), 1\},$$

$$\{\sin(\frac{x}{2}), 1\},$$

$$\{\cos(\frac{x}{2}), 1\}$$

$$\text{triggcd}(-5+\cos(2x)-6\sin(x), -7+\cos(2x)-8\sin(x), x/2);$$

$$2*\cos(\frac{x}{2})*\sin(\frac{x}{2}) + 1$$

$$\text{triggcd}(1-2\cosh(x)+\cosh(2x), 1+2\cosh(x)+\cosh(2x), x/2);$$

$$2*\sinh(\frac{x}{2})^2 + 1$$

Bibliography

- [1] Roach, Kelly: Difficulties with Trigonometrics. Notes of a talk.
- [2] Hearn, A.C.: COMPACT User Manual.

16.63 TURTLE: Turtle Graphics Interface for REDUCE

Author: Caroline Cotter

This package is a simple implementation of the “Turtle Graphics” style of drawing graphs in REDUCE. The background and ideas of “Turtle Graphics” are outlined below.

16.63.1 Turtle Graphics

Turtle Graphics was originally developed in the 1960’s as part of the LOGO system, and used in the classroom as an introduction to graphics and using computers to help with mathematics.

The LOGO language was created as part of an experiment to test the idea that programming may be used as an educational discipline to teach children. It was first intended to be used for problem solving, for illustrating mathematical concepts usually difficult to grasp, and for creation of experiments with abstract ideas.

At first LOGO had no graphics capabilities, but fast development enabled the incorporation of graphics, known as “Turtle Graphics” into the language. “Turtle Graphics” is regarded by many as the main use of LOGO.

Main Idea: To use simple commands directing a turtle, such as forward, back, turnleft, in order to construct pictures as opposed to drawing lines connecting cartesian coordinate points.

The ‘turtle’ is at all times determined by its state $\{x, y, a, p\}$ - where x, y determine its position in the (x, y) -plane, a determines the angle (which describes the direction the turtle is facing) and p signals whether the pen is up or down (i. e. whether or not it is drawing on the paper).

16.63.2 Implementation

Some alterations to the original “Turtle Graphics” commands have been made in this implementation due to the design of the graphics package *gnuplot* used in REDUCE.

- It is not possible to draw lines individually and to see each separate line as it is added to the graph since *gnuplot* automatically replaces the last graph each time it calls on the plot function.

Thus the whole sequence of commands must be input together if the complete picture is to be seen.

- This implementation does not make use of the standard turtle commands ‘pen-up’ or ‘pen-down’ . Instead, ‘set’ commands are included which allow the turtle to move without drawing a line.
- No facility is provided here to change the pen-colour, but gnuplot does have the capability to handle a few different colours (which could be included later).
- Many of the commands are long and difficult to type out repeatedly, therefore all the commands included under ‘*Turtle Functions*’ (below) are listed alongside an equivalent abbreviated form.
- The user has no control over the range of output that can be seen on the screen since the gnuplot program automatically adjusts the picture to fit the window. Hence the size of each specified ‘step’ the turtle takes in any direction is not a fixed unit of length, rather it is relative to the scale chosen by gnuplot.

16.63.3 Turtle Functions

As previously mentioned, the turtle is determined at all times by its state $\{x,y,a\}$: its position on the (x,y) -plane and its angle(a) - its *heading* - which determines the direction the turtle is facing, in degrees, relative anticlockwise to the positive x-axis.

User Setting Functions

setheading Takes a number as its argument and resets the heading to this number. If the number entered is negative or greater than or equal to 360 then it is automatically checked to lie between 0 and 360.

Returns the turtle position $\{x,y\}$

SYNTAX: `setheading(θ)`

Abbreviated form: `sh(θ)`

leftturn The turtle is turned anticlockwise through the stated number of degrees. Takes a number as its argument and resets the heading by adding this number to the previous heading setting.

Returns the turtle position $\{x,y\}$

SYNTAX: `leftturn(α)`

Abbreviated form: `slt(α)`

rightturn Similar to `leftturn`, but the turtle is turned clockwise through the stated number of degrees. Takes a number as its argument and resets the heading by subtracting this number from the previous heading setting.

Returns the turtle position $\{x,y\}$

SYNTAX: `rightturn(β)`

Abbreviated form: `srt(β)`

setx Relocates the turtle in the x direction. Takes a number as its argument and repositions the state of the turtle by changing its x-coordinate.

Returns $\{\}$

SYNTAX: `setx(x)`

Abbreviated form: `sx(x)`

sety Relocates the turtle in the y direction. Takes a number as its argument and repositions the state of the turtle by changing its y-coordinate.

Returns $\{\}$

SYNTAX: `sety(y)`

Abbreviated form: `sy(y)`

setposition Relocates the turtle from its current position to the new cartesian coordinate position described. Takes a pair of numbers as its arguments and repositions the state of the turtle by changing the x and y coordinates.

Returns $\{\}$

SYNTAX: `setposition(x,y)`

Abbreviated form: `spn(x,y)`

setheadingtowards Resets the heading so that the turtle is facing towards the given point, with respect to its current position on the coordinate axes. Takes a pair of numbers as its arguments and changes the heading, but the turtle stays in the same place.

Returns the turtle position $\{x,y\}$

SYNTAX: `setheadingtowards(x,y)`

Abbreviated form: `shto(x,y)`

setforward Relocates the turtle from its current position by moving forward (in the direction of its heading) the number of steps given. Takes a number as its argument and repositions the state of the turtle by changing the x and y coordinates.

Returns $\{\}$

SYNTAX: `setforward(n)`

Abbreviated form: `sfwd(n)`

setback As with `setforward`, but moves back (in the opposite direction of its heading) the number of steps given.

Returns {}

SYNTAX: `setback(n)`

Abbreviated form: `sbk(n)`

Line-Drawing Functions

forward Moves the turtle forward (in the direction its heading) the number of steps given. Takes a number as its argument and draws a line from its current position to a new position on the coordinate plane. The x and y coordinates are reset to the new values.

Returns the list of points { {*old x,old y*}, {*new x,new y*} }

SYNTAX: `forward(s)`

Abbreviated form: `fwd(s)`

back As with `forward` except the turtle moves back (in the opposite direction to its heading) the number of steps given.

Returns the list of points { {*old x,old y*}, {*new x,new y*} }

SYNTAX: `back(s)`

Abbreviated form: `bk(s)`

move Moves the turtle to a specified point on the coordinate plane. Takes a pair of numbers as its arguments and draws a line from its current position to the position described. The x and y coordinates are set to these new values.

Returns the list of points { {*old x,old y*}, {*new x,new y*} }

SYNTAX: `move(x,y)`

Abbreviated form: `mv(x,y)`

Plotting Functions

draw This is the function the user calls within REDUCE to draw the list of turtle commands given into a picture. Takes a list as its argument, with each separate command being separated by a comma, and returns the graph drawn by following the commands.

SYNTAX: `draw{command(command_args), ..., command(command_args)}`

Note: all commands may be entered in either long or shorthand form, and with a space before the arguments instead of parentheses only if just one argument is needed. Commands taking more than one argument must be written in parentheses and arguments separated by a comma.

fdraw This function is also called in REDUCE by the user and outputs the same as the `draw` command, but it takes a filename as its argument. The file which is called upon by `fdraw` must contain only the turtle commands and other functions defined by the user for turtle graphics. (This is intended to make it easier for the user to make small changes without constantly typing out long series of commands.)

SYNTAX: `fdraw{"filename"}` Note: commands may be entered in long or shorthand form but each command must be written on a separate line of the file. Also, arguments are to be written without parentheses and separated with a space, not a comma, regardless of the number of arguments given to the function.

Other Important Functions

info This function is called on its own in REDUCE to tell user the current state of the turtle. Takes no arguments but returns a list containing the current values of the x and y coordinates and the heading variable.

Returns the list $\{x_coord, y_coord, heading\}$

SYNTAX: `info()` or simply `info`

clearscreen This is also called on its own in REDUCE to get rid of the last gnuplot window, displaying the last turtle graphics picture, and to reset all the variables to 0. Takes no arguments and returns no printed output to the screen but the graphics window is simply cleared.

SYNTAX: `clearscreen()` or simply `clearscreen`

Abbreviated form: `cls()` or `cls`

home This is a command which can be called within a plot function as well as outside of one. Takes no arguments, and simply resets the x and y coordinates and the heading variable to 0. When used in a series of turtle commands, it moves the turtle from its current position to the origin and sets the direction of the turtle along the x-axis, without drawing a line.

Returns $\{0,0\}$

SYNTAX: `home()` or simply `home`

Defining Functions

It is possible to use conditional statements (if ... then ... else ...) and 'for' statements (for i:=... collect{...}) in calls to `draw`. However, care must be taken - when using conditional statements the final else statement must return a point or at least

{x_coord,y_coord} if the picture is to be continued at that point. Also, 'for' statements *must* include 'collect' followed by a list of turtle commands (in addition, the variable must begin counting from 0 if it is to be joined to the previous list of turtle commands at that point exactly, e. g. for i:=0:10 collect {...}).

SYNTAX: { (For user-defined Turtle functions)}

```

procedure func_name(func_args);
begin [scalar additional variables];
  :
  (the procedure body containing some turtle commands)
  :
  return (a list, or label to a list, of turtle commands
         as accepted by draw)
end;
```

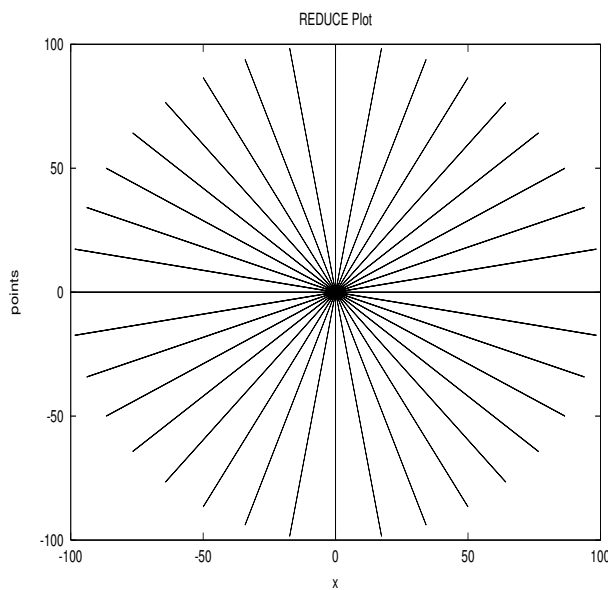
For convenience, it is recommended that all user defined functions, such as those involving if...then...else... or for i:=...collect{...} are defined together in a separate file, then called into REDUCE using the in "filename" command.

16.63.4 Examples

The following examples are taken from the tur.tst file. Examples 1,2,5 & 6 are simple calls to draw. Examples 3 & 4 show how more complicated commands can be built (which can take their own set of arguments) using procedures. Examples 7 & 8 show the difference between the draw and fdraw commands.

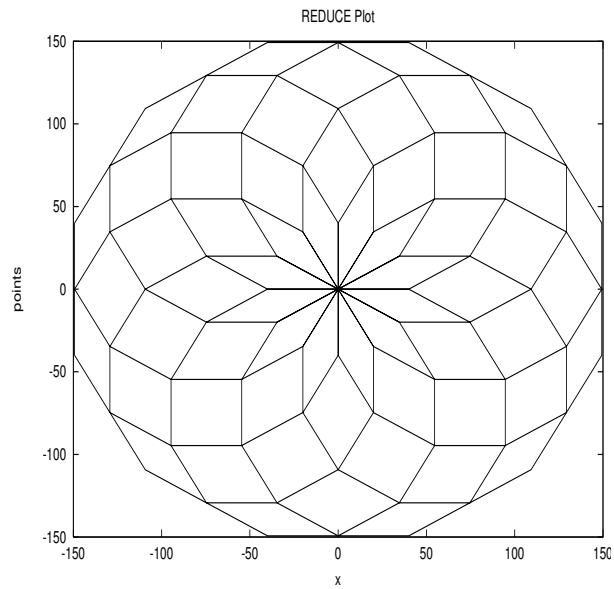
```
% (1) Draw 36 rays of length 100
```

```
draw {for i:=1:36 collect{setheading(i*10), forward 100, back 100} };
```



```
% (2) Draw 12 regular polygons with 12 sides of length 40,each polygon
%forming an angle of 360/n degrees with the previous one.
```

```
draw {for i:=1:12 collect
      {leftturn(30), for j:=1:12 collect
        {forward 40, leftturn(30)}} };
```



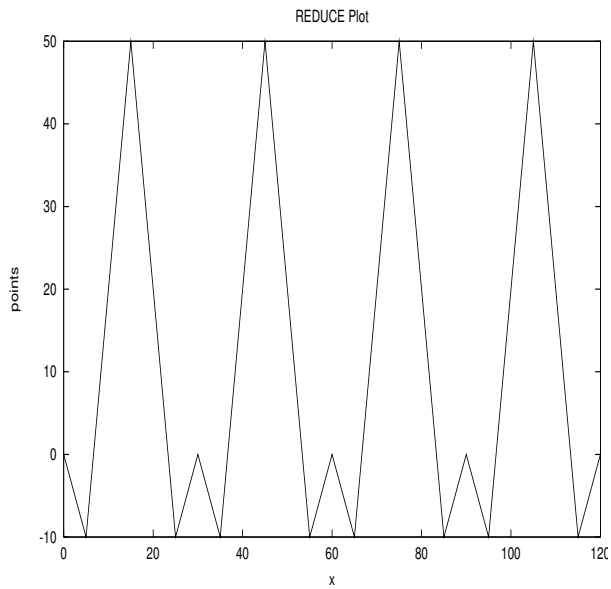
% (3) A "peak" pattern - an example of a recursive procedure.

```

procedure peak(r);
begin;
  return for i:=0:r collect
    {move(x_coord+5,y_coord-10), move(x_coord+10,y_coord+60),
     move(x_coord+10,y_coord-60),move(x_coord+5,y_coord+10)};
end;

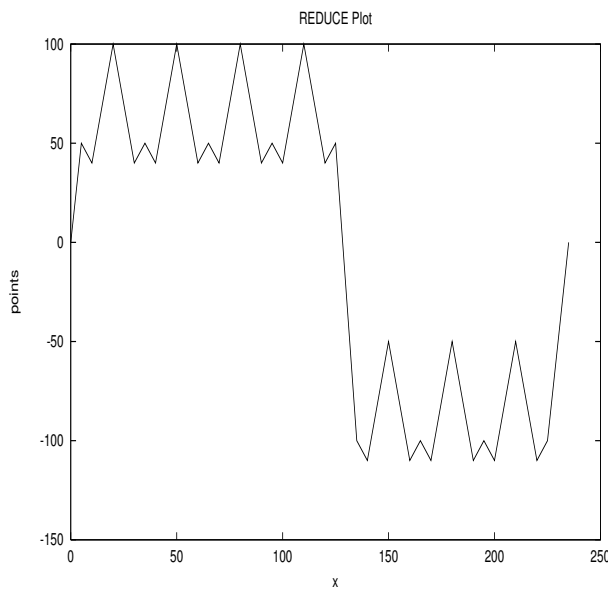
draw {home(), peak(3)};

```



%This procedure can then be part of a longer chain of commands:

```
draw {home(), move(5,50), peak(3), move(x_coord+10,-100),
      peak(2), move(x_coord+10,0)};
```



% (4) Write a recursive procedure which draws "trees" such that every
%branch is half the length of the previous branch.

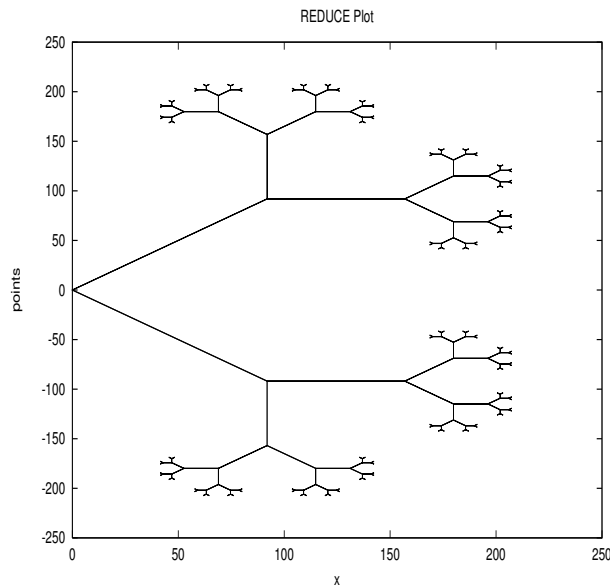
```

procedure tree(a,b);          %Here: a is the start length, b is the
                                %number of levels
begin;
    return if fixpb and b>0    %checking b is a positive integer

        then {leftturn(45), forward a, tree(a/2,b-1),
              back a, rightturn(90), forward a, tree(a/2,b-1),
              back a, leftturn(45)}
        else {x_coord,y_coord}; %default: Turtle stays still
end;

draw {home(), tree(130,7)};

```

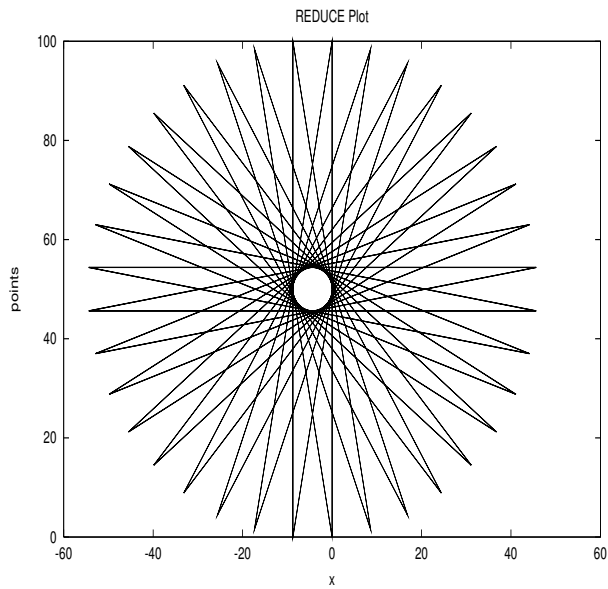


% (5) A 36-point star.

```

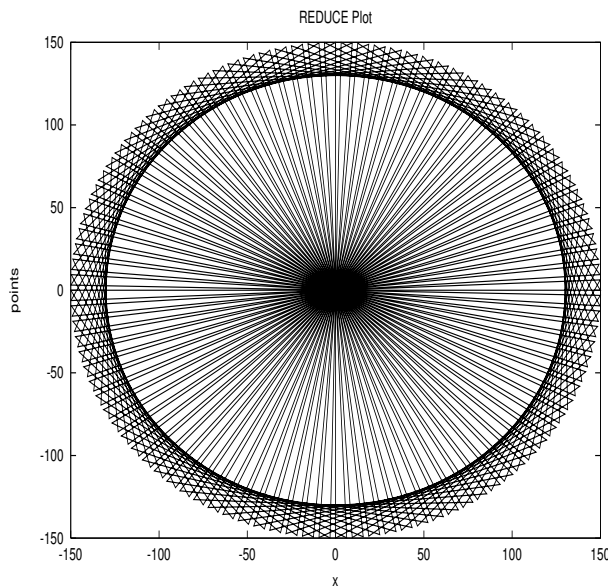
draw {home(), for i:=1:36 collect
    {leftturn(10), forward 100, leftturn(10), back 100}

```

% (6) Draw 100 equilateral triangles with the leading points
%equally spaced on a circular path.

```
draw {home(), for i:=1:100 collect
      {forward 150, rightturn(60), back(150),
       rightturn(60), forward 150, setheading(i*3.6)} };
```

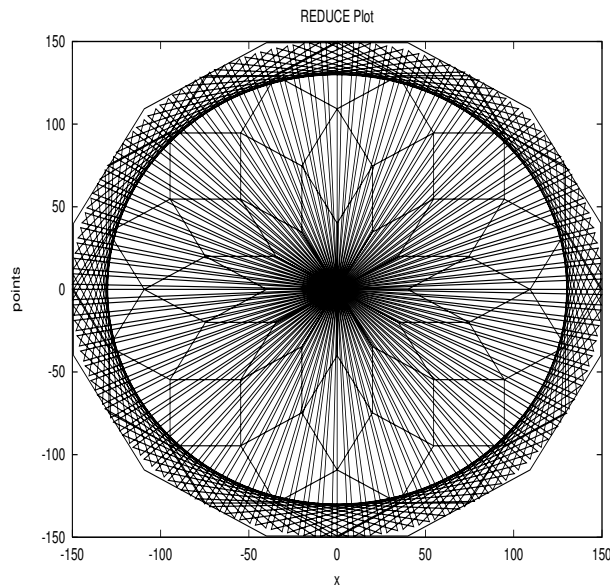


% (7) Two or more graphs can be drawn together (this is easier
 %if the graphs are named). Here we show graphs 2 and 6 on top of one
 %another:

```
gr2:={home(), for i:=1:12 collect
      {leftturn(30), for j:=1:12 collect
        {forward 40, leftturn(30)}} }$

gr6:={home(), for i:=1:100 collect
      {forward 150, rightturn(60), back(150),
       rightturn(60), forward 150, setheading(i*3.6)} }$

draw {gr2, gr6};
```



% (8) Example 7 could have been tackled another way, which makes use of
 %the fdraw command.
 %By inputting gr2 and gr6 as procedures into reduce, they can then be
 %used at any time in the same reduce session in a call to draw and even
 %fdraw.

%First save the procedures in a file, say fxp (fdraw example procedures)

```
procedure gr2;
begin;
  return {home, for i:=1:12 collect
```

```

                                {leftturn(30), for j:=1:12 collect
                                                                {forward 40, leftturn(30)}} };
end;

procedure gr6;
begin;
    return {home(), for i:=1:100 collect
                                {forward 150, rightturn(60), back(150),
                                rightturn(60), forward 150, setheading(i*3.6)} };
end;

%Then create another file where the functions may be called to fdraw,
%e.g. fx:

gr2
gr6

%Now in reduce, after loading the turtle package just type the following:

in "fxp";
fdraw "fx";

%..and the graphs will appear.

%This method is useful if the user wants to define many of their own
%functions, and, using fdraw, subtle changes can be made quickly without
%having to type out the whole string of commands to plot each time. It
%is particularly useful if there are several pictures to plot at once and
%it is an easy way to build pictures so that the difference an extra
%command makes to the overall picture can be clearly seen.
%(In the above example, the file called to fdraw was only 2 lines long,
%so this method did not have any advantage over the normal draw command.
%However, when the list of commands is longer it is clearly advantageous
%to use fdraw)

```

16.63.5 References

1. **An Implementation of Turtle Graphics for Teaching Purposes**
Zoran I. Putnik & Zoran d.Budimac
2. **Mapletech - Maple in Mathematics and the Sciences,**
Special Issue 1994

An Implementation of “Turtle Graphics” in Maple V

Eugenio Roanes Lozano & Eugenio Roanes Macias

16.64 WU: Wu algorithm for polynomial systems

This is a simple implementation of the Wu algorithm implemented in REDUCE working directly from “A Zero Structure Theorem for Polynomial-Equations-Solving,” Wu Wen-tsun, Institute of Systems Science, Academia Sinica, Beijing.

Author: Russell Bradford.

This is a simple implementation of the Wu algorithm implemented in Reduce 3.3, working directly from “A Zero Structure Theorem for Polynomial-Equations-Solving,” Wu Wen-tsun, Institute of Systems Science, Academia Sinica, Beijing.

Its purpose was to aid my understanding of the algorithm, so the code is simple, and has a lot of tracing included. This is a working implementation, but there is magnificent scope for improvement and optimisation. Things like using intelligent sorts on polynomial lists, and avoiding the re-computation of various data spring easily to mind. Also, an attempt at factorization of the input polynomials at each pass might have beneficial results. Of course, exploitation of the natural parallel structure is a must!

All bug fixes and improvements are welcomed.

The interface:

```
wu( {x^2+y^2+z^2-r^2, x*y+z^2-1, x*y*z-x^2-y^2-z+1}, {x,y,z});
```

calls wu with the named polynomials, and with the variable ordering $x > y > z$. In this example, r is a parameter.

The result is

$$\begin{aligned} & \{ \{ r^2 + z^3 - z^2 - 1, \\ & r^2 * y^2 + r^2 * z^2 + r^2 - y^4 - y^2 * z^2 + z^2 - z - 2, \\ & x * y + z^2 - 1 \}, \\ & y \}, \\ & \{ \{ r^6 * z^4 - 2 * r^6 * z^2 + r^6 + 3 * r^4 * z^7 - 3 * r^4 * z^6 - 6 * r^4 * z^5 + 3 * r^4 * z^4 + 3 * \\ & r^4 * z^3 + 3 * r^4 * z^2 - 3 * r^4 + 3 * r^2 * z^{10} - 6 * r^2 * z^9 - 3 * r^2 * z^8 + 6 * r^2 * z^7 + \end{aligned}$$

$$\begin{aligned}
& 3r^2z^6 + 6r^2z^5 - 6r^2z^4 - 6r^2z^3 + 3r^2 + z^{13} - 3z^{12} + z^{11} \\
& + 2z^{10} + z^9 + 2z^8 - 6z^7 - z^6 + 2z^4 + 3z^3 - z^2 - 1, \\
& y^2(r^2 + z^3 - z^2 - 1), \\
& xy + z^2 - 1, \\
& y(r^2 + z^3 - z^2 - 1) \} \}
\end{aligned}$$

namely, a list of pairs of characteristic sets and initials for the characteristic sets.

Thus, the first pair above has the characteristic set

$$r^2 + z^3 - z^2 - 1, r^2y^2 + r^2z + r^2 - y^4 - y^2z^2 + z^2 - z - 2, xy + z^2 - 1$$

and initial y .

According to Wu's theorem, the set of roots of the original polynomials is the union of the sets of roots of the characteristic sets, with the additional constraints that the corresponding initial is non-zero. Thus, for the first pair above, we find the roots of $\{r^2 + z^3 - z^2 - 1, \dots\}$ under the constraint that $y \neq 0$. These roots, together with the roots of the other characteristic set (under the constraint of $y(r^2 + z^3 - z^2 - 1) \neq 0$), comprise all the roots of the original set.

Additional information about the working of the algorithm can be gained by

```
on trwu;
```

This prints out details of the choice of basic sets, and the computation of characteristic sets.

The second argument (the list of variables) may be omitted, when all the variables in the input polynomials are implied with some random ordering.

16.65 XCOLOR: Color factor in some field theories

This package calculates the color factor in non-abelian gauge field theories using an algorithm due to Cvitanovich.

Documentation for this package is in plain text.

Author: A. Kryukov.

Program "xCOLOR" is intended for calculation the colour factor in non-abelian gauge field theories. It is realized Cvitanovich algorithm [1]. In comparison with "COLOR" program [2] it was made many improvements. The package was written by symbolic mode. This version is faster then [2] more then 10 times.

After load the program by the following command `load xcolor;` user can be able to use the next additional commands and operators.

Command SUDim.

Format: `SUDim <any expression>;`

Set the order of SU group.

The default value is 3, i.e. SU(3).

Command SpTT.

Format: `SpTT <any expression>;`

Set the normalization coefficient A: $\text{Sp}(\text{TiTj}) = A * \text{Delta}(i,j)$. Default value is 1/2.

Operator QG.

Format: `QG(inQuark,outQuark,Gluon)`

Describe the quark-gluon vertex. Parameters may be any identifiers. First and second of then must be in- and out- quarks correspondently. Third one is a gluon.

Operator G3.

Format: `G3(Gluon1,Gluon2,Gluon3)`

Describe the three-gluon vertex. Parameters may be any identifiers. The order of gluons must be clock.

In terms of QG and G3 operators you input diagram in "color" space as a product of these operators. For example.

Diagram:

REDUCE expression:

$$\begin{array}{c}
 e1 \\
 \text{---->----} \\
 / \qquad \qquad \backslash \\
 | \qquad e2 \qquad | \\
 v1 * \text{.....} * v2 \\
 | \qquad \qquad | \\
 \backslash \qquad e3 \qquad / \\
 \text{----<----}
 \end{array}
 \quad \Longleftrightarrow \quad
 QG(e3, e1, e2) * QG(e1, e3, e2)$$

Here: --->--- quark

..... gluon

More detail see [2].

References.

[1] P.Cvitanovic, Phys. Rev. D14(1976), p.1536.

[2] A.Kryukov & A.Rodionov, Comp. Phys. Comm., 48(1988), pp.327-334.

Please send any remarks to my address above!

Good luck!

16.66 XIDEAL: Gröbner Bases for exterior algebra

XIDEAL constructs Gröbner bases for solving the left ideal membership problem: Gröbner left ideal bases or GLIBs. For graded ideals, where each form is homogeneous in degree, the distinction between left and right ideals vanishes. Furthermore, if the generating forms are all homogeneous, then the Gröbner bases for the non-graded and graded ideals are identical. In this case, XIDEAL is able to save time by truncating the Gröbner basis at some maximum degree if desired.

Author: David Hartley.

16.66.1 Description

The method of Gröbner bases in commutative polynomial rings introduced by Buchberger (e.g. [1]) is a well-known and very important tool in polynomial ideal theory, for example in solving the ideal membership problem. XIDEAL extends the method to exterior algebras using algorithms from [2] and [3].

There are two main departures from the commutative polynomial case. First, owing to the non-commutative product in exterior algebras, ideals are no longer automatically two-sided, and it is necessary to distinguish between left and right ideals. Secondly, because there are zero divisors, confluent reduction relations are no longer sufficient to solve the ideal membership problem: a unique normal form for every polynomial does not guarantee that all elements in the ideal reduce to zero. This leads to two possible definitions of Gröbner bases as pointed out by Stokes [4].

XIDEAL constructs Gröbner bases for solving the left ideal membership problem: Gröbner left ideal bases or GLIBs. For graded ideals, where each form is homogeneous in degree, the distinction between left and right ideals vanishes. Furthermore, if the generating forms are all homogeneous, then the Gröbner bases for the non-graded and graded ideals are identical. In this case, XIDEAL is able to save time by truncating the Gröbner basis at some maximum degree if desired.

XIDEAL uses the exterior calculus package EXCALC of E. Schrüfer [5] to provide the exterior algebra definitions. EXCALC is loaded automatically with XIDEAL. The exterior variables may be specified explicitly, or extracted automatically from the input polynomials. All distinct exterior variables in the input are assumed to be linearly independent – if a dimension has been fixed (using the EXCALC `spacedim` or `coframe` statements), then input containing distinct exterior variables with degrees totaling more than this number will generate an error.

16.66.2 Declarations

xorder

`xorder` sets the term ordering for all other calculations. The syntax is

```
xorder k
```

where `k` is one of `lex`, `gradlex` or `deglex`. The lexicographical ordering `lex` is based on the prevailing EXCALC kernel ordering for the exterior variables. The EXCALC kernel ordering can be changed with the `REDUCE korder` or EXCALC `forder` declarations. The graded lexicographical ordering `gradlex` puts terms with more factors first (irrespective of their exterior degrees) and sorts terms of the same grading lexicographically. The degree lexicographical ordering `deglex` takes account of the exterior degree of the variables, putting highest degree first and then sorting terms of the same degree lexicographically. The default ordering is `deglex`.

xvars

It is possible to consider scalar and 0-form variables in exterior polynomials in two ways: as variables or as coefficient parameters. This difference is crucial for Gröbner basis calculations. By default, all scalar variables which have been declared as 0-forms are treated as exterior variables, along with any EXCALC kernels of degree 0. This division can be changed with the `xvars` declaration. The syntax is

```
xvars U,V,W,...
```

where the arguments are either kernels or lists of kernels. All variables specified in the `xvars` declaration are treated as exterior variables in subsequent XIDEAL calculations with exterior polynomials, and any other scalars are treated as parameters. This is true whether or not the variables have been declared as 0-forms. The declaration

```
xvars {}
```

causes all degree 0 variables to be treated as parameters, and

```
xvars nil
```

restores the default. Of course, p -form kernels with $p \neq 0$ are always considered as exterior variables. The order of the variables in an `xvars` declaration has no

effect on the REDUCE kernel ordering or XIDEAL term ordering.

16.66.3 Operators

xideal

`xideal` calculates a Gröbner left ideal basis in an exterior algebra. The syntax is

```
xideal(S:list of forms[,V:list of kernels][,R:integer])
      :list of forms.
```

`xideal` calculates a Gröbner basis for the left ideal generated by S using the current term ordering. The resulting list can be used for subsequent reductions with `xmod` as long as the term ordering is not changed. Which 0-form variables are to be regarded as exterior variables can be specified in an optional argument V (just like an `xvars` declaration). The order of variables in V has no effect on the term ordering. If the set of generators S is graded, an optional parameter R can be given, and `xideal` produces a truncated basis suitable for reducing exterior forms of degree less than or equal to R in the left ideal. This can save time and space with large problems, but the result cannot be used for exterior forms of degree greater than R . The forms returned by `xideal` are sorted in increasing order. See also the switches `trxideal` and `xfullreduction`.

xmodideal

`xmodideal` reduces exterior forms to their (unique) normal forms modulo a left ideal. The syntax is

```
xmodideal(F:form, S:list of forms):form
```

or

```
xmodideal(F:list of forms, S:list of forms)
      :list of forms.
```

An alternative infix syntax is also available:

```
F xmodideal S.
```

`xmodideal(F, S)` first calculates a Gröbner basis for the left ideal generated by S , and then reduces F . F may be either a single exterior form, or a list of forms, and S is a list of forms. If F is a list of forms, each element is reduced, and any which vanish are deleted from the result. If the set of generators S is graded, then a

truncated Gröbner basis is calculated using the degree of F (or the maximal degree in F). See also `trxmod`.

xmod

`xmod` reduces exterior forms to their (not necessarily unique) normal forms modulo a set of exterior polynomials. The syntax is

```
xmod(F:form, S:list of forms):form
```

or

```
xmod(F:list of forms, S:list of forms):list of forms.
```

An alternative infix syntax is also available:

```
F xmod S.
```

`xmod(F, S)` reduces F with respect to the set of exterior polynomials S , which is not necessarily a Gröbner basis. F may be either a single exterior form, or a list of forms, and S is a list of forms. This operator can be used in conjunction with `xideal` to produce the same effect as `xmodideal`: for a single homogeneous form F and a set of exterior forms S , F `xmodideal` S is equivalent to F `xmod` `xideal(S, exdegree F)`. See also `trxmod`.

xauto

`xauto` autoreduces a set of exterior forms. The syntax is

```
xauto(S:list of forms):list of forms.
```

`xauto S` returns a set of exterior polynomials which generate the same left ideal, but which are in normal form with respect to each other. For linear expressions, this is equivalent to finding the reduced row echelon form of the coefficient matrix.

excoeffs

The operator `excoeffs`, with syntax

```
excoeffs(F:form):list of expressions
```

returns the coefficients from an exterior form as a list. The coefficients are always scalars, but which degree 0 variables count as coefficient parameters is controlled by the command `xvars`.

exvars

The operator `exvars`, with syntax

```
exvars(F:form):list of kernels
```

returns the exterior powers from an exterior form as a list. All non-scalar variables are returned, but which degree 0 variables count as coefficient parameters is controlled by the command `xvars`.

16.66.4 Switches

xfullreduce

on `xfullreduce` allows `xideal` and `xmodideal` to calculate reduced, monic Gröbner bases, which speeds up subsequent reductions, and guarantees a unique form for the Gröbner basis. off `xfullreduce` turns off this feature, which may speed up calculation of some Gröbner basis. `xfullreduce` is on by default.

trxideal

on `trxideal` produces a trace of the calculations done by `xideal` and `xmodideal`, showing the basis polynomials and the results of the critical element calculations. This can generate profuse amounts of output. `trxideal` is off by default.

trxmod

on `trxmod` produces a trace of reductions to normal form during calculations by XIDEAL operators. `trxmod` is off by default.

16.66.5 Examples

Suppose XIDEAL has been loaded, the switches are at their default settings, and the following exterior variables have been declared:

```
pform x=0,y=0,z=0,t=0,f(i)=1,h=0,hx=0,ht=0;
```

In a commutative polynomial ring, a single polynomial is its own Gröbner basis. This is no longer true for exterior algebras because of the presence of zero divisors, and can lead to some surprising reductions:

```
xideal {d x^d y - d z^d t};

      {d t^d z + d x^d y,

      d x^d y^d z,

      d t^d x^d y}

f(3)^f(4)^f(5)^f(6)
xmodideal {f(1)^f(2) + f(3)^f(4) + f(5)^f(6)};

0
```

The heat equation, $h_{xx} = h_t$ can be represented by the following exterior differential system.

```
S := {d h - ht*d t - hx*d x,
      d ht^d t + d hx^d x,
      d hx^d t - ht*d x^d t};
```

`xmodideal` can be used to check that the exterior differential system is closed under exterior differentiation.

```
d S xmodideal S;

{}
```

`xvars`, or a second argument to `xideal` can be used to change the division between exterior variables of degree 0 and parameters.

```
xideal {a*d x+y*d y};

      d x*a + d y*y
      {-----}
      a

xvars {a};
xideal {a*d x+y*d y};
```

```

      {d x*a + d y*y, d x^d y}

xideal({a*d x+y*d y},{a,y});

      {d x*a + d y*y,
      d x^d y*y}

xvars {};          % all 0-forms are coefficients
excoeffs(d u - (a*p - q)*d y);

      {1, - a*p + q}

exvars(d u - (a*p - q)*d y);

      {d u, d y}

xvars {p,q};      % p,q are no longer coefficients
excoeffs(d u - (a*p - q)*d y);

      { - a, 1, 1}

exvars(d u - (a*p - q)*d y);

      {d y*p, d y*q, d u}

xvars nil;

```

Non-graded left and right ideals are no longer the same:

```

d t^(d z+d x^d y) xmodideal {d z+d x^d y};

0

(d z+d x^d y)^d t xmodideal {d z+d x^d y};

- 2*d t^d z

```

Any form containing a 0-form term generates the whole ideal:

```

xideal {1 + f(1) + f(1)^f(2) + f(2)^f(3)^f(4)};

{1}

```

Bibliography

- [1] B. Buchberger, *Gröbner Bases: an algorithmic method in polynomial ideal theory*, in *Multidimensional Systems Theory* ed. N.K. Bose (Reidel, Dordrecht, 1985) chapter 6.
- [2] D. Hartley and P.A. Tuckey, *A Direct Characterisation of Gröbner Bases in Clifford and Grassmann Algebras*, Preprint MPI-Ph/93–96 (1993).
- [3] J. Apel, *A relationship between Gröbner bases of ideals and vector modules of G -algebras*, *Contemporary Math.* **131**(1992)195–204.
- [4] T. Stokes, *Gröbner bases in exterior algebra*, *J. Automated Reasoning* **6**(1990)233–250.
- [5] E. Schrüfer, *EXCALC, a system for doing calculations in the calculus of modern differential geometry, User's manual*, (The Rand Corporation, Santa Monica, 1986).

16.67 ZEILBERG: Indefinite and definite summation

This package is a careful implementation of the Gosper and Zeilberger algorithms for indefinite and definite summation of hypergeometric terms, respectively. Extensions of these algorithms are also included that are valid for ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials that are rational-linear in their arguments.

Authors: Gregor Stölting and Wolfram Koepf.

16.67.1 Introduction

This package is a careful implementation of the Gosper⁴⁴ and Zeilberger algorithms for indefinite, and definite summation of hypergeometric terms, respectively. Further, extensions of these algorithms given by the first author are covered. An expression a_k is called a *hypergeometric term* (or *closed form*), if a_k/a_{k-1} is a rational function with respect to k . Typical hypergeometric terms are ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials (Pochhammer symbols) that are integer-linear in their arguments.

The extensions of Gosper's and Zeilberger's algorithm mentioned in particular are valid for ratios of products of powers, factorials, Γ function terms, binomial coefficients, and shifted factorials that are rational-linear in their arguments.

16.67.2 Gosper Algorithm

The Gosper algorithm [1] is a *decision procedure*, that decides by algebraic calculations whether or not a given hypergeometric term a_k has a hypergeometric term antidifference g_k , i. e. $g_k - g_{k-1} = a_k$ with rational g_k/g_{k-1} , and returns g_k if the procedure is successful, in which case we call a_k *Gosper-summable*. Otherwise *no hypergeometric term antidifference exists*. Therefore if the Gosper algorithm does not return a closed form solution, it has *proved* that no such solution exists, an information that may be quite useful and important. The Gosper algorithm is the discrete analogue of the Risch algorithm for integration in terms of elementary functions.

Any antidifference is uniquely determined up to a constant, and is denoted by

$$g_k = \sum_k a_k .$$

Finding g_k given a_k is called *indefinite summation*. The antidifference operator Σ is the inverse of the downward difference operator $\nabla a_k = a_k - a_{k-1}$. There is

⁴⁴The `sum` package contains also a partial implementation of the Gosper algorithm.

an analogous summation theory corresponding to the upward difference operator $\Delta a_k = a_{k+1} - a_k$.

In case, an antidifference g_k of a_k is known, any sum

$$\sum_{k=m}^n a_k = g_n - g_{m-1}$$

can be easily calculated by an evaluation of g at the boundary points like in the integration case. Note, however, that the sum

$$\sum_{k=0}^n \binom{n}{k} \quad (16.67)$$

e. g. is not of this type since the summand $\binom{n}{k}$ depends on the upper boundary point n explicitly. This is an example of a definite sum that we consider in the next section.

Our package supports the input of powers (a^k), factorials (`factorial(k)`), Γ function terms (`gamma(a)`), binomial coefficients (`binomial(n,k)`), shifted factorials (`pochhammer(a,k) = a(a+1)⋯(a+k-1) = $\Gamma(a+k)/\Gamma(a)$`), and partially products (`prod(f,k,k1,k2)`). It takes care of the necessary simplifications, and therefore provides you with the solution of the decision problem as long as the memory or time requirements are not too high for the computer used.

16.67.3 Zeilberger Algorithm

The (fast) Zeilberger algorithm [10]–[11] deals with the *definite summation* of hypergeometric terms. Zeilberger's paradigm is to find (and return) a linear homogeneous recurrence equation with polynomial coefficients (called *holonomic equation*) for an *infinite sum*

$$s(n) = \sum_{k=-\infty}^{\infty} f(n, k),$$

the summation to be understood over all integers k , if $f(n, k)$ is a hypergeometric term with respect to both k and n . The existence of a holonomic recurrence equation for $s(n)$ is then generally guaranteed.

If one is lucky, and the resulting recurrence equation is of first order

$$p(n) s(n-1) + q(n) s(n) = 0 \quad (p, q \text{ polynomials}),$$

$s(n)$ turns out to be a hypergeometric term, and a closed form solution can be easily established using a suitable initial value, and is represented by a ratio of Pochhammer or Γ function terms if the polynomials p , and q can be factored.

Zeilberger's algorithm does not guarantee to find the holonomic equation of lowest order, but often it does.

If the resulting recurrence equation has order larger than one, this information can be used for identification purposes: Any other expression satisfying the same recurrence equation, and the same initial values, represents the same function.

Note that a *definite sum* $\sum_{k=m_1}^{m_2} f(n, k)$ is an infinite sum if $f(n, k) = 0$ for $k < m_1$ and $k > m_2$. This is often the case, an example of which is the sum (16.67) considered above, for which the hypergeometric recurrence equation $2s(n-1) - s(n) = 0$ is generated by Zeilberger's algorithm, leading to the closed form solution $s(n) = 2^n$.

Definite summation is trivial if the corresponding indefinite sum is Gosper-summable analogously to the fact that definite integration is trivial as soon as an elementary antiderivative is known. If this is not the case, the situation is much more difficult, and it is therefore quite remarkable and non-obvious that Zeilberger's method is just a clever application of Gosper's algorithm.

Our implementation is mainly based on [3] and [2]. More examples can be found in [5], [7], [8], and [9] many of which are contained in the test file `zeilberg.tst`.

16.67.4 REDUCE operator GOSPER

The ZEILBERG package must be loaded by:

```
1: load zeilberg;
```

The `gosper` operator is an implementation of the Gosper algorithm.

- `gosper(a, k)` determines a closed form antidifference. If it does not return a closed form solution, then a closed form solution does not exist.
- `gosper(a, k, m, n)` determines

$$\sum_{k=m}^n a_k$$

using Gosper's algorithm. This is only successful if Gosper's algorithm applies.

Example:

```
2: gosper((-1)^(k+1)*(4*k+1)*factorial(2*k)/
      (factorial(k)*4^k*(2*k-1)*factorial(k+1)), k);
```

$$\frac{\binom{k}{-1} \text{factorial}(2*k)}{2^{2*k} \text{factorial}(k+1) \text{factorial}(k)}$$

This solves a problem given in SIAM Review ([6], Problem 94–2) where it was asked to determine the infinite sum

$$S = \lim_{n \rightarrow \infty} S_n, \quad S_n = \sum_{k=1}^n \frac{(-1)^{k+1} (4k+1)(2k-1)!!}{2^k (2k-1)(k+1)!},$$

(($2k-1$)!! = $1 \cdot 3 \cdots (2k-1) = \frac{(2k)!}{2^k k!}$). The above calculation shows that the summand is Gosper-summable, and the limit $S = 1$ is easily established using Stirling's formula.

The implementation solves further deep and difficult problems some examples of which are:

```
3: gosper(sub(n=n+1,binomial(n,k)^2/binomial(2*n,n))-
    binomial(n,k)^2/binomial(2*n,n),k);

((binomial(n+1,k)^2*binomial(2*n,n)
- binomial(2*(n+1),n+1)*binomial(n,k)^2)*(2*k-3*n-1)
*(k-n-1)^2)/((2*(2*(n+1)-k)*(2*n+1)*k-3*n^3-7*n^2-5*n
-1)*binomial(2*(n+1),n+1)*binomial(2*n,n))

4: gosper(binomial(k,n),k);

(k+1)*binomial(k,n)
-----
n+1

5: gosper((-25+15*k+18*k^2-2*k^3-k^4)/
    (-23+479*k+613*k^2+137*k^3+53*k^4+5*k^5+k^6),k);

- (2*k^2-15*k+8)*k
-----
3      2
23*(k + 4*k + 27*k + 23)
```

The Gosper algorithm is not capable to give antidifferences depending on the harmonic numbers

$$H_k := \sum_{j=1}^k \frac{1}{j},$$

e. g. $\sum_k H_k = (k+1)(H_{k+1} - 1)$, but, is able to give a proof, instead, for the fact that H_k does not possess a closed form evaluation:

```
6: gosper(1/k, k);
```

```
***** Gosper algorithm: no closed form solution exists
```

The following code gives the solution to a summation problem proposed in Gosper's original paper [1]. Let

$$f_k = \prod_{j=1}^k (a + b j + c j^2) \quad \text{and} \quad g_k = \prod_{j=1}^k (e + b j + c j^2).$$

Then a closed form solution for

$$\sum_k \frac{f_{k-1}}{g_k}$$

is found by the definitions

```
7: operator ff, gg$
```

```
8: let {ff(~k+~m) => ff(k+m-1) * (c*(k+m)^2+b*(k+m)+a)
      when (fixp(m) and m>0),
      ff(~k+~m) => ff(k+m+1) / (c*(k+m+1)^2+b*(k+m+1)+a)
      when (fixp(m) and m<0)}$
```

```
9: let {gg(~k+~m) => gg(k+m-1) * (c*(k+m)^2+b*(k+m)+e)
      when (fixp(m) and m>0),
      gg(~k+~m) => gg(k+m+1) / (c*(k+m+1)^2+b*(k+m+1)+e)
      when (fixp(m) and m<0)}$
```

and the calculation

```
10: gosper(ff(k-1)/gg(k), k);
```

```
      ff(k)
-----
(a - e)*gg(k)
```

```
11: clear ff, gg$
```

Similarly closed form solutions of $\sum_k \frac{f_{k-m}}{g_k}$ for positive integers m can be obtained, as well as of $\sum_k \frac{f_{k-1}}{g_k}$ for

$$f_k = \prod_{j=1}^k (a + bj + cj^2 + dj^3) \quad \text{and} \quad g_k = \prod_{j=1}^k (e + bj + cj^2 + dj^3)$$

and for analogous expressions of higher degree polynomials.

16.67.5 REDUCE operator EXTENDED_GOSPER

The `extended_gosper` operator is an implementation of an extended version of Gosper's algorithm given by Koepf [2].

- `extended_gosper(a, k)` determines an antidifference g_k of a_k whenever there is a number m such that $h_k - h_{k-m} = a_k$, and h_k is an m -fold hypergeometric term, i. e.

$$h_k/h_{k-m} \text{ is a rational function with respect to } k.$$

If it does not return a solution, then such a solution does not exist.

- `extended_gosper(a, k, m)` determines an m -fold antidifference h_k of a_k , i. e. $h_k - h_{k-m} = a_k$, if it is an m -fold hypergeometric term.

Examples:

```
12: extended_gosper(binomial(k/2, n), k);
```

$$\frac{(k+2) \binom{k}{2} + (k+1) \binom{k-1}{2}}{2(n+1)}$$

```
13: extended_gosper(k*factorial(k/7), k, 7);
```

$$(k+7) \frac{\text{factorial}(k)}{7}$$

16.67.6 REDUCE operator SUMRECURSION

The `sumrecursion` operator is an implementation of the (fast) Zeilberger algorithm.

- `sumrecursion(f, k, n)` determines a holonomic recurrence equation for

$$\text{sum}(n) = \sum_{k=-\infty}^{\infty} f(n, k)$$

with respect to n , applying `extended_sumrecursion` if necessary, see § 16.67.7. The resulting expression equals zero.

- `sumrecursion(f, k, n, j)` searches for a holonomic recurrence equation of order j . This operator does not use `extended_sumrecursion` automatically. Note that if j is too large, the recurrence equation may not be unique, and only one particular solution is returned.

A simple example deals with Equation (16.67)⁴⁵

```
14: sumrecursion(binomial(n, k), k, n);

2*sum(n - 1) - sum(n)
```

The whole *hypergeometric database* of the *Vandermonde*, *Gauß*, *Kummer*, *Saalschütz*, *Dixon*, *Clausen* and *Dougall identities* (see [9]), and many more identities (see e. g. [2]), can be obtained using `sumrecursion`. As examples, we consider the difficult cases of *Clausen* and *Dougall*:

```
15: summand:=factorial(a+k-1)*factorial(b+k-1)/(factorial(k)*
    factorial(-1/2+a+b+k))*factorial(a+n-k-1)*factorial(b+n-k-1)/
    (factorial(n-k)*factorial(-1/2+a+b+n-k))$

16: sumrecursion(summand, k, n);

(2*a + 2*b + 2*n - 1)*(2*a + 2*b + n - 1)*sum(n)*n
- 2*(2*a + n - 1)*(a + b + n - 1)*(2*b + n - 1)*sum(n - 1)

17: summand:=pochhammer(d, k)*pochhammer(1+d/2, k)*pochhammer(d+b-a, k)*
    pochhammer(d+c-a, k)*pochhammer(1+a-b-c, k)*pochhammer(n+a, k)*
    pochhammer(-n, k)/(factorial(k)*pochhammer(d/2, k)*
    pochhammer(1+a-b, k)*pochhammer(1+a-c, k)*pochhammer(b+c+d-a, k)*
    pochhammer(1+d-a-n, k)*pochhammer(1+d+n, k))$

18: sumrecursion(summand, k, n);

(2*a - b - c - d + n)*(b + n - 1)*(c + n - 1)*(d + n)*sum(n - 1) +
```

⁴⁵Note that with REDUCE Version 3.5 we use the global operator `summ` instead of `sum` to denote the sum.

```
(a - b - c - d - n + 1)*(a - b + n)*(a - c + n)*(a - d + n - 1)
*sum(n)
```

corresponding to the statements

$${}_4F_3\left(\begin{matrix} a, b, 1/2 - a - b - n, -n \\ 1/2 + a + b, 1 - a - n, 1 - b - n \end{matrix} \middle| 1\right) = \frac{(2a)_n (a+b)_n (2b)_n}{(2a+2b)_n (a)_n (b)_n}$$

and

$${}_7F_6\left(\begin{matrix} d, 1 + d/2, d + b - a, d + c - a, 1 + a - b - c, n + a, -n \\ d/2, 1 + a - b, 1 + a - c, b + c + d - a, 1 + d - a - n, 1 + d + n \end{matrix} \middle| 1\right) \\ = \frac{(d+1)_n (b)_n (c)_n (1 + 2a - b - c - d)_n}{(a-d)_n (1 + a - b)_n (1 + a - c)_n (b + c + d - a)_n}$$

(compare next section), respectively.

Other applications of the Zeilberger algorithm are connected with the verification of identities. To prove the identity

$$\sum_{k=0}^n \binom{n}{k}^3 = \sum_{k=0}^n \binom{n}{k}^2 \binom{2k}{n},$$

e. g., we may prove that both sums satisfy the same recurrence equation

```
19: sumrecursion(binomial(n,k)^3,k,n);

      2                                2                                2
(7*n  - 7*n + 2)*sum(n-1) + 8*(n-1) *sum(n-2) - sum(n)*n

20: sumrecursion(binomial(n,k)^2*binomial(2*k,n),k,n);

      2                                2                                2
(7*n  - 7*n + 2)*sum(n-1) + 8*(n-1) *sum(n-2) - sum(n)*n
```

and finally check the initial conditions:

```
21: sub(n=0,k=0,binomial(n,k)^3);

1

22: sub(n=0,k=0,binomial(n,k)^2*binomial(2*k,n));

1
```



```
23: sub(n=1,k=0,binomial(n,k)^3)+sub(n=1,k=1,binomial(n,k)^3);
```

```
2
```

```
24: sub(n=1,k=0,binomial(n,k)^2*binomial(2*k,n))+
    sub(n=1,k=1,binomial(n,k)^2*binomial(2*k,n));
```

```
2
```

16.67.7 REDUCE operator EXTENDED_SUMRECURSION

The `extended_sumrecursion` operator is an implementation of an extension of the (fast) Zeilberger algorithm given by Koepf [2].

- `extended_sumrecursion(f,k,n,m,l)` determines a holonomic recurrence equation for $\text{sum}(n) = \sum_{k=-\infty}^{\infty} f(n,k)$ with respect to n if $f(n,k)$ is an (m,l) -fold hypergeometric term with respect to (n,k) , i. e.

$$\frac{F(n,k)}{F(n-m,k)} \quad \text{and} \quad \frac{F(n,k)}{F(n,k-l)}$$

are rational functions with respect to both n and k . The resulting expression equals zero.

- `sumrecursion(f,k,n)` invokes `extended_sumrecursion(f,k,n,m,l)` with suitable values m and l , and covers therefore the extended algorithm completely.

Examples:

```
25: extended_sumrecursion(binomial(n,k)*binomial(k/2,n),k,n,1,2);
```

```
sum(n - 1) + 2*sum(n)
```

which can be obtained automatically by

```
26: sumrecursion(binomial(n,k)*binomial(k/2,n),k,n);
```

```
sum(n - 1) + 2*sum(n)
```

Similarly, we get

```
27: extended_sumrecursion(binomial(n/2,k),k,n,2,1);
```

```

2*sum(n - 2) - sum(n)

28: sumrecursion(binomial(n/2,k),k,n);

2*sum(n - 2) - sum(n)

29: sumrecursion(hyperterm({a,b,a+1/2-b,1+2*a/3,-n},
    {2*a+1-2*b,2*b,2/3*a,1+a+n/2},4,k)/(factorial(n)*2^(-n)/
    factorial(n/2))/hyperterm({a+1,1},{a-b+1,b+1/2},1,n/2),k,n);

sum(n - 2) - sum(n)

```

In the last example, the program chooses $m = 2$, and $l = 1$ to derive the resulting recurrence equation (see [2], Table 3, (1.3)).

16.67.8 REDUCE operator HYPERRECURSION

Sums to which the Zeilberger algorithm applies, in general are special cases of the *generalized hypergeometric function*

$${}_pF_q \left(\begin{matrix} a_1, & a_2, & \dots, & a_p \\ b_1, & b_2, & \dots, & b_q \end{matrix} \middle| x \right) := \sum_{k=0}^{\infty} \frac{(a_1)_k \cdot (a_2)_k \cdots (a_p)_k}{(b_1)_k \cdot (b_2)_k \cdots (b_q)_k k!} x^k$$

with upper parameters $\{a_1, a_2, \dots, a_p\}$, and lower parameters $\{b_1, b_2, \dots, b_q\}$. If a recursion for a generalized hypergeometric function is to be established, you can use the following REDUCE operator:

- `hyperrecursion(upper, lower, x, n)` determines a holonomic recurrence equation with respect to n for ${}_pF_q \left(\begin{matrix} a_1, & a_2, & \dots, & a_p \\ b_1, & b_2, & \dots, & b_q \end{matrix} \middle| x \right)$, where `upper` = $\{a_1, a_2, \dots, a_p\}$ is the list of upper parameters, and `lower` = $\{b_1, b_2, \dots, b_q\}$ is the list of lower parameters depending on n . If Zeilberger's algorithm does not apply, `extended_sumrecursion` of § 16.67.7 is used.
- `hyperrecursion(upper, lower, x, n, j)` ($j \in \mathbb{N}$) searches only for a holonomic recurrence equation of order j . This operator does not use `extended_sumrecursion` automatically.

Therefore

```

30: hyperrecursion({-n,b},{c},1,n);

(b - c - n + 1)*sum(n - 1) + (c + n - 1)*sum(n)

```

establishes the Vandermonde identity

$${}_2F_1\left(\begin{matrix} -n, & b \\ c \end{matrix} \middle| 1\right) = \frac{(c-b)_n}{(c)_n},$$

whereas

```
31: hyperrecursion({d, 1+d/2, d+b-a, d+c-a, 1+a-b-c, n+a, -n},
                  {d/2, 1+a-b, 1+a-c, b+c+d-a, 1+d-a-n, 1+d+n}, 1, n);

(2*a - b - c - d + n)*(b + n - 1)*(c + n - 1)*(d + n)*sum(n - 1) +

(a - b - c - d - n + 1)*(a - b + n)*(a - c + n)*(a - d + n - 1)

*sum(n)
```

proves Dougall's identity, again.

If a hypergeometric expression is given in hypergeometric notation, then the use of `hyperrecursion` is more natural than the use of `sumrecursion`.

Moreover you may use the `REDUCE` operator

- `hyperterm(upper, lower, x, k)` that yields the hypergeometric term

$$\frac{(a_1)_k \cdot (a_2)_k \cdots (a_p)_k}{(b_1)_k \cdot (b_2)_k \cdots (b_q)_k} x^k$$

with upper parameters `upper = {a1, a2, ..., ap}`, and lower parameters `lower = {b1, b2, ..., bq}`

in connection with hypergeometric terms.

The operator `sumrecursion` can also be used to obtain three-term recurrence equations for systems of orthogonal polynomials with the aid of known hypergeometric representations. By ([4], (2.7.11a)), the discrete Krawtchouk polynomials $k_n^{(p)}(x, N)$ have the hypergeometric representation

$$k_n^{(p)}(x, N) = (-1)^n p^n \binom{N}{n} {}_2F_1\left(\begin{matrix} -n, & -x \\ -N \end{matrix} \middle| \frac{1}{p}\right),$$

and therefore we declare

```
32: krawtchoukterm:=
    (-1)^n*p^n*binomial(NN,n)*hyperterm({-n,-x},{-NN},1/p,k)$
```

and get the three three-term recurrence equations

```

33: sumrecursion(krawtchoukterm,k,n);

((2*p - 1)*n - nn*p - 2*p + x + 1)*sum(n - 1)
- (n - nn - 2)*(p - 1)*sum(n - 2)*p - sum(n)*n

34: sumrecursion(krawtchoukterm,k,x);

(2*(x - 1)*p + n - nn*p - x + 1)*sum(x - 1)
- ((x - 1) - nn)*sum(x)*p - (p - 1)*(x - 1)*sum(x - 2)

35: sumrecursion(krawtchoukterm,k,NN);

((p - 2)*nn + n + x + 1)*sum(nn - 1) + (n - nn)*(p - 1)*sum(nn)
+ (nn - x - 1)*sum(nn - 2)

```

with respect to the parameters n , x , and N respectively.

16.67.9 REDUCE operator HYPERSUM

With the operator `hypersum`, hypergeometric sums are directly evaluated in closed form whenever the extended Zeilberger algorithm leads to a recurrence equation containing only two terms:

- `hypersum(upper, lower, x, n)` determines a closed form representation for ${}_pF_q \left(\begin{matrix} a_1, & a_2, & \dots, & a_p \\ b_1, & b_2, & \dots, & b_q \end{matrix} \middle| x \right)$, where `upper` = $\{a_1, a_2, \dots, a_p\}$ is the list of upper parameters, and `lower` = $\{b_1, b_2, \dots, b_q\}$ is the list of lower parameters depending on n . The result is given as a hypergeometric term with respect to n .

If the result is a list of length m , we call it m -fold *symmetric*, which is to be interpreted as follows: Its j^{th} part is the solution valid for all n of the form $n = mk + j - 1$ ($k \in \mathbb{N}_0$). In particular, if the resulting list contains two terms, then the first part is the solution for even n , and the second part is the solution for odd n .

Examples [2]:

```

36: hypersum({a, 1+a/2, c, d, -n}, {a/2, 1+a-c, 1+a-d, 1+a+n}, 1, n);

pochhammer(a - c - d + 1, n)*pochhammer(a + 1, n)
-----
pochhammer(a - c + 1, n)*pochhammer(a - d + 1, n)

```

```
37: hypersum({a, 1+a/2, d, -n}, {a/2, 1+a-d, 1+a+n}, -1, n);
```

$$\frac{\text{pochhammer}(a + 1, n)}{\text{pochhammer}(a - d + 1, n)}$$

Note that the operator `togamma` converts expressions given in factorial- Γ -binomial-Pochhammer notation into a pure Γ function representation:

```
38: togamma(ws);
```

$$\frac{\text{gamma}(a - d + 1) * \text{gamma}(a + n + 1)}{\text{gamma}(a - d + n + 1) * \text{gamma}(a + 1)}$$

Here are some m -fold symmetric results:

```
39: hypersum({-n, -n, -n}, {1, 1}, 1, n);
```

$$\left\{ \frac{(-27)^{n/2} \text{pochhammer}\left(\frac{2}{3}, \frac{n}{2}\right) \text{pochhammer}\left(\frac{1}{3}, \frac{n}{2}\right)}{\text{factorial}\left(\frac{n}{2}\right)}, 0 \right\}$$

```
40: hypersum({-n, n+3*a, a}, {3*a/2, (3*a+1)/2}, 3/4, n);
```

$$\left\{ \frac{\text{pochhammer}\left(\frac{2}{3}, \frac{n}{3}\right) \text{pochhammer}\left(\frac{1}{3}, \frac{n}{3}\right)}{\text{pochhammer}\left(\frac{3*a+2}{3}, \frac{n}{3}\right) \text{pochhammer}\left(\frac{3*a+1}{3}, \frac{n}{3}\right)}, 0, 0 \right\}$$

These results correspond to the formulas (compare [2])

$${}_3F_2\left(\begin{matrix} -n, -n, -n \\ 1, 1 \end{matrix} \middle| 1\right) = \begin{cases} 0 & \text{if } n \text{ odd} \\ \frac{(1/3)_{n/2} (2/3)_{n/2}}{(n/2)!^2} (-27)^{n/2} & \text{otherwise} \end{cases}$$

and

$${}_3F_2\left(\begin{matrix} -n, n+3a, a \\ 3a/2, (3a+1)/2 \end{matrix} \middle| \frac{3}{4}\right) = \begin{cases} 0 & \text{if } n \not\equiv 0 \pmod{3} \\ \frac{(1/3)_{n/3} (2/3)_{n/3}}{(a+1/3)_{n/3} (a+2/3)_{n/3}} & \text{otherwise} \end{cases}.$$

16.67.10 REDUCE operator SUMTOHYPER

With the operator `sumtohyper`, sums given in factorial- Γ -binomial-Pochhammer notation are converted into hypergeometric notation.

- `sumtohyper(f, k)` determines the hypergeometric representation of $\sum_{k=-\infty}^{\infty} f_k$, i. e. its output is `c*hypergeometric(upper, lower, x)`, corresponding to the representation

$$\sum_{k=-\infty}^{\infty} f_k = c \cdot {}_pF_q\left(\begin{matrix} a_1, & a_2, & \dots, & a_p \\ b_1, & b_2, & \dots, & b_q \end{matrix} \middle| x\right),$$

where `upper` = $\{a_1, a_2, \dots, a_p\}$ and `lower` = $\{b_1, b_2, \dots, b_q\}$ are the lists of upper and lower parameters.

Examples:

```
41: sumtohyper(binomial(n, k)^3, k);
```

```
hypergeometric({ - n, - n, - n}, {1, 1}, -1)
```

```
42: sumtohyper(binomial(n, k)/2^n-sub(n=n-1, binomial(n, k)/2^n), k);
```

$$- \text{hypergeometric}\left(\left\{\frac{-n+2}{2}, -n, 1\right\}, \left\{1, \frac{-n}{2}\right\}, -1\right)$$

$$\frac{n}{2}$$

16.67.11 Simplification Operators

For the decision that an expression a_k is a hypergeometric term, it is necessary to find out whether or not a_k/a_{k-1} is a rational function with respect to k . For the purpose to decide whether or not an expression involving powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols is a hypergeometric term, the following simplification operators can be used:

- `simplify_gamma(f)` simplifies an expression f involving only rational, powers and Γ function terms according to a recursive application of the simplification rule $\Gamma(a+1) = a\Gamma(a)$ to the expression tree. Since all Γ arguments with integer difference are transformed, this gives a decision procedure for rationality for integer-linear Γ term product ratios.
- `simplify_combinatorial(f)` simplifies an expression f involving powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols by converting factorials, binomial coefficients, and Pochhammer symbols into Γ function terms, and applying `simplify_gamma` to its result. If the output is not rational, it is given in terms of Γ functions. If you prefer factorials you may use
- `gammatofactorial(rule)` converting Γ function terms into factorials using $\Gamma(x) \rightarrow (x-1)!$.
- `simplify_gamma2(f)` uses the duplication formula of the Γ function to simplify f .
- `simplify_gamman(f,n)` uses the multiplication formula of the Γ function to simplify f .

The use of `simplify_combinatorial(f)` is a safe way to decide the rationality for any ratio of products of powers, factorials, Γ function terms, binomial coefficients, and Pochhammer symbols.

Example:

```
43: simplify_combinatorial(sub(k=k+1,krawtchoukterm)/krawtchoukterm);
```

$$\frac{(k-n)*(k-x)}{(k-nn)*(k+1)*p}$$

From this calculation, we see again that the upper parameters of the hypergeometric representation of the Krawtchouk polynomials are given by $\{-n, -x\}$, its lower parameter is $\{-N\}$, and the argument of the hypergeometric function is $1/p$.

Other examples are

```
44: simplify_combinatorial(binomial(n,k)/binomial(2*n,k-1));
```

$$\frac{\text{gamma}(-(k-2*n-2))*\text{gamma}(n+1)}{\text{gamma}(-(k-n-1))*\text{gamma}(2*n+1)*k}$$

```
45: ws where gammatofactorial;
```

```

factorial(-k + 2*n + 1)*factorial(n)
-----
factorial(-k + n)*factorial(2*n)*k
46: simplify_gamma2(gamma(2*n)/gamma(n));

      2*n      2*n + 1
      *gamma(-----)
              2
-----
      2*sqrt(pi)

47: simplify_gamman(gamma(3*n)/gamma(n), 3);

      3*n      3*n + 2      3*n + 1
      *gamma(-----)*gamma(-----)
              3              3
-----
      2*sqrt(3)*pi

```

16.67.12 Tracing

If you set

```
48: on zb_trace;
```

tracing is enabled, and you get intermediate results, see [2].

Example for the Gosper algorithm:

```
49: gosper(pochhammer(k-n,n), k);
```

$$a(k)/a(k-1) := \frac{k-1}{k-n-1}$$

Gosper algorithm applicable

```
p:= 1
```

```
q:= k - 1
```

```
r:= k - n - 1
```

```
degreebound := 0
```


$$f := \frac{1}{n+1}$$

Gosper algorithm successful

$$\frac{\text{pochhammer}(k-n, n) * k}{n+1}$$

Example for the Zeilberger algorithm:

```
50: sumrecursion(binomial(n,k)^2,k,n);
```

$$F(n,k)/F(n-1,k) := \frac{n^2}{(k-n)^2}$$

$$F(n,k)/F(n,k-1) := \frac{(k-n-1)^2}{k^2}$$

Zeilberger algorithm applicable

applying Zeilberger algorithm for order:= 1

$$p := \text{zb_sigma}(1) * k^2 - 2 * \text{zb_sigma}(1) * k * n + \text{zb_sigma}(1) * n^2 + n^2$$

$$q := k^2 - 2 * k * n - 2 * k + n^2 + 2 * n + 1$$

$$r := k^2$$

degreebound := 1

$$f := \frac{2 * k - 3 * n + 2}{n}$$

$$p := \frac{-4 * k^2 * n + 2 * k^2 + 8 * k * n^2 - 4 * k * n^3 - 3 * n^3 + 2 * n^2}{n}$$

```

Zeilberger algorithm successful

4*sum(n - 1)*n - 2*sum(n - 1) - sum(n)*n

51: off zb_trace;

```

16.67.13 Global Variables and Switches

The following global variables and switches can be used in connection with the ZEILBERG package:

- `zb_trace`, switch; default setting `off`. Turns tracing on and off.
- `zb_direction`, variable; settings: `down`, `up`; default setting `down`.

In the case of the Gosper algorithm, either a downward or a forward antidifference is calculated, i. e., `gosper` finds g_k with either

$$a_k = g_k - g_{k-1} \quad \text{or} \quad a_k = g_{k+1} - g_k,$$

respectively.

In the case of the Zeilberger algorithm, either a downward or an upward recurrence equation is returned. Example:

```

52: zb_direction:=up$

53: sumrecursion(binomial(n,k)^2,k,n);

sum(n + 1)*n + sum(n + 1) - 4*sum(n)*n - 2*sum(n)

54: zb_direction:=down$

```

- `zb_order`, variable; settings: any nonnegative integer; default setting 5. Gives the maximal order for the recurrence equation that `sumrecursion` searches for.
- `zb_factor`, switch; default setting `on`. If `off`, the factorization of the output usually producing nicer results is suppressed.
- `zb_proof`, switch; default setting `off`. If `on`, then several intermediate results are stored in global variables:
- `gosper_representation`, variable; default setting `nil`.

If a `gosper` command is issued, and if the Gosper algorithm is applicable, then the variable `gosper_representation` is set to the list of polynomials (with respect to k) $\{p, q, r, f\}$ corresponding to the representation

$$\frac{a_k}{a_{k-1}} = \frac{p_k}{p_{k-1}} \frac{q_k}{r_k}, \quad g_k = \frac{q_{k+1}}{p_k} f_k a_k,$$

see [1]. Examples:

```
55: on zb_proof;

56: gosper(k*factorial(k),k);

(k + 1)*factorial(k)

57: gosper_representation;

{k,k,1,1}

58: gosper(
    1/(k+1)*binomial(2*k,k)/(n-k+1)*binomial(2*n-2*k,n-k),k);

((2*k - n + 1)*(2*k + 1)*binomial(-2*(k - n), -(k - n))
 *binomial(2*k,k))/((k + 1)*(n + 2)*(n + 1))

59: gosper_representation;

{1,

(2*k - 1)*(k - n - 2),

(2*k - 2*n - 1)*(k + 1),

- (2*k - n + 1)
-----}
(n + 2)*(n + 1)
```

- `zeilberger_representation`, variable; default setting `nil`.

If a `sumrecursion` command is issued, and if the Zeilberger algorithm is successful, then the variable `zeilberger_representation` is set to the final Gosper representation used, see [3].

16.67.14 Messages

The following messages may occur:

- ***** Gosper algorithm: no closed form solution exists

Example input:

```
gosper(factorial(k), k).
```

- ***** Gosper algorithm not applicable

Example input:

```
gosper(factorial(k/2), k).
```

The term ratio a_k/a_{k-1} is not rational.

- ***** illegal number of arguments

Example input:

```
gosper(k).
```

- ***** Zeilberger algorithm fails. Enlarge zb_order

Example input:

```
sumrecursion(binomial(n, k)*binomial(6*k, n), k, n)
```

For this example a setting `zb_order:=6` is needed.

- ***** Zeilberger algorithm not applicable

Example input:

```
sumrecursion(binomial(n/2, k), k, n)
```

One of the term ratios $f(n, k)/f(n-1, k)$ or $f(n, k)/f(n, k-1)$ is not rational.

- ***** SOLVE given inconsistent equations

You can ignore this message that occurs with Version 3.5.

Bibliography

- [1] Gosper Jr., R. W.: Decision procedure for indefinite hypergeometric summation. *Proc. Natl. Acad. Sci. USA* **75**, 1978, 40–42.
- [2] Koepf, W.: Algorithms for the indefinite and definite summation. Konrad-Zuse-Zentrum Berlin (ZIB), Preprint SC 94-33, 1994.
- [3] Koornwinder, T. H.: On Zeilberger's algorithm and its q -analogue: a rigorous description. *J. of Comput. and Appl. Math.* **48**, 1993, 91–111.
- [4] Nikiforov, A. F., Suslov, S. K., and Uvarov, V. B.: *Classical orthogonal polynomials of a discrete variable*. Springer-Verlag, Berlin–Heidelberg–New York, 1991.

- [5] Paule, P. and Schorn, M.: A MATHEMATICA version of Zeilberger's algorithm for proving binomial coefficient identities. J. Symbolic Computation, 1994, to appear.
- [6] Problem 94–2, SIAM Review **36**, March 1994.
- [7] Strehl, V.: Binomial sums and identities. Maple Technical Newsletter **10**, 1993, 37–49.
- [8] Wilf, H. S.: *Generatingfunctionology*. Academic Press, Boston, 1990.
- [9] Wilf, H. S.: Identities and their computer proofs. “SPICE” Lecture Notes, August 31–September 2, 1993. Anonymous ftp file `pub/wilf/lecnotes.ps` on the server `ftp.cis.upenn.edu`.
- [10] Zeilberger, D.: A fast algorithm for proving terminating hypergeometric identities. Discrete Math. **80**, 1990, 207–211.
- [11] Zeilberger, D.: The method of creative telescoping. J. Symbolic Computation **11**, 1991, 195–204.

16.68 ZTRANS: Z -transform package

This package is an implementation of the Z -transform of a sequence. This is the discrete analogue of the Laplace Transform.

Authors: Wolfram Koepf and Lisa Temme.

16.68.1 Z -Transform

The Z -Transform of a sequence $\{f_n\}$ is the discrete analogue of the Laplace Transform, and

$$\mathcal{Z}\{f_n\} = F(z) = \sum_{n=0}^{\infty} f_n z^{-n}.$$

This series converges in the region outside the circle $|z| = |z_0| = \limsup_{n \rightarrow \infty} \sqrt[n]{|f_n|}$.

SYNTAX: `ztrans(f_n , n , z)` where f_n is an expression, and n, z are identifiers.

16.68.2 Inverse Z -Transform

The calculation of the Laurent coefficients of a regular function results in the following inverse formula for the Z -Transform:

If $F(z)$ is a regular function in the region $|z| > \rho$ then \exists a sequence $\{f_n\}$ with $\mathcal{Z}\{f_n\} = F(z)$ given by

$$f_n = \frac{1}{2\pi i} \oint F(z) z^{n-1} dz$$

SYNTAX: `invztrans($F(z)$, z , n)` where $F(z)$ is an expression, and z, n are identifiers.

16.68.3 Input for the Z -Transform

This package can compute the Z -Transforms of the following list of f_n , and certain combinations thereof.

1	$e^{\alpha n}$	$\frac{1}{(n+k)}$
$\frac{1}{n!}$	$\frac{1}{(2n)!}$	$\frac{1}{(2n+1)!}$

$$\begin{array}{lll}
\frac{\sin(\beta n)}{n!} & \sin(\alpha n + \phi) & e^{\alpha n} \sin(\beta n) \\
\frac{\cos(\beta n)}{n!} & \cos(\alpha n + \phi) & e^{\alpha n} \cos(\beta n) \\
\frac{\sin(\beta(n+1))}{n+1} & \sinh(\alpha n + \phi) & \frac{\cos(\beta(n+1))}{n+1} \\
\cosh(\alpha n + \phi) & \binom{n+k}{m} &
\end{array}$$

Other Combinations

Linearity $\mathcal{Z}\{af_n + bg_n\} = a\mathcal{Z}\{f_n\} + b\mathcal{Z}\{g_n\}$

Multiplication by n $\mathcal{Z}\{n^k \cdot f_n\} = -z \frac{d}{dz} (\mathcal{Z}\{n^{k-1} \cdot f_n, n, z\})$

Multiplication by λ^n $\mathcal{Z}\{\lambda^n \cdot f_n\} = F\left(\frac{z}{\lambda}\right)$

Shift Equation $\mathcal{Z}\{f_{n+k}\} = z^k \left(F(z) - \sum_{j=0}^{k-1} f_j z^{-j} \right)$

Symbolic Sums $\mathcal{Z}\left\{\sum_{k=0}^n f_k\right\} = \frac{z}{z-1} \cdot \mathcal{Z}\{f_n\}$

$\mathcal{Z}\left\{\sum_{k=p}^{n+q} f_k\right\}$ combination of the above

where $k, \lambda \in \mathbf{N} - \{0\}$; and a, b are variables or fractions; and $p, q \in \mathbf{Z}$ or are functions of n ; and α, β & ϕ are angles in radians.

16.68.4 Input for the Inverse Z -Transform

This package can compute the Inverse Z -Transforms of any rational function, whose denominator can be factored over \mathbf{Q} , in addition to the following list of $F(z)$.

$$\begin{array}{ll}
\sin\left(\frac{\sin(\beta)}{z}\right) e^{\left(\frac{\cos(\beta)}{z}\right)} & \cos\left(\frac{\sin(\beta)}{z}\right) e^{\left(\frac{\cos(\beta)}{z}\right)} \\
\sqrt{\frac{z}{A}} \sin\left(\sqrt{\frac{z}{A}}\right) & \cos\left(\sqrt{\frac{z}{A}}\right) \\
\sqrt{\frac{z}{A}} \sinh\left(\sqrt{\frac{z}{A}}\right) & \cosh\left(\sqrt{\frac{z}{A}}\right)
\end{array}$$

$$z \log \left(\frac{z}{\sqrt{z^2 - Az + B}} \right) \qquad z \log \left(\frac{\sqrt{z^2 + Az + B}}{z} \right)$$

$$\arctan \left(\frac{\sin(\beta)}{z + \cos(\beta)} \right)$$

where $k, \lambda \in \mathbf{N} - \{0\}$ and A, B are fractions or variables ($B > 0$) and α, β , & ϕ are angles in radians.

16.68.5 Application of the Z -Transform

Solution of difference equations

In the same way that a Laplace Transform can be used to solve differential equations, so Z -Transforms can be used to solve difference equations.

Given a linear difference equation of k -th order

$$f_{n+k} + a_1 f_{n+k-1} + \dots + a_k f_n = g_n \quad (16.68)$$

with initial conditions $f_0 = h_0, f_1 = h_1, \dots, f_{k-1} = h_{k-1}$ (where h_j are given), it is possible to solve it in the following way. If the coefficients a_1, \dots, a_k are constants, then the Z -Transform of (16.68) can be calculated using the shift equation, and results in a solvable linear equation for $\mathcal{Z}\{f_n\}$. Application of the Inverse Z -Transform then results in the solution of (16.68).

If the coefficients a_1, \dots, a_k are polynomials in n then the Z -Transform of (16.68) constitutes a differential equation for $\mathcal{Z}\{f_n\}$. If this differential equation can be solved then the Inverse Z -Transform once again yields the solution of (16.68). Some examples of these methods of solution can be found in §16.68.6.

16.68.6 EXAMPLES

Here are some examples for the Z -Transform

```
1: ztrans((-1)^n*n^2,n,z);
```

$$\frac{z^3(-z+1)}{z^3+3z^2+3z+1}$$

2: ztrans(cos(n*omega*t),n,z);

$$\frac{z(\cos(\omega t) - z)}{2\cos(\omega t)z^2 - z^2 - 1}$$

3: ztrans(cos(b*(n+2))/(n+2),n,z);

$$z(-\cos(b) + \log(\frac{z}{\sqrt{-2\cos(b)z + z^2 + 1}}))z$$

4: ztrans(n*cos(b*n)/factorial(n),n,z);

$$\frac{e^{\cos(b)/z} \left(\cos\left(\frac{\sin(b)}{z}\right) \cos(b) - \sin\left(\frac{\sin(b)}{z}\right) \sin(b) \right)}{z}$$

5: ztrans(sum(1/factorial(k),k,0,n),n,z);

$$\frac{e^{1/z} - 1}{z}$$

6: operator f\$

7: ztrans((1+n)^2*f(n),n,z);

$$z^2 \frac{d}{dz} (z \frac{d}{dz} (z \frac{d}{dz} f(n))) - \frac{d}{dz} (z \frac{d}{dz} (z \frac{d}{dz} f(n))) + z \frac{d}{dz} f(n)$$

Here are some examples for the Inverse Z-Transform

8: invztrans((z^2-2*z)/(z^2-4*z+1),z,n);

$$\frac{(\sqrt{3} - 2)^n}{2} + \frac{(\sqrt{3} + 2)^n}{2}$$

$$\frac{z}{(z-a)(z-b)}$$

9: invztrans(z/((z-a)*(z-b)),z,n);

$$\frac{a^n - b^n}{a - b}$$

10: invztrans(z/((z-a)*(z-b)*(z-c)),z,n);

$$\frac{a^n * b - a^n * c - b^n * a + b^n * c + c^n * a - c^n * b}{a^2 * b - a^2 * c - a * b^2 + a * c^2 + b^2 * c - b * c^2}$$

11: invztrans(z*log(z/(z-a)),z,n);

$$\frac{a^n * a}{n + 1}$$

12: invztrans(e^(1/(a*z)),z,n);

$$\frac{1}{a * \text{factorial}(n)}$$

13: invztrans(z*(z-cosh(a))/(z^2-2*z*cosh(a)+1),z,n);

$$\cosh(a * n)$$

Examples: Solutions of Difference Equations

- I (See [1], p. 651, Example 1).
Consider the homogeneous linear difference equation

$$f_{n+5} - 2f_{n+3} + 2f_{n+2} - 3f_{n+1} + 2f_n = 0$$

with initial conditions $f_0 = 0, f_1 = 0, f_2 = 9, f_3 = -2, f_4 = 23$. The Z-Transform of the left hand side can be written as $F(z) = P(z)/Q(z)$ where $P(z) = 9z^3 - 2z^2 + 5z$ and $Q(z) = z^5 - 2z^3 + 2z^2 - 3z + 2 = (z-1)^2(z+2)(z^2+1)$, which can be inverted to give

$$f_n = 2n + (-2)^n - \cos \frac{\pi}{2} n.$$

The following REDUCE session shows how the present package can be used to solve the above problem.

```

14: operator f$ f(0):=0$ f(1):=0$ f(2):=9$ f(3):=-2$ f(4):=23$

20: equation:=ztrans(f(n+5)-2*f(n+3)+2*f(n+2)-3*f(n+1)+2*f(n),n,z);

equation := ztrans(f(n),n,z)*z5 - 2*ztrans(f(n),n,z)*z3
+ 2*ztrans(f(n),n,z)*z2 - 3*ztrans(f(n),n,z)*z
+ 2*ztrans(f(n),n,z) - 9*z3 + 2*z2 - 5*z

21: ztransresult:=solve(equation,ztrans(f(n),n,z));

ztransresult := {ztrans(f(n),n,z)= $\frac{z*(9*z^2 - 2*z + 5)}{z^5 - 2*z^3 + 2*z^2 - 3*z + 2}$ }

22: result:=invztrans(part(first(ztransresult),2),z,n);

result :=  $\frac{2*(-2)^n - i*(-1)^n - i + 4*n}{2}$ 

```

- II (See [1], p. 651, Example 2).
Consider the inhomogeneous difference equation:

$$f_{n+2} - 4f_{n+1} + 3f_n = 1$$

with initial conditions $f_0 = 0, f_1 = 1$. Giving

$$\begin{aligned} F(z) &= \mathcal{Z}\{1\} \left(\frac{1}{z^2-4z+3} + \frac{z}{z^2-4z+3} \right) \\ &= \frac{z}{z-1} \left(\frac{1}{z^2-4z+3} + \frac{z}{z^2-4z+3} \right). \end{aligned}$$

The Inverse Z -Transform results in the solution

$$f_n = \frac{1}{2} \left(\frac{3^{n+1}-1}{2} - (n+1) \right).$$

The following REDUCE session shows how the present package can be used to solve the above problem.

```
23: clear(f)$ operator f$ f(0):=0$ f(1):=1$

27: equation:=ztrans(f(n+2)-4*f(n+1)+3*f(n)-1,n,z);

equation := (ztrans(f(n),n,z)*z3 - 5*ztrans(f(n),n,z)*z2
+ 7*ztrans(f(n),n,z)*z - 3*ztrans(f(n),n,z) - z2)/(z2 - 1)

28: ztransresult:=solve(equation,ztrans(f(n),n,z));

result := {ztrans(f(n),n,z)=2z2-----}
3z3 - 5*z2 + 7*z - 3

29: result:=invztrans(part(first(ztransresult),2),z,n);
```

$$\text{result} := \frac{3 \cdot 3^n - 2 \cdot n - 3}{4}$$

III Consider the following difference equation, which has a differential equation for $\mathcal{Z}\{f_n\}$.

$$(n+1) \cdot f_{n+1} - f_n = 0$$

with initial conditions $f_0 = 1, f_1 = 1$. It can be solved in REDUCE using the present package in the following way.

```

30: clear(f)$ operator f$ f(0):=1$ f(1):=1$

34: equation:=ztrans((n+1)*f(n+1)-f(n),n,z);

equation := - (df(ztrans(f(n),n,z),z)*z2 + ztrans(f(n),n,z))

35: operator tmp;

36: equation:=sub(ztrans(f(n),n,z)=tmp(z),equation);

equation := - (df(tmp(z),z)*z2 + tmp(z))

37: load(odesolve);

38: ztransresult:=odesolve(equation,tmp(z),z);

ztransresult := {tmp(z)=e1/z*arbconst(1)}

39: preresult:=invztrans(part(first(ztransresult),2),z,n);

preresult := -----
               arbconst(1)
               factorial(n)

```

```
40: solve({sub(n=0,preresult)=f(0),sub(n=1,preresult)=f(1)},
arbconst(1));

{arbconst(1)=1}

41: result:=preresult where ws;

      1
result := -----
      factorial(n)
```

Bibliography

- [1] Bronstein, I.N. and Semedjajew, K.A., *Taschenbuch der Mathematik*, Verlag Harri Deutsch, Thun und Frankfurt(Main), 1981.
ISBN 3 87144 492 8.

Chapter 17

Symbolic Mode

At the system level, REDUCE is based on a version of the programming language Lisp known as *Standard Lisp* which is described in J. Marti, Hearn, A. C., Griss, M. L. and Griss, C., “Standard LISP Report” SIGPLAN Notices, ACM, New York, 14, No 10 (1979) 48-68. We shall assume in this section that the reader is familiar with the material in that paper. This also assumes implicitly that the reader has a reasonable knowledge about Lisp in general, say at the level of the LISP 1.5 Programmer’s Manual (McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I., “LISP 1.5 Programmer’s Manual”, M.I.T. Press, 1965) or any of the books mentioned at the end of this section. Persons unfamiliar with this material will have some difficulty understanding this section.

Although REDUCE is designed primarily for algebraic calculations, its source language is general enough to allow for a full range of Lisp-like symbolic calculations. To achieve this generality, however, it is necessary to provide the user with two modes of evaluation, namely an algebraic mode and a symbolic mode. To enter symbolic mode, the user types `symbolic;` (or `lisp;`) and to return to algebraic mode one types `algebraic;`. Evaluations proceed differently in each mode so the user is advised to check what mode he is in if a puzzling error arises. He can find his mode by typing

```
eval_mode;
```

The current mode will then be printed as `ALGEBRAIC` or `SYMBOLIC`.

Expression evaluation may proceed in either mode at any level of a calculation, provided the results are passed from mode to mode in a compatible manner. One simply prefixes the relevant expression by the appropriate mode. If the mode name prefixes an expression at the top level, it will then be handled as if the global system mode had been changed for the scope of that particular calculation.

For example, if the current mode is `ALGEBRAIC`, then the commands

```
symbolic car '(a);
x+y;
```

will cause the first expression to be evaluated and printed in symbolic mode and the second in algebraic mode. Only the second evaluation will thus affect the expression workspace. On the other hand, the statement

```
x + symbolic car '(12);
```

will result in the algebraic value $X+12$.

The use of `SYMBOLIC` (and equivalently `ALGEBRAIC`) in this manner is the same as any operator. That means that parentheses could be omitted in the above examples since the meaning is obvious. In other cases, parentheses must be used, as in

```
symbolic(x := 'a);
```

Omitting the parentheses, as in

```
symbolic x := a;
```

would be wrong, since it would parse as

```
symbolic(x) := a;
```

For convenience, it is assumed that any operator whose *first* argument is quoted is being evaluated in symbolic mode, regardless of the mode in effect at that time. Thus, the first example above could be equally well written:

```
car '(a);
```

Except where explicit limitations have been made, most `REDUCE` algebraic constructions carry over into symbolic mode. However, there are some differences. First, expression evaluation now becomes Lisp evaluation. Secondly, assignment statements are handled differently, as we shall discuss shortly. Thirdly, local variables and array elements are initialized to `NIL` rather than `0`. (In fact, any variables not explicitly declared `INTEGER` are also initialized to `NIL` in algebraic mode, but the algebraic evaluator recognizes `NIL` as `0`.) Finally, function definitions follow the conventions of Standard Lisp.

To begin with, we mention a few extensions to our basic syntax which are designed primarily if not exclusively for symbolic mode.

17.1 Symbolic Infix Operators

There are three binary infix operators in REDUCE intended for use in symbolic mode, namely `.` (CONS), `EQ` and `MEMQ`. The precedence of these operators was given in another section.

17.2 Symbolic Expressions

These consist of scalar variables and operators and follow the normal rules of the Lisp meta language.

Examples:

```
x
car u . reverse v
simp (u+v^2)
```

17.3 Quoted Expressions

Because symbolic evaluation requires that each variable or expression has a value, it is necessary to add to REDUCE the concept of a quoted expression by analogy with the Lisp `QUOTE` function. This is provided by the single quote mark `'`. For example,

<code>' a</code>	represents the Lisp S-expression	<code>(quote a)</code>
<code>' (a b c)</code>	represents the Lisp S-expression	<code>(quote (a b c))</code>

Note, however, that strings are constants and therefore evaluate to themselves in symbolic mode. Thus, to print the string `"A String"`, one would write

```
prin2 "A String";
```

Within a quoted expression, identifier syntax rules are those of REDUCE. Thus `(A ! . B)` is the list consisting of the three elements `A`, `!`, and `B`, whereas `(A . B)` is the dotted pair of `A` and `B`.

17.4 Lambda Expressions

LAMBDA expressions provide the means for constructing Lisp LAMBDA expressions in symbolic mode. They may not be used in algebraic mode.

Syntax:

$$\langle \text{LAMBDA expression} \rangle \longrightarrow \\ \text{LAMBDA } \langle \text{varlist} \rangle \langle \text{terminator} \rangle \langle \text{statement} \rangle$$

where

$$\langle \text{varlist} \rangle ((\langle \text{variable} \rangle, \dots, \langle \text{variable} \rangle))$$

e.g.,

```
lambda (x,y); car x . cdr y;
```

is equivalent to the Lisp LAMBDA expression

```
(lambda (x y) (cons (car x) (cdr y)))
```

The parentheses may be omitted in specifying the variable list if desired.

LAMBDA expressions may be used in symbolic mode in place of prefix operators, or as an argument of the reserved word FUNCTION.

In those cases where a LAMBDA expression is used to introduce local variables to avoid recomputation, a WHERE statement can also be used. For example, the expression

```
(lambda (x,y); list(car x,cdr x,car y,cdr y))
(reverse u,reverse v)
```

can also be written

```
{car x,cdr x,car y,cdr y} where x=reverse u,y=reverse v
```

Where possible, WHERE syntax is preferred to LAMBDA syntax, since it is more natural.

17.5 Symbolic Assignment Statements

In symbolic mode, if the left side of an assignment statement is a variable, a SETQ of the right-hand side to that variable occurs. If the left-hand side is an expression, it must be of the form of an array element, otherwise an error will result. For example, $x := y$ translates into (SETQ X Y) whereas $a(3) := 3$ will be valid if A has been previously declared a single dimensioned array of at least four elements.

17.6 FOR EACH Statement

The `FOR EACH` form of the `FOR` statement, designed for iteration down a list, is more general in symbolic mode. Its syntax is:

```
FOR EACH ID:identifier {IN|ON} LST:list
      {DO|COLLECT|JOIN|PRODUCT|SUM} EXPRN:S-expr
```

As in algebraic mode, if the keyword `IN` is used, iteration is on each element of the list. With `ON`, iteration is on the whole list remaining at each point in the iteration. As a result, we have the following equivalence between each form of `FOR EACH` and the various mapping functions in Lisp:

	DO	COLLECT	JOIN
IN	MAPC	MAPCAR	MAPCAN
ON	MAP	MAPLIST	MAPCON

Example: To list each element of the list `(a b c)`:

```
for each x in '(a b c) collect list x;
```

17.7 Symbolic Procedures

All the functions described in the Standard Lisp Report are available to users in symbolic mode. Additional functions may also be defined as symbolic procedures. For example, to define the Lisp function `ASSOC`, the following could be used:

```
symbolic procedure assoc(u,v);
  if null v then nil
  else if u = caar v then car v
  else assoc(u, cdr v);
```

If the default mode were symbolic, then `SYMBOLIC` could be omitted in the above definition. `MACROS` may be defined by prefixing the keyword `PROCEDURE` by the word `MACRO`. (In fact, ordinary functions may be defined with the keyword `EXPR` prefixing `PROCEDURE` as was used in the Standard Lisp Report.) For example, we could define a `MACRO` `CONSCONS` by

```
symbolic macro procedure conscons l;
  expand(cdr l, 'cons);
```

Another form of macro, the `SMACRO` is also available. These are described in the Standard Lisp Report. The Report also defines a function type `FEXPR`. However, its use is discouraged since it is hard to implement efficiently, and most uses can be replaced by macros. At the present time, there are no `FEXPR`s in the core `REDUCE` system.

17.8 Standard Lisp Equivalent of Reduce Input

A user can obtain the Standard Lisp equivalent of his `REDUCE` input by turning on the switch `DEFN` (for definition). The system then prints the Lisp translation of his input but does not evaluate it. Normal operation is resumed when `DEFN` is turned off.

17.9 Communicating with Algebraic Mode

One of the principal motivations for a user of the algebraic facilities of `REDUCE` to learn about symbolic mode is that it gives one access to a wider range of techniques than is possible in algebraic mode alone. For example, if a user wishes to use parts of the system defined in the basic system source code, or refine their algebraic code definitions to make them more efficient, then it is necessary to understand the source language in fairly complete detail. Moreover, it is also necessary to know a little more about the way `REDUCE` operates internally. Basically, `REDUCE` considers expressions in two forms: prefix form, which follow the normal Lisp rules of function composition, and so-called canonical form, which uses a completely different syntax.

Once these details are understood, the most critical problem faced by a user is how to make expressions and procedures communicate between symbolic and algebraic mode. The purpose of this section is to teach a user the basic principles for this.

If one wants to evaluate an expression in algebraic mode, and then use that expression in symbolic mode calculations, or vice versa, the easiest way to do this is to assign a variable to that expression whose value is easily obtainable in both modes. To facilitate this, a declaration `SHARE` is available. `SHARE` takes a list of identifiers as argument, and marks these variables as having recognizable values in both modes. The declaration may be used in either mode.

E.g.,

```
share x,y;
```

says that `X` and `Y` will receive values to be used in both modes.

If a `SHARE` declaration is made for a variable with a previously assigned algebraic value, that value is also made available in symbolic mode.

17.9.1 Passing Algebraic Mode Values to Symbolic Mode

If one wishes to work with parts of an algebraic mode expression in symbolic mode, one simply makes an assignment of a shared variable to the relevant expression in algebraic mode. For example, if one wishes to work with $(a+b)^2$, one would say, in algebraic mode:

```
x := (a+b) ^2;
```

assuming that `x` was declared shared as above. If we now change to symbolic mode and say

```
x;
```

its value will be printed as a prefix form with the syntax:

```
(*SQ <standard quotient> T)
```

This particular format reflects the fact that the algebraic mode processor currently likes to transfer prefix forms from command to command, but doesn't like to re-convert standard forms (which represent polynomials) and standard quotients back to a true Lisp prefix form for the expression (which would result in excessive computation). So `*SQ` is used to tell the algebraic processor that it is dealing with a prefix form which is really a standard quotient and the second argument (`T` or `NIL`) tells it whether it needs further processing (essentially, an *already simplified* flag).

So to get the true standard quotient form in symbolic mode, one needs `CADR` of the variable. E.g.,

```
z := cadr x;
```

would store in `z` the standard quotient form for $(a+b)^2$.

Once you have this expression, you can now manipulate it as you wish. To facilitate this, a standard set of selectors and constructors are available for getting at parts of the form. Those presently defined are as follows:

REDUCE Selectors

DENR	denominator of standard quotient
LC	leading coefficient of polynomial
LDEG	leading degree of polynomial
LPOW	leading power of polynomial
LT	leading term of polynomial
MVAR	main variable of polynomial
NUMR	numerator (of standard quotient)
PDEG	degree of a power
RED	reductum of polynomial
TC	coefficient of a term
TDEG	degree of a term
TPOW	power of a term

REDUCE Constructors

- . + add a term to a polynomial
- . / divide (two polynomials to get quotient)
- . * multiply power by coefficient to produce term
- . ^ raise a variable to a power

For example, to find the numerator of the standard quotient above, one could say:

```
numr z;
```

or to find the leading term of the numerator:

```
lt numr z;
```

Conversion between various data structures is facilitated by the use of a set of functions defined for this purpose. Those currently implemented include:

- ! *A2F convert an algebraic expression to a standard form. If result is rational, an error results;
- ! *A2K converts an algebraic expression to a kernel. If this is not possible, an error results;
- ! *F2A converts a standard form to an algebraic expression;
- ! *F2Q convert a standard form to a standard quotient;
- ! *K2F convert a kernel to a standard form;
- ! *K2Q convert a kernel to a standard quotient;
- ! *P2F convert a standard power to a standard form;
- ! *P2Q convert a standard power to a standard quotient;
- ! *Q2F convert a standard quotient to a standard form. If the quotient denominator is not 1, an error results;
- ! *Q2K convert a standard quotient to a kernel. If this is not possible, an error results;
- ! *T2F convert a standard term to a standard form
- ! *T2Q convert a standard term to a standard quotient.

17.9.2 Passing Symbolic Mode Values to Algebraic Mode

In order to pass the value of a shared variable from symbolic mode to algebraic mode, the only thing to do is make sure that the value in symbolic mode is a prefix expression. E.g., one uses `(expt (plus a b) 2)` for $(a+b)^2$, or the format `(*sq <standard quotient> t)` as described above. However, if you have been working with parts of a standard form they will probably not be in this form. In that case, you can do the following:

1. If it is a standard quotient, call `PREPSQ` on it. This takes a standard quotient as argument, and returns a prefix expression. Alternatively, you can call `MK! *SQ` on it, which returns a prefix form like `(*SQ <standard quotient> T)` and avoids translation of the expression into a true prefix form.
2. If it is a standard form, call `PREPF` on it. This takes a standard form as argument, and returns the equivalent prefix expression. Alternatively, you can convert it to a standard quotient and then call `MK! *SQ`.
3. If it is a part of a standard form, you must usually first build up a standard form out of it, and then go to step 2. The conversion functions described earlier may be used for this purpose. For example,

- (a) If Z is an expression which is a term, `!*T2F Z` is a standard form.
- (b) If Z is a standard power, `!*P2F Z` is a standard form.
- (c) If Z is a variable, you can pass it direct to algebraic mode.

For example, to pass the leading term of $(a+b)^2$ back to algebraic mode, one could say:

```
y := mk!*sq !*t2q lt numr z;
```

where Y has been declared shared as above. If you now go back to algebraic mode, you can work with Y in the usual way.

17.9.3 Complete Example

The following is the complete code for doing the above steps. The end result will be that the square of the leading term of $(a+b)^2$ is calculated.

```
share x,y;                                % declare X and Y as shared
x := (a+b)^2;                             % store (a+b)^2 in X
symbolic;                                 % transfer to symbolic mode
z := cadr x;                              % store a true standard quotient
                                           % in Z
lt numr z;                                % print the leading term of the
                                           % numerator of Z
y := mk!*sq !*t2q lt numr z;              % store the prefix form of this
                                           % leading term in Y
algebraic;                               % return to algebraic mode
y^2;                                       % evaluate square of the
                                           % leading term of (a+b)^2
```

17.9.4 Defining Procedures for Intermode Communication

If one wishes to define a procedure in symbolic mode for use as an operator in algebraic mode, it is necessary to declare this fact to the system by using the declaration `OPERATOR` in symbolic mode. Thus

```
symbolic operator leadterm;
```

would declare the procedure `LEADTERM` as an algebraic operator. This declaration *must* be made in symbolic mode as the effect in algebraic mode is different. The value of such a procedure must be a prefix form.

The algebraic processor will pass arguments to such procedures in prefix form. Therefore if you want to work with the arguments as standard quotients you must first convert them to that form by using the function `SIMP!*`. This function takes a prefix form as argument and returns the evaluated standard quotient.

For example, if you want to define a procedure `LEADTERM` which gives the leading term of an algebraic expression, one could do this as follows:

```
symbolic operator leadterm; % Declare LEADTERM as a symbolic
                           % mode procedure to be used in
                           % algebraic mode.

symbolic procedure leadterm u; % Define LEADTERM.
  mk!*sq !*t2q lt numr simp!* u;
```

Note that this operator has a different effect than the operator `LTERM`. In the latter case, the calculation is done with respect to the second argument of the operator. In the example here, we simply extract the leading term with respect to the system's choice of main variable.

Finally, if you wish to use the algebraic evaluator on an argument in a symbolic mode definition, the function `REVAL` can be used. The one argument of `REVAL` must be the prefix form of an expression. `REVAL` returns the evaluated expression as a true Lisp prefix form.

17.10 Rlisp '88

Rlisp '88 is a superset of the Rlisp that has been traditionally used for the support of REDUCE. It is fully documented in the book Marti, J.B., "RLISP '88: An Evolutionary Approach to Program Design and Reuse", World Scientific, Singapore (1993). Rlisp '88 adds to the traditional Rlisp the following facilities:

1. more general versions of the looping constructs `for`, `repeat` and `while`;
2. support for a backquote construct;
3. support for active comments;
4. support for vectors of the form `name[index]`;
5. support for simple structures;
6. support for records.

In addition, "-" is a letter in Rlisp '88. In other words, `A-B` is an identifier, not the difference of the identifiers `A` and `B`. If the latter construct is required, it is necessary to put spaces around the `-` character. For compatibility between the two versions of Rlisp, we recommend this convention be used in all symbolic mode programs.

To use Rlisp '88, type on `rlisp88;`. This switches to symbolic mode with the Rlisp '88 syntax and extensions. While in this environment, it is impossible to switch to algebraic mode, or prefix expressions by "algebraic". However, symbolic mode programs written in Rlisp '88 may be run in algebraic mode provided the `rlisp88` package has been loaded. We also expect that many of the extensions defined in Rlisp '88 will migrate to the basic Rlisp over time. To return to traditional Rlisp or to switch to algebraic mode, say `"off rlisp88;"`.

17.11 References

There are a number of useful books which can give you further information about LISP. Here is a selection:

Allen, J.R., "The Anatomy of LISP", McGraw Hill, New York, 1978.

McCarthy J., P.W. Abrahams, J. Edwards, T.P. Hart and M.I. Levin, "LISP 1.5 Programmer's Manual", M.I.T. Press, 1965.

Touretzky, D.S., "LISP: A Gentle Introduction to Symbolic Computation", Harper & Row, New York, 1984.

Winston, P.H. and Horn, B.K.P., "LISP", Addison-Wesley, 1981.

Chapter 18

Calculations in High Energy Physics

A set of REDUCE commands is provided for users interested in symbolic calculations in high energy physics. Several extensions to our basic syntax are necessary, however, to allow for the different data structures encountered.

18.1 High Energy Physics Operators

We begin by introducing three new operators required in these calculations.

18.1.1 \cdot (Cons) Operator

Syntax:

```
(EXPRN1:vector_expression)
    . (EXPRN2:vector_expression):algebraic.
```

The binary \cdot operator, which is normally used to denote the addition of an element to the front of a list, can also be used in algebraic mode to denote the scalar product of two Lorentz four-vectors. For this to happen, the second argument must be recognizable as a vector expression at the time of evaluation. With this meaning, this operator is often referred to as the *dot* operator. In the present system, the index handling routines all assume that Lorentz four-vectors are used, but these routines could be rewritten to handle other cases.

Components of vectors can be represented by including representations of unit vectors in the system. Thus if \mathbf{e}_0 represents the unit vector $(1, 0, 0, 0)$, $(\mathbf{p} \cdot \mathbf{e}_0)$ represents the zeroth component of the four-vector \mathbf{P} . Our metric and notation fol-

low's Bjorken and Drell "Relativistic Quantum Mechanics" (McGraw-Hill, New York, 1965). Similarly, an arbitrary component P may be represented by $(p.u)$. If contraction over components of vectors is required, then the declaration `INDEX` must be used. Thus

```
index u;
```

declares U as an index, and the simplification of

```
p.u * q.u
```

would result in

```
P.Q
```

The metric tensor $g^{\mu\nu}$ may be represented by $(u.v)$. If contraction over U and V is required, then they should be declared as indices.

Errors occur if indices are not properly matched in expressions.

If a user later wishes to remove the index property from specific vectors, he can do it with the declaration `REIND`. Thus `remind v1, ..., vn;` removes the index flags from the variables $V1$ through Vn . However, these variables remain vectors in the system.

18.1.2 G Operator for Gamma Matrices

Syntax:

```
G(ID:identifier[,EXPRN:vector_expression])
   :gamma_matrix_expression.
```

G is an n -ary operator used to denote a product of γ matrices contracted with Lorentz four-vectors. Gamma matrices are associated with fermion lines in a Feynman diagram. If more than one such line occurs, then a different set of γ matrices (operating in independent spin spaces) is required to represent each line. To facilitate this, the first argument of G is a line identification identifier (not a number) used to distinguish different lines.

Thus

```
g(l1,p) * g(l2,q)
```

denotes the product of $\gamma.p$ associated with a fermion line identified as $L1$, and $\gamma.q$ associated with another line identified as $L2$ and where p and q are Lorentz

four-vectors. A product of γ matrices associated with the same line may be written in a contracted form.

Thus

$$g(l1, p1, p2, \dots, p3) = g(l1, p1) * g(l1, p2) * \dots * g(l1, p3) \quad .$$

The vector A is reserved in arguments of G to denote the special γ matrix γ^5 . Thus

$$\begin{aligned} g(l, a) &= \gamma^5 && \text{associated with the line } L \\ g(l, p, a) &= \gamma \cdot p \times \gamma^5 && \text{associated with the line } L. \end{aligned}$$

γ^μ (associated with the line L) may be written as $g(l, u)$, with U flagged as an index if contraction over U is required.

The notation of Bjorken and Drell is assumed in all operations involving γ matrices.

18.1.3 EPS Operator

Syntax:

$$\text{EPS}(\text{EXPRN1:vector_expression}, \dots, \text{EXPRN4:vector_exp}) \\ \text{:vector_exp.}$$

The operator EPS has four arguments, and is used only to denote the completely antisymmetric tensor of order 4 and its contraction with Lorentz four-vectors. Thus

$$\epsilon_{ijkl} = \begin{cases} +1 & \text{if } i, j, k, l \text{ is an even permutation of } 0, 1, 2, 3 \\ -1 & \text{if } i, j, k, l \text{ is an odd permutation of } 0, 1, 2, 3 \\ 0 & \text{otherwise} \end{cases}$$

A contraction of the form $\epsilon_{ij\mu\nu} p_\mu q_\nu$ may be written as $\text{eps}(i, j, p, q)$, with I and J flagged as indices, and so on.

18.2 Vector Variables

Apart from the line identification identifier in the G operator, all other arguments of the operators in this section are vectors. Variables used as such must be declared so by the type declaration VECTOR, for example:

```
vector p1, p2;
```

declares $P1$ and $P2$ to be vectors. Variables declared as indices or given a mass are automatically declared vector by these declarations.

18.3 Additional Expression Types

Two additional expression types are necessary for high energy calculations, namely

18.3.1 Vector Expressions

These follow the normal rules of vector combination. Thus the product of a scalar or numerical expression and a vector expression is a vector, as are the sum and difference of vector expressions. If these rules are not followed, error messages are printed. Furthermore, if the system finds an undeclared variable where it expects a vector variable, it will ask the user in interactive mode whether to make that variable a vector or not. In batch mode, the declaration will be made automatically and the user informed of this by a message.

Examples:

Assuming P and Q have been declared vectors, the following are vector expressions

$$\begin{aligned} &P \\ &2*Q/3 \\ &2*x*y*p - p.Q*Q/(3*Q.Q) \end{aligned}$$

whereas $p*q$ and p/q are not.

18.3.2 Dirac Expressions

These denote those expressions which involve γ matrices. A γ matrix is implicitly a 4×4 matrix, and so the product, sum and difference of such expressions, or the product of a scalar and Dirac expression is again a Dirac expression. There are no Dirac variables in the system, so whenever a scalar variable appears in a Dirac expression without an associated γ matrix expression, an implicit unit 4 by 4 matrix is assumed. For example, $g(l,p) + m$ denotes $g(l,p) + m*(unit\ 4\ by\ 4\ matrix)$. Multiplication of Dirac expressions, as for matrix expressions, is of course non-commutative.

18.4 Trace Calculations

When a Dirac expression is evaluated, the system computes one quarter of the trace of each γ matrix product in the expansion of the expression. One quarter of each trace is taken in order to avoid confusion between the trace of the scalar M , say, and M representing $M * \langle \text{unit } 4 \text{ by } 4 \text{ matrix} \rangle$. Contraction over indices occurring in such expressions is also performed. If an unmatched index is found in such an expression, an error occurs.

The algorithms used for trace calculations are the best available at the time this system was produced. For example, in addition to the algorithm developed by Chisholm for contracting indices in products of traces, REDUCE uses the elegant algorithm of Kahane for contracting indices in γ matrix products. These algorithms are described in Chisholm, J. S. R., *Il Nuovo Cimento X*, 30, 426 (1963) and Kahane, J., *Journal Math. Phys.* 9, 1732 (1968).

It is possible to prevent the trace calculation over any line identifier by the declaration `NOSPUR`. For example,

```
nospur l1,l2;
```

will mean that no traces are taken of γ matrix terms involving the line numbers `L1` and `L2`. However, in some calculations involving more than one line, a catastrophic error

```
This NOSPUR option not implemented
```

can occur (for the reason stated!) If you encounter this error, please let us know!

A trace of a γ matrix expression involving a line identifier which has been declared `NOSPUR` may be later taken by making the declaration `SPUR`.

See also the CVIT package for an alternative mechanism (chapter [16.15](#)).

18.5 Mass Declarations

It is often necessary to put a particle “on the mass shell” in a calculation. This can, of course, be accomplished with a `LET` command such as

```
let p.p= m^2;
```

but an alternative method is provided by two commands `MASS` and `MSHELL`. `MASS` takes a list of equations of the form:

$$\langle \text{vector variable} \rangle = \langle \text{scalar variable} \rangle$$

for example,

```
mass p1=m, q1=mu;
```

The only effect of this command is to associate the relevant scalar variable as a mass with the corresponding vector. If we now say

```
mshell <vector variable>, ... , <vector variable><terminator>
```

and a mass has been associated with these arguments, a substitution of the form

$$\langle \text{vector variable} \rangle \cdot \langle \text{vector variable} \rangle = \langle \text{mass} \rangle^2$$

is set up. An error results if the variable has no preassigned mass.

18.6 Example

We give here as an example of a simple calculation in high energy physics the computation of the Compton scattering cross-section as given in Bjorken and Drell Eqs. (7.72) through (7.74). We wish to compute the trace of

$$\frac{\alpha^2}{2} \left(\frac{k'}{k} \right)^2 \left(\frac{\gamma \cdot p_f + m}{2m} \right) \left(\frac{\gamma \cdot e' \gamma \cdot e \gamma \cdot k_i}{2k \cdot p_i} + \frac{\gamma \cdot e \gamma \cdot e' \gamma \cdot k_f}{2k' \cdot p_i} \right) \\ \left(\frac{\gamma \cdot p_i + m}{2m} \right) \left(\frac{\gamma \cdot k_i \gamma \cdot e \gamma \cdot e'}{2k \cdot p_i} + \frac{\gamma \cdot k_f \gamma \cdot e' \gamma \cdot e}{2k' \cdot p_i} \right)$$

where k_i and k_f are the four-momenta of incoming and outgoing photons (with polarization vectors e and e' and laboratory energies k and k' respectively) and p_i , p_f are incident and final electron four-momenta.

Omitting therefore an overall factor $\frac{\alpha^2}{2m^2} \left(\frac{k'}{k} \right)^2$ we need to find one quarter of the trace of

$$(\gamma \cdot p_f + m) \left(\frac{\gamma \cdot e' \gamma \cdot e \gamma \cdot k_i}{2k \cdot p_i} + \frac{\gamma \cdot e \gamma \cdot e' \gamma \cdot k_f}{2k' \cdot p_i} \right) \times \\ (\gamma \cdot p_i + m) \left(\frac{\gamma \cdot k_i \gamma \cdot e \gamma \cdot e'}{2k \cdot p_i} + \frac{\gamma \cdot k_f \gamma \cdot e' \gamma \cdot e}{2k' \cdot p_i} \right)$$

A straightforward REDUCE program for this, with appropriate substitutions (using P1 for p_i , PF for p_f , KI for k_i and KF for k_f) is


```

on div; % this gives output in same form as Bjorken and Drell.
mass ki= 0, kf= 0, pl= m, pf= m; vector e,ep;
% if e is used as a vector, it loses its scalar identity
%      as the base of natural logarithms.
mshell ki,kf,pl,pf;
let pl.e= 0, pl.ep= 0, pl.pf= m^2+ki.kf, pl.ki= m*k,pl.kf=
    m*kp, pf.e= -kf.e, pf.ep= ki.ep, pf.ki= m*kp, pf.kf=
    m*k, ki.e= 0, ki.kf= m*(k-kp), kf.ep= 0, e.e= -1,
    ep.ep=-1;
operator gp;
for all p let gp(p)= g(l,p)+m;
comment this is just to save us a lot of writing;
gp(pf)*(g(l,ep,e,ki)/(2*ki.pl) + g(l,e,ep,kf)/(2*kf.pl))
    * gp(pl)*(g(l,ki,e,ep)/(2*ki.pl) + g(l,kf,ep,e)/
    (2*kf.pl))$
write "The Compton cxn is ",ws;

```

(We use P_1 instead of P_I in the above to avoid confusion with the reserved variable P_I).

This program will print the following result

$$\begin{array}{ccccccc} & 2 & 1 & -1 & 1 & -1 & \\ \text{The Compton cxn is } & 2 * E.EP & + & \frac{1}{2} * K * KP & + & \frac{1}{2} * K & * KP - 1 \end{array}$$

18.7 Extensions to More Than Four Dimensions

In our discussion so far, we have assumed that we are working in the normal four dimensions of QED calculations. However, in most cases, the programs will also work in an arbitrary number of dimensions. The command

`vecdim \langle expression \rangle \langle terminator \rangle`

sets the appropriate dimension. The dimension can be symbolic as well as numerical. Users should note however, that the EPS operator and the γ_5 symbol (A) are not properly defined in other than four dimensions and will lead to an error if used.

Chapter 19

REDUCE and Rlisp Utilities

REDUCE and its associated support language system Rlisp include a number of utilities which have proved useful for program development over the years. The following are supported in most of the implementations of REDUCE currently available.

19.1 The Standard Lisp Compiler

Many versions of REDUCE include a Standard Lisp compiler that is automatically loaded on demand. You should check your system specific user guide to make sure you have such a compiler. To make the compiler active, the switch `COMP` should be turned on. Any further definitions input after this will be compiled automatically. If the compiler used is a derivative version of the original Griss-Hearn compiler, (M. L. Griss and A. C. Hearn, "A Portable LISP Compiler", *SOFTWARE — Practice and Experience* 11 (1981) 541-605), there are other switches that might also be used in this regard. However, these additional switches are not supported in all compilers. They are as follows:

`PLAP` If ON, causes the printing of the portable macros produced by the compiler;

`PGWD` If ON, causes the printing of the actual assembly language instructions generated from the macros;

`PWRDS` If ON, causes a statistic message of the form
`<function> COMPILED, <words> WORDS, <words> LEFT`
to be printed. The first number is the number of words of binary program space the compiled function took, and the second number the number of words left unused in binary program space.

19.2 Fast Loading Code Generation Program

In most versions of REDUCE, it is possible to take any set of Lisp, Rlisp or REDUCE commands and build a fast loading version of them. In Rlisp or REDUCE, one does the following:

```
faslout <filename>;  
<commands or IN statements>  
faslend;
```

To load such a file, one uses the command `LOAD`, e.g. `load foo;` or `load foo,bah;`

This process produces a fast-loading version of the original file. In some implementations, this means another file is created with the same name but a different extension. For example, in PSL-based systems, the extension is `b` (for binary). In CSL-based systems, however, this process adds the fast-loading code to a single file in which all such code is stored. Particular functions are provided by CSL for managing this file, and described in the CSL user documentation.

In doing this build, as with the production of a Standard Lisp form of such statements, it is important to remember that some of the commands must be instantiated during the building process. For example, macros must be expanded, and some property list operations must happen. The REDUCE sources should be consulted for further details on this.

To avoid excessive printout, input statements should be followed by a `$` instead of the semicolon. With `LOAD` however, the input doesn't print out regardless of which terminator is used with the command.

If you subsequently change the source files used in producing a fast loading file, don't forget to repeat the above process in order to update the fast loading file correspondingly. Remember also that the text which is read in during the creation of the fast load file, in the compiling process described above, is *not* stored in your REDUCE environment, but only translated and output. If you want to use the file just created, you must then use `LOAD` to load the output of the fast-loading file generation program.

When the file to be loaded contains a complete package for a given application, `LOAD_PACKAGE` rather than `LOAD` should be used. The syntax is the same. However, `LOAD_PACKAGE` does some additional bookkeeping such as recording that this package has now been loaded, that is required for the correct operation of the system.

19.3 The Standard Lisp Cross Reference Program

CREF is a Standard Lisp program for processing a set of Standard LISP function definitions to produce:

1. A “summary” showing:
 - (a) A list of files processed;
 - (b) A list of “entry points” (functions which are not called or are only called by themselves);
 - (c) A list of undefined functions (functions called but not defined in this set of functions);
 - (d) A list of variables that were used non-locally but not declared `GLOBAL` or `FLUID` before their use;
 - (e) A list of variables that were declared `GLOBAL` but not used as `FLUIDS`, i.e., bound in a function;
 - (f) A list of `FLUID` variables that were not bound in a function so that one might consider declaring them `GLOBALS`;
 - (g) A list of all `GLOBAL` variables present;
 - (h) A list of all `FLUID` variables present;
 - (i) A list of all functions present.
2. A “global variable usage” table, showing for each non-local variable:
 - (a) Functions in which it is used as a declared `FLUID` or `GLOBAL`;
 - (b) Functions in which it is used but not declared;
 - (c) Functions in which it is bound;
 - (d) Functions in which it is changed by `SETQ`.
3. A “function usage” table showing for each function:
 - (a) Where it is defined;
 - (b) Functions which call this function;
 - (c) Functions called by it;
 - (d) Non-local variables used.

The program will also check that functions are called with the correct number of arguments, and print a diagnostic message otherwise.

The output is alphabetized on the first seven characters of each function name.

19.3.1 Restrictions

Algebraic procedures in REDUCE are treated as if they were symbolic, so that algebraic constructs will actually appear as calls to symbolic functions, such as AEVAL.

19.3.2 Usage

To invoke the cross reference program, the switch CREF is used. `on cref` causes the cref program to load and the cross-referencing process to begin. After all the required definitions are loaded, `off cref` will cause the cross-reference listing to be produced. For example, if you wish to cross-reference all functions in the file `tst.red`, and produce the cross-reference listing in the file `tst.crf`, the following sequence can be used:

```
out "tst.crf";
on cref;
in "tst.red"$
off cref;
shut "tst.crf";
```

To process more than one file, more `IN` statements may be added before the call of `off cref`, or the `IN` statement changed to include a list of files.

19.3.3 Options

Functions with the flag `NOLIST` will not be examined or output. Initially, all Standard Lisp functions are so flagged. (In fact, they are kept on a list `NOLIST!*`, so if you wish to see references to *all* functions, then CREF should be first loaded with the command `load cref`, and this variable then set to `NIL`).

It should also be remembered that any macros with the property list flag `EXPAND`, or, if the switch `FORCE` is on, without the property list flag `NOEXPAND`, will be expanded before the definition is seen by the cross-reference program, so this flag can also be used to select those macros you require expanded and those you do not.

19.4 Prettyprinting Reduce Expressions

REDUCE includes a module for printing REDUCE syntax in a standard format. This module is activated by the switch `PRET`, which is normally off.

Since the system converts algebraic input into an equivalent symbolic form, the

printing program tries to interpret this as an algebraic expression before printing it. In most cases, this can be done successfully. However, there will be occasional instances where results are printed in symbolic mode form that bears little resemblance to the original input, even though it is formally equivalent.

If you want to prettyprint a whole file, say `off output, msg; MSG` and (hopefully) only clean output will result. Unlike `DEFN`, input is also evaluated with `PRET` on.

19.5 Prettyprinting Standard Lisp S-Expressions

`REDUCE` includes a module for printing S-expressions in a standard format. The Standard Lisp function for this purpose is `Prettyprint` which takes a Lisp expression and prints the formatted equivalent.

Users can also have their `REDUCE` input printed in this form by use of the switch `DEFN`. This is in fact a convenient way to convert `REDUCE` (or `Rlisp`) syntax into Lisp. `off msg;` will prevent warning messages from being printed.

NOTE: When `DEFN` is on, input is not evaluated.

Chapter 20

Maintaining REDUCE

Since January 1, 2009 REDUCE is Open Source Software. It is hosted at

<http://reduce-algebra.sourceforge.net/>

We mention here three ways in which REDUCE is maintained. The first is the collection of queries, observations and bug-reports. All users are encouraged to subscribe to the [mailing list](#) that Sourceforge.net provides so that they will receive information about updates and concerns. Also on SourceForge there is a [bug tracker](#) and a [forum](#). The expectation is that the maintainers and keen users of REDUCE will monitor those and try to respond to issues. However these resources are not there to seek answers to Maths homework problems - they are intended specifically for issues to do with the use and support of REDUCE.

The second level of support is provided by the fact that all the sources of REDUCE are available, so any user who is having difficulty either with a bug or understanding system behaviour can consult the code to see if (for instance) comments in it clarify something that was unclear from the regular documentation.

The source files for REDUCE are available on SourceForge in the [Subversion repository](#). Check the "code/SVN" tab on the SourceForge page to find instructions for using a Subversion client to fetch the most up to date copy of everything. From time to time there may be one-file archives of a snapshot of the sources placed in the download area on SourceForge, and eventually some of these may be marked as "stable" releases, but at present it is recommended that developers use a copy from the Subversion repository.

The files fetched there come with a directory called "trunk" that holds the main current REDUCE, and one called "branches" that is reserved for future experimental versions. All the files that we have for creating help files and manuals should also be present in the files you fetch.

The packages that make up the source for the algebraic capabilities of REDUCE are in the “packages” sub-directory, and often there are test files for a package present there and especially for contributed packages there will be documentation in the form of a \LaTeX file. Although REDUCE is coded in its own language many people in the past have found that it does not take too long to start to get used to it.

In various cases even fairly “ordinary end users” may wish to fetch the source version of REDUCE and compile it all for themselves. This may either be because they need the benefit of a bug-fix only recently checked into the subversion repository or because no pre-compiled binary is available for the particular computer and operating system they use. This latter is to some extent unavoidable since REDUCE can run on both 32 and 64-bit Windows, the various MacOSX options (eg Intel and Powerpc), many different distributions of Linux, some BSD variants and Solaris (at least). It is not practically feasible for us to provide a constant stream of up to date ready-built binaries for all these.

There are instructions for compiling REDUCE present at the top of the trunk source tree. Usually the hardest issue seems to be ensuring that your computer has an adequate set of development tools and libraries installed before you start, but once that is sorted out the hope is that the compilation of REDUCE should proceed uneventfully if sometimes tediously.

In a typical Open Source way the hope is that some of those who build REDUCE from source or explore the source (out of general interest or to pursue an understanding of some bug or detail) will transform themselves into contributors or developers which moves on to the third level of support.

At this third level any user can contribute proposals for bug fixes or extensions to REDUCE or its documentation. It might be valuable to collect a library of additional user-contributed examples illustrating the use of the system too. To do this first ensure that you have a fully up to date copy of the sources from Subversion, and then depending on just what sort of change is being proposed provide the updates to the developers via the SourceForge bug tracker or other route. In time we may give more concrete guidance about the format of changes that will be easiest to handle. It is obviously important that proposed changes have been properly tested and that they are accompanied with a clear explanation of why they are of benefit. A specific concern here is that in the past fixes to a bug in one part of REDUCE have had bad effects on some other applications and packages, so some degree of caution is called for. Anybody who develops a significant whole new package for REDUCE is encouraged to make the developers aware so that it can be considered for inclusion.

So the short form explanation about Support and Maintenance is that it is mainly focussed around the SourceForge system. That if discussions about bugs, requirements or issues are conducted there then all users and potential users of REDUCE will be able to benefit from reviewing them, and the Sourceforge mailing lists,

tracker, forums and wiki will grow to be both a static repository of answers to common questions, an active set of locations to to get new issues looked at and a focus for guiding future development.

Appendix A

Reserved Identifiers

We list here all identifiers that are normally reserved in REDUCE including names of commands, operators and switches initially in the system. Excluded are words that are reserved in specific implementations of the system.

Commands	ALGEBRAIC ANTISYMMETRIC ARRAY BYE CLEAR CLEARRULES COMMENT CONT DECOMPOSE DEFINE DEPEND DISPLAY ED EDITDEF END EVEN FACTOR FOR FORALL FOREACH GO GOTO IF IN INDEX INFIX INPUT INTEGER KORDER LET LINEAR LISP LISTARGP LOAD LOAD_PACKAGE MASS MATCH MATRIX MSHELL NODEPEND NONCOM NONZERO NOSPUR ODD OFF ON OPERATOR ORDER OUT PAUSE PRECEDENCE PRINT_PRECISION PROCEDURE QUIT REAL REMEMBER REMFAC REMIND RETRY RETURN SAVEAS SCALAR SETMOD SHARE SHOWTIME SHUT SPUR SYMBOLIC SYMMETRIC VECDIM VECTOR WEIGHT WRITE WTLEVEL
Boolean Operators	EVENP FIXP FREEOF NUMBERP ORDP PRIMEP
Infix Operators	: = > = > < = < = > + - * / ^ ** . WHERE SETQ OR AND MEMBER MEMQ EQUAL NEQ EQ GEQ GREATERP LEQ LESSP PLUS DIFFERENCE MINUS TIMES QUOTIENT RECIP EXPT CONS
Numerical Operators	ABS ACOS ACOSH ACOT ACOTH ACSC ACSCH AIRY_AI AIRY_AIPRIME AIRY_BI AIRY_BIPRIME ASEC ASECH ASIN ASINH ATAN ATANH ATAN2

BERNOULLI BESSELI BESSELJ BESSELK
 BESSELY BETA COS COSH COT COTH CSC CSCH
 CSCH EXP FACTORIAL FIX FLOOR GAMMA
 HANKEL1 HANKEL2 HYPOT IBETA IGAMMA
 KUMMER KUMMERU LERCH.PHI LN LOG LOGB
 LOG10 LOMMEL1 LOMMEL2 NEXTPRIME
 POCHHAMMER POLYGAMMA PSI ROUND SEC SECH
 SIN SINH SQRT STRUVEH STRUVEL TAN TANH
 WHITTAKER WHITTAKERU ZETA

Prefix Operators

APPEND ARBCOMPLEX ARBINT ARGLENGTH
 CEILING CI COEFF COEFFN COFACTOR CONJ
 CONTINUED_FRACTION DEG DEN DET DF DILOG
 EI EPS ERF EXPAND_CASES FACTORIZE
 FIBONACCI FIBONACCIP FIRST GCD G
 HYPERGEOMETRIC IMPART INT INTERPOL LCM
 LCOF LENGTH LHS LINELENGTH LIST LPOWER
 LTERM MAINVAR MAP MAT MATEIGEN MAX
 MEIJERG MIN MKID MOTZKIN NULLSPACE NUM
 ONE_OF PART PF PRECISION PROD RANDOM
 RANDOM_NEW_SEED RANK REDERR REDUCT
 REMAINDER REPART REST RESULTANT REVERSE
 RHS ROOT_OF ROOT_VAL SECOND SELECT SET
 SHOWRULES SI SIGN SOLVE SOLIDHARMONICY
 SPHERICALHARMONICY STRUCTR SUB SUM
 THIRD TOTALDEG TP TRACE VARNAME

Reserved Variables

LINE ASSUMPTIONS CARD_NO CATALAN E
 EULER_GAMMA EVAL_MODE FORT_WIDTH
 GOLDEN_RATIO HIGH_POW I INFINITY K!*
 KHINCHIN LOW_POW NIL PI REQUIREMENTS
 ROOT_MULTPLICITIES T

Switches

ADJPREC ALGINT ALLBRANCH ALLFAC
 ALLOWDFINT ARBVARs BALANCE_MOD BEZOUT
 BFSPACE COMBINEEXPT COMBINELOGS
 COMMUTEDF COMP COMPLEX CRAMER CREF DEFN
 DEMO DFINT DIV ECHO ERRCONT EVALHSEQP
 EXP EXPANDDF EXPANDLOGS EZGCD FACTOR
 FAILHARD FORT FORTUPPER FULLROOTS GCD
 IFACOR INT INTSTR LCM LIST LISTARGS MCD
 MODULAR MSG MULTIPLICITIES NAT NERO
 NOCOMMUTEDF NOCONVERT NOLNR NOSPLIT
 OUTPUT PERIOD PRECISE PRECISE_COMPLEX
 PRET PRI RAT RATARG RATIONAL RATIONALIZE

RATPRI REVPRI RLISP88 ROUNDALL ROUNDBF
ROUNDED SAVESTRUCTR SIMPNONCOMDF
SOLVESINGULAR TIME TRA TRFAC TRIGFORM
TRINT VAROPT

Other Reserved Ids

BEGIN DO THEN EXPR FEXPR INPUT LAMBDA
LISP MACRO PRODUCT REPEAT SMACRO SUM
THEN UNTIL WHEN WHILE WS

Appendix B

Changes since Version 3.8

New packages assert bibasis breduce cdiff clprl gcref guardian lessons libreduce lpdo redfront reduce4 utf8

Package rlisp Support for namespaces (::)

Default value in switch statement

Support for utf8 characters

Package alg New switch precise_complex

Improvements for switch combineexpt (exptchk.red)

Operators si, ci, gamma, igamma, psi, polygamma, beta, ibeta, euler, bernoulli, pochhammer, lerch_phi, polylog, zeta, besselj, bessely, besseli, bessell, hankel1, hankel2, kummerM, kummerU, struveh, struvel, lommel1, lommel2, whittakerm, whittakerw, Airy_Ai, Airy_Bi, Airy_AiPrime, Airy_biprime, binomial, solidharmonic, sphericalharmonic, fibonacci, fibonaccip, motzkin, hypergeometric, MeijerG now known as part of the core, as well as constants catalan, euler_gamma, golden_ratio, khinchin

Package poly Improvements for differentiation: new switches expanddf, allowdfint etc (from odesolve)

Package solve polyp

Package modsr legendre_symbol

Index

- !!FLIM, [637](#)
- !!NFPD, [637](#)
- !*CSYSTEMS global (AVECTOR), [211](#)
- !*MODE, [333](#)
- !\$# in GENTRAN, [364](#)
- !\$0 subprogram name, [364](#)
- !\$n parameters, [364](#)
- ><
 - 3-D vector, [585](#)
 - diphthong, [585](#)
- |_ operator, [293](#), [306](#)
- *
 - 3-D vector, [585](#)
 - algebraic numbers, [201](#)
 - power series, [688](#)
 - vector, [209](#)
- **
 - power series, [688](#)
- +
 - 3-D vector, [585](#)
 - algebraic numbers, [201](#)
 - power series, [688](#)
 - vector, [209](#)
- - 3-D vector, [585](#)
 - power series, [688](#)
 - vector, [209](#)
- .
 - (CONS), [44](#)
- /
 - 3-D vector, [585](#)
 - algebraic numbers, [201](#)
 - power series, [688](#)
 - vector, [209](#)
- ::=, [334](#), [424](#)
- ::=:, [334](#), [424](#)
- :=, [334](#), [424](#)
- ;BEGIN; marker, [353](#), [404](#)
- ;END; marker, [353](#), [404](#)
- @
 - partial differentiation, [306](#)
 - tangent vector, [306](#)
- @ operator, [289](#)
- #
 - Hodge-* operator, [293](#), [306](#)
- ^
 - 3-D vector, [585](#)
 - exterior multiplication, [288](#), [306](#)
- _| operator, [292](#), [306](#)
- _LINE_, [152](#)
- 3j and 6j symbols, [661](#)
- ABS, [63](#), [132](#)
- ACOS, [67](#), [71](#)
- ACOSH, [67](#), [71](#)
- ACOT, [67](#), [71](#)
- ACOTH, [67](#), [71](#)
- ACSC, [67](#), [71](#)
- ACSCH, [67](#), [71](#)
- ADJ, [602](#)
- ADJPREC, [127](#)
- AEVAL, [397](#)
- Airy functions, [661](#)
- AIRY_AI, [67](#)
- Airy_Ai, [661](#)
- AIRY_AIPRIME, [67](#)
- Airy_Aiprime, [661](#)
- AIRY_BI, [67](#)
- Airy_Bi, [661](#)
- AIRY_BIPRIME, [67](#)
- Airy_Biprime, [661](#)
- ALGEBRAIC, [757](#)
- Algebraic mode, [757](#), [762](#), [763](#)

- algebraic number fields, 201
- algebraic numbers, 201
- ALGINT, 176
- ALL! *, 329, 367
- ALLFAC, 99, 101
- ALLOWDFINT, 74
- ansatz of symmetry generator, 656
- ANTICOM, 601
- ANTICOMM, 601
- ANTICOMMUTE, 602
- ANTISYMMETRIC, 92
- APPEND, 44
- APPLYSYM, 177
- approximation, 72
- ARBVARS, 86
- ARGLength, 112
- ARNUM, 201
- ARNUM package, 201
- ARNUM package
 - example, 202, 203, 205
- ARRAY, 339
- ARRAY, 59
- ASEC, 67, 71
- ASECH, 67, 71
- ASIN, 67, 71
- ASINH, 67, 71
- ASSERT, 207
- Assignment, 48, 51, 55, 760, 763
- ASSIST, 208
- ASSUMPTIONS, 88
- Asymptotic command, 137, 149
- ATAN, 67, 71, 76
- ATAN2, 67, 71
- ATANH, 67, 71
- Automatic Circuitry Code Generator, 388
- AVEC function, 209
- AVECTOR, 208
- AVECTOR package, 208
- AVECTOR package
 - example, 212–214
- BALANCED_MOD, 127
- Barnes, Alan, 680
- BEGIN . . . END, 54–56
- Bernoulli, 71, 661
- Bernoulli numbers, 71, 661
- Bessel functions, 661
- BESSELI, 67
- Besseli, 661
- BESSELJ, 67
- BesselJ, 661
- BESSELK, 67
- BesselK, 661
- BESSELY, 67
- Bessely, 661
- BETA, 67
- Beta, 661
- Beta function, 661
- BEZOUT, 119
- BFSPACE, 126
- BIBASIS, 215
- Binomial, 661
- Binomial coefficients, 661
- Block, 54, 56
- BNDEQ! *, 295
- BOOLEAN, 222
- Boolean, 39
- booleans, 352
- BOUNDS, 553
- BYE, 61
- C, 326, 328
- C(I), 651
- CALI, 226
- Call by value, 168, 170
- call-by-address, 409
- call-by-value, 409
- CAMAL, 227
- Canonical form, 95
- CARD_NO, 105
- cartesian coordinates, 584
- CATALAN, 32
- caveats
 - TAYLOR package, 676
- CCURRIND! *, 379
- CDIFF, 240
- CEILING, 64
- chain rule, 291

- CHANGEVR, [241](#)
- Character set, [29](#)
- Chebyshev fit, [553](#)
- Chebyshev polynomials, [661](#)
- ChebyshevT, [661](#)
- ChebyshevU, [661](#)
- CI, [67](#)
- CLEAR, [140](#), [143](#)
- CLEARPHYSOP, [598](#)
- CLEARRULES, [144](#)
- Clebsch Gordan coefficients, [661](#)
- Clebsch_Gordan, [661](#)
- CLINELEN
 - CLINELEN
 - *, [425](#)
- CLINELEN! *, [380](#)
- code generation, [396](#)
- code templates, [352](#)
- COEFF, [110](#)
- Coefficient, [125–128](#)
- COEFFN, [111](#)
- COFACTOR, [163](#)
- coframe, [295](#), [299](#)
- COFRAME
 - WITH METRIC, [300](#)
 - WITH SIGNATURE, [300](#)
- COFRAME command, [306](#)
- COLLECT, [50](#)
- COMBINEEXPT, [70](#)
- COMBINELOGS, [69](#)
- COMM, [601](#)
- COMM, [650](#)
- Command, [59](#)
- Command terminator, [151](#)
- COMMENT, [34](#)
- comments
 - in GENTRAN, [341](#), [353](#), [402](#)
- COMMUTE, [602](#)
- COMMUTEDF, [73](#)
- COMP, [777](#)
- COMPACT, [245](#)
- Compiler, [777](#)
- COMPLEX, [331](#), [349](#)
- COMPLEX, [128](#), [635](#)
- Complex coefficient, [128](#)
- complex numbers, [349](#)
- COMPLEX*16, [331](#)
- Compound statement, [53](#), [55](#)
- Conditional statement, [49](#), [50](#)
- CONJ, [64](#)
- Constructor, [763](#)
- CONT, [157](#)
- continuation lines, [343](#)
- CONTINUED_FRACTION, [72](#)
- CONTRACT, [600](#)
- COORDINATES operator, [210](#)
- COORDS vector, [210](#)
- COS, [67](#), [71](#)
- COSH, [67](#), [71](#)
- COT, [67](#), [71](#)
- COTH, [67](#), [71](#)
- CR! *, [341](#), [402](#)
- CRACK, [246](#)
- CRAMER, [82](#), [161](#)
- CREF, [779](#), [780](#)
- CRESYS, [650](#), [651](#)
- CROSS
 - vector, [209](#)
- cross product, [209](#), [587](#)
- Cross reference, [779](#)
- CSC, [67](#), [71](#)
- CSCH, [67](#), [71](#)
- CURL
 - operator, [210](#)
- curl
 - vector field, [210](#)
- curl operator, [588](#)
- CVIT, [247](#)
- cylindrical coordinates, [584](#)
- d
 - exterior differentiation, [306](#)
- Declaration, [59](#)
- DECLARE, [394](#), [426](#)
- DECLARE function, [336](#), [399](#), [424](#)
- DECOMPOSE, [120](#)
- defaults
 - TAYLOR package, [676](#)

- DEFINE, [61](#), [62](#)
- definite integration (simple), [212](#)
- DEFINT, [256](#)
- DEFINT function, [212](#)
- DEFLINEINT function, [213](#)
- DEFN, [762](#), [781](#)
- DEFPOLY statement, [202](#)
- DEFTYPE
 - DEFTYPE
 - *, [426](#)
- DEFTYPE! *, [340](#), [341](#), [376](#), [379](#)
- DEG, [121](#)
- Degree, [122](#)
- DELSQ
 - operator, [210](#)
- delsq operator, [588](#)
- DEMO, [60](#)
- DEN, [113](#), [122](#)
- DEPEND, [89](#), [93](#)
- DEPEND statement, [589](#)
- DEQ (I), [651](#)
- derivative
 - variational, [295](#)
- derivatives, [274](#)
- DESIR, [266](#)
- DET, [95](#), [161](#)
- determinant
 - in DETM!*, [300](#)
- DETM! *, [300](#)
- DF, [73](#), [75](#)
- DFINT, [74](#)
- DFPART, [274](#)
- differential geometry, [286](#)
- Differentiation, [73](#), [75](#), [93](#)
- differentiation
 - partial, [289](#)
 - vector, [210](#)
- Digamma, [661](#)
- Digamma function, [661](#)
- DILOG, [67](#), [76](#)
- Dilog, [661](#)
- Dilogarithm function, [661](#)
- dimension, [289](#)
- Dirac γ matrix, [770](#)
- DISPLAY, [156](#)
- Display, [95](#)
- DISPLAYFRAME command, [303](#), [306](#)
- Displaying structure, [108](#)
- DIV, [100](#), [125](#)
- DIV
 - operator, [210](#)
- div operator, [588](#)
- divergence
 - vector field, [210](#)
- DLINEINT, [590](#)
- DO, [50–52](#)
- Dollar sign, [47](#)
- DOT, [599](#)
- DOT
 - vector, [209](#)
- Dot product, [209](#), [769](#)
- dot product, [587](#)
- DOTGRAD operator, [588](#)
- dotgrad operator, [588](#)
- DOUBLE PRECISION, [331](#), [340](#)
- DOUBLE switch, [331](#), [375](#), [425](#)
- DUMMY, [279](#)
- DUMMY package, [279](#)
- DVINT, [590](#)
- DVOLINT, [590](#)

- E, [32](#)
- e, [351](#)
- ECHO, [151](#)
- ED, [155](#), [156](#)
- EDITDEF, [157](#)
- Ei, [67](#)
- EllipticE, [661](#)
- EllipticF, [661](#)
- EllipticTheta, [661](#)
- END, [61](#)
- EPS, [303](#), [771](#)
- EPS
 - Levi-Civita tensor, [306](#)
- Equation, [40](#), [41](#)
- ERF, [76](#)
- ERRCONT, [155](#)
- errors

- TAYLOR package, 677
- ETA (ALFA), 651
- euclidean metric, 300
- Euler, 71, 661
- Euler numbers, 71, 661
- Euler polynomials, 661
- EULER_GAMMA, 32
- EulerP, 661
- EVAL, 333, 394, 396, 424, 426
- EVAL_MODE, 757
- EVALHSEQP, 41
- EVEN, 89
- Even operator, 89
- EVENP, 39
- EXCALC, 286
- EXCALC package, 286
- EXCALC package
 - example, 288–290, 292, 293, 295–297, 300–304, 307
- Exclamation mark, 29
- EXCLUDE, 632
- EXDEGREE, 306
- EXDEGREE command, 288
- EXP, 351
- EXP, 67, 71, 76, 114, 117
- EXPAND_CASES, 83
- EXPANDDF, 74
- EXPANDLOGS, 69
- EXPR, 761
- Expression, 37
- exterior calculus, 286
- exterior differentiation, 290
- exterior form
 - declaration, 287
 - vector, 287
 - with indices, 287, 296
- exterior product, 288, 305
- EZGCD, 117
- FACTOR, 99, 114, 115
- FACTORIAL, 64, 171
- Factorization, 114
- FACTORIZE, 115
- false, 352
- Fast loading of code, 778
- FDOMAIN command, 289, 306
- FEXPR, 762
- Fibonacci, 71
- Fibonacci numbers, 71
- Fibonacci Polynomials, 71
- FibonacciP, 71
- FIDE, 318
- File handling, 151
- files
 - in GENTRAN, 355, 370
- FIRST, 44
- FIRSTROOT, 633
- FIX, 64
- FIXP, 39
- FLOOR, 65
- FOR, 57
- FOR ALL, 138, 139
- FOR EACH, 51, 52, 761
- FOR loop, 420
- FORDER command, 304, 306
- FORT, 105
- FORTCURRIND! *, 379
- FORTLINELEN
 - FORTLINELEN
 - *, 425
- FORTLINELEN! *, 380
- FORTTRAN, 104, 106, 326, 328
- FORTUPPER, 107
- FPS, 319
- FRAME command, 302, 306
- FREEOF, 39
- FULLROOTS, 84
- FUNCTION, 338
- Function, 172
- G, 770
- GAMMA, 67
- Gamma, 661
- Gamma function, 661
- GCD, 116, 117
- GCREF, 323
- Gegenbauer polynomials, 661
- GegenbauerP, 661

- GEN (I) , 651
- GENDECS switch, 348, 361, 376, 425
- GENDECS switch, 426
- Generalized Hypergeometric functions, 662
- generic function, 274
- GENSTMTINCR
 - GENSTMTINCR
 - *, 425
- GENSTMTINCR! *, 379
- GENSTMTNUM
 - GENSTMTNUM
 - *, 425
- GENSTMTNUM! *, 379
- GENTRAN, 325
- GENTRAN
 - command, 328
 - file output, 364
 - package, 325
 - preevaluation, 334, 353
 - templates, 352
- GENTRAN command, 375, 424
- GENTRAN package , 325
 - example, 330, 333, 334, 336–338, 342, 343, 345, 347, 349, 354, 369, 371, 373, 381, 384, 388, 395, 397, 399, 402
 - switches, 375
 - variables, 377
- GENTRANIN command, 352, 355, 357, 375, 424
- GENTRANLANG
 - GENTRANLANG
 - *, 425
- GENTRANLANG! *, 328, 353, 377
- GENTRANOPT switch, 327, 376
- GENTRANOUT command, 365, 405, 424
- GENTRANPOP command, 370, 372, 405, 424
- GENTRANPUSH command, 370, 405, 424
- GENTRANSEG switch, 343, 376, 425
- GENTRANSHUT command, 365, 368, 405, 424
- GETCSYSTEM command, 211
- GETDECS switch, 340, 376, 425
- GETROOT, 634
- GFNEWT, 634
- GFROOT, 634
- GL (I) , 651
- GNUPLOT, 442
- GNUPLOT package, 442
- GO TO, 55
- GOLDEN_RATIO, 32
- Gosper's Algorithm, 665
- GRAD
 - operator, 210
- grad operator, 588
- gradient
 - vector field, 210
- Graeffe's Root-Squaring Method, 381
- GROEBNER, 455
- Groebner, 82
- Groebner Bases, 532
- Group statement, 49, 53
- GUARDIAN, 455
- Hankel functions, 661
- HANKEL1, 67
- Hankel1, 661
- HANKEL2, 67
- Hankel2, 661
- Hermite polynomials, 661
- HermiteP, 661
- HFACTORS scale factors, 210
- High energy trace, 773
- High energy vector expression, 769, 772
- HIGH_POW, 110
- History, 155
- Hodge-* duality operator, 293, 303
- HYPOT, 67, 71
- I, 32
- i, 201
- IBETA, 67
- IDEALS, 473

- Identifier, 31
- IF, 49
- IFACTOR, 115
- IGAMMA, 67
- imaginary unit, 201
- IMPART, 64–66, 131
- IMPLICIT option, 337
- implicit type declarations, 344
- IMPLICIT! atom, 399
- IMPLICIT.TAYLOR operator, 672
- IN, 151
- Indefinite integration, 75
- INDEX, 770
- INDEX_SYMMETRIES command, 306
- INDEXRANGE command, 306
- INDEXSYMMETRIES command, 299
- INEQ, 476
- INFINITY, 32, 632
- INFIX, 93
- Infix operator, 34–36
- inner product, 587
- inner product
 - exterior form, 292
- INPUT, 156
- Input, 151
- Instant evaluation, 60, 112, 138, 160, 161
- INT, 75, 157
- INTEGER, 54
- Integer, 38
- INTEGER declaration, 340
- Integration, 75, 91
- integration
 - definite (simple), 212
 - line, 213
 - volume, 212
- interactive code generation, 381
- Interactive use, 155, 157
- INTERPOL, 121
- intrinsic functions, 350
- Introduction, 25
- INTSTR, 96
- INVBASE, 478
- INVERSE.TAYLOR, 672
- ISOLATER, 632
- Jacobi Elliptic Functions and Integrals, 661
- Jacobi's polynomials, 661
- Jacobiamplitude, 661
- Jacobicn, 661
- Jacobidn, 661
- JacobiP, 661
- Jacobisn, 661
- Jacobizeta, 661
- JOIN, 50
- KEEP command, 304, 306
- KEEPDECS switch, 341, 377, 425
- Kernel, 95, 99, 109
- kernel form, 96
- KHINCHIN, 32
- KORDER, 598
- KORDER, 109
- Kummer functions, 661
- KUMMER, 67
- KummerM, 661
- KUMMERU, 67
- KummerU, 661
- L'Hôpital's rule, 489, 589
- Label, 55
- Laguerre polynomials, 661
- LaguerreP, 661
- LAMBDA, 759
- Lambert's W, 82
- LAPLACE, 483
- Laplacian
 - vector field, 210
- Laurent series, 672
- LCM, 118
- LCOF, 122
- Leading coefficient, 122
- Legendre polynomials, 169, 661
- LegendreP, 661
- LENGTH, 43, 60, 77, 113, 115, 161
- LET, 69, 74, 87, 91–93, 136, 144, 170, 171
- Levi-Cevita tensor, 303

- LHS, [41](#)
- LIE, [485](#)
- Lie Derivative, [293](#)
- LIMIT, [489](#), [590](#)
- LIMIT+, [489](#)
- LIMIT-, [489](#)
- LIMIT0, [489](#)
- LIMIT1, [489](#)
- LIMIT2, [490](#)
- LIMITS, [489](#)
- LIMITS package, [489](#)
- LINALG, [491](#)
- line integrals, [213](#)
- LINEAR, [90](#)
- Linear Algebra package, [491](#)
- Linear operator, [90](#), [91](#), [93](#)
- LINEINT, [590](#)
- LINEINT function, [213](#)
- LINELENGTH, [98](#)
- LISP, [757](#)
- Lisp, [757](#)
- LIST, [100](#)
- List, [43](#)
- list, [80](#)
- List operation, [43](#), [45](#)
- LISTARGP, [45](#)
- LISTARGS, [45](#)
- LITERAL, [394](#), [402](#), [425](#), [426](#)
- LITERAL command, [421](#)
- literals
 - in GENTRAN, [402](#)
- LN, [67](#), [71](#)
- LOAD, [778](#)
- LOAD_PACKAGE, [175](#), [778](#)
- LOG, [67](#), [71](#), [76](#)
- LOG10, [67](#), [71](#)
- LOGB, [67](#), [71](#)
- Lommel functions, [661](#)
- LOMMEL1, [67](#)
- Lommel1, [661](#)
- LOMMEL2, [67](#)
- Lommel2, [661](#)
- Loop, [50](#), [51](#)
- loop indices
 - in GENTRAN, [340](#)
- LOW_POW, [111](#)
- LPDO, [521](#)
- LPOWER, [123](#)
- LRSETQ, [336](#), [394](#), [396](#), [424](#), [426](#)
- lrsetq operator, [334](#)
- LSETQ, [334](#), [394](#), [396](#), [424](#), [426](#)
- lsetq operator, [334](#)
- LTERM, [123](#), [767](#)
- MACRO, [761](#)
- MAINVAR, [123](#)
- MAKECALLS switch, [351](#), [377](#), [425](#)
- MAP, [77](#)
- map, [80](#)
- marking temporary variables, [346](#)
- MARKVAR, [346](#), [426](#)
- MASS, [772](#), [773](#)
- MAT, [159](#), [160](#)
- MATCH, [143](#)
- MATEIGEN, [162](#)
- Mathematical function, [67](#)
- matrices
 - in GENTRAN, [334](#), [335](#), [354](#)
- MATRIX, [159](#)
- Matrix assignment, [165](#)
- Matrix calculations, [159](#)
- MAX, [65](#)
- MAXEXPPRINTLEN
 - MAXEXPPRINTLEN
 - *, [425](#)
- MAXEXPPRINTLEN! *, [343](#), [376](#), [378](#)
- MCD, [116](#), [118](#)
- Meijer's G function, [662](#)
- METRIC command, [306](#)
- metric structure, [299](#)
- MIN, [65](#)
- MINCLINELEN
 - MINCLINELEN
 - *, [425](#)
- MINCLINELEN! *, [381](#)
- MINFORTLINELEN
 - MINFORTLINELEN
 - *, [425](#)

- MINFORTLINELEN! *, 381
- Minimum, 553
- MINPASCLINELEN
 - MINPASCLINELEN
 - *, 426
- MINPASCLINELEN! *, 381
- MINRATLINELEN
 - MINRATLINELEN
 - *, 425
- MINRATLINELEN! *, 381
- MKID, 79
- MKPOLY, 634
- MM, 651
- Mode, 60
- Mode communication, 762
- MODSR, 531
- MODULAR, 127
- Modular coefficient, 127
- Motzkin, 72
- Motzkin number, 72
- MSHELL, 773
- Multiple assignment statement, 48
- MULTIPLICITIES, 83
- MULTIROOT, 636
- NAT, 107
- NAT flag, 296
- NCPOLY, 532
- NEARESTROOT, 633, 635
- NEARESTROOTS, 634
- NEGATIVE, 632
- negativity, 132
- NERO, 104
- Newton's method, 553
- NEXTPRIME, 65
- NN, 651
- NOCOMMUTEDF, 73
- NOCONVERT, 126
- NODEPEND, 94
- NODEPEND statement, 589
- NOETHER function, 296, 306
- Non-commuting operator, 91
- NONCOM, 594
- NONCOM, 91
- NONZERO, 89
- NOREALVALUED, 132
- NORMFORM, 539
- NORMFORM package, 539
- NOSPLIT, 100
- NOSPUR, 773
- NOSUM command, 299, 306
- NOSUM switch, 299
- NOXPND
 - @, 291
 - D, 290
- NOXPND @ command, 306
- NOXPND command, 306
- NS dummy variable, 297
- NULLSPACE, 163
- NUM, 124
- NUM_INT, 553
- NUM_MIN, 553
- NUM_ODESOLVE, 553
- NUM_SOLVE, 553
- Number, 30
- NUMBERP, 39
- numeric code, 331
- NUMERIC package, 553
- Numerical operator, 63
- Numerical precision, 32
- ODD, 89
- Odd operator, 89
- ODESOLVE, 564
- OFF, 60, 61
- ON, 60, 61
- ONE_OF, 83
- OPAPPLY, 602
- OPERATOR, 766
- Operator, 34, 36
- Operator precedence, 35, 36
- OPORDER, 599
- optimization, code, 326
- ORDER, 98, 109
- ordering
 - exterior form, 304
- ORDP, 39, 91
- Orthogonal polynomials, 661

- ORTHOVEC, [583](#)
- ORTHOVEC package, [583](#)
- ORTHOVEC package
 - example, [591](#), [593](#)
- OUT, [151](#), [152](#)
- OUT command, [364](#)
- OUTPUT, [97](#)
- Output, [102](#), [106](#)
- Output declaration, [98](#)
- output redirection (temporary), [375](#), [405](#)
- Padget, Julian, [680](#)
- PART, [673](#)
- PART, [43](#), [109](#), [111](#)
- PART Operator, [677](#)
- PART operator, [673](#)
- partial derivatives, [274](#)
- partial differentiation, [289](#)
- PASCAL, [328](#)
- PASCCURRIND ! *, [380](#)
- PASCLINELEN
 - PASCLINELEN
 - *, [425](#)
- PASCLINELEN ! *, [380](#)
- PAUSE, [157](#)
- PCLASS, [651](#), [652](#), [654](#)
- Percent sign, [34](#)
- PERIOD, [107](#), [329](#)
- PERIOD switch, [377](#), [425](#)
- PF, [79](#)
- PFORM command, [306](#)
- PFORM statement, [287](#)
- PHYSINDEX, [596](#)
- PHYSOP, [594](#)
- PI, [33](#)
- PLOT, [442](#)
- PM, [607](#)
- POCHHAMMER, [67](#)
- Pochhammer, [79](#), [661](#)
- Pochhammer notation, [79](#)
- Pochhammer's symbol, [661](#)
- POLYGAMMA, [67](#)
- Polygamma, [661](#)
- Polygamma functions, [661](#)
- Polynomial, [113](#)
- Polynomial equations, [455](#)
- POSITIVE, [632](#)
- positivity, [132](#)
- power series, [680](#)
- power series
 - arithmetic, [688](#)
 - composition, [685](#)
 - differentiation, [688](#)
 - of integral, [681](#)
 - of user defined function, [681](#)
- Power series expansions, [680](#)
- PRECEDENCE, [93](#)
- PRECISE, [69](#), [70](#)
- PRECISE_COMPLEX, [70](#)
- PRECISION, [126](#), [637](#)
- precision, [331](#), [375](#)
- PRECISION command, [332](#)
- Prefix, [63](#), [92](#), [93](#)
- Prefix operator, [34](#), [35](#)
- PRET, [780](#), [781](#)
- PRETTYPRINT, [781](#)
- Prettyprinting, [780](#), [781](#)
- PRGEN, [650](#)
- PRI, [98](#)
- PRIMEP, [39](#)
- PRINT!-PRECISION command, [332](#)
- PRINT_PRECISION, [126](#)
- PROCEDURE, [167](#)
- Procedure body, [169](#), [170](#)
- Procedure heading, [168](#)
- PROD operator, [665](#)
- PRODUCT, [50](#), [51](#)
- Program, [33](#)
- Program structure, [29](#)
- Proper statement, [41](#), [47](#)
- PRSYS, [650](#), [653](#)
- PS, [680](#)
- PS operator, [680](#)
- PSCHANGEVAR operator, [684](#)
- PSCOMPOSE operator, [685](#)
- PSCOPY operator, [687](#)
- PSDEPVAR operator, [683](#)
- PSEUDO_DIVIDE, [119](#)

- PSEUDO_REMAINDER, 119
- PSEXPAISONPT operator, 684
- PSEXPLIM Operator, 682
- PSEXPLIM operator, 681
- PSFUNCTION operator, 684
- PSI, 67
- Psi, 661
- Psi function, 661
- PSORDER operator, 683
- PSORDLIM operator, 682
- PSREVERSE operator, 684
- PSSETORDER operator, 683
- PSSUM operator, 686
- PSTERM operator, 683
- PSTRUNCATE operator, 687
- Puiseux expansion, 685
- Puiseux series, 672
- PUTCSYSTEM command, 211

- Quadrature, 553
- QUIT, 61
- QUOTE, 759

- RANDOM, 66
- RANDOM_NEW_SEED, 66
- RANDPOLY, 607
- RANK, 164
- RAT, 101
- RATARG, 110, 121
- RATCURRIND! *, 379
- RATFOR, 326, 328
- RATIONAL, 125
- Rational coefficient, 125
- Rational function, 113
- rational number, 72
- RATIONALIZE, 128
- RATLINELEN
 - RATLINELEN
 - *, 425
- RATLINELEN! *, 380
- RATPRI, 101
- RATROOT, 636
- REACTEQN, 617
- REAL, 54
- Real, 30, 31
- Real coefficient, 125, 126
- REAL declaration, 340
- REAL*8, 340
- REALROOTS, 632, 634
- REALVALUED, 131
- REALVALUEDP, 132
- REDERR, 170
- REDLOG, 621
- REDUCT, 124
- REMAINDER, 118
- REMEMBER, 173
- REMFAC, 99
- REMFORDER command, 304, 306
- REMIND, 770
- RENOSUM command, 299, 306
- REPART, 64–66
- REPEAT, 53–55, 57
- REPEAT loop, 420
- REQUIREMENTS, 87
- Reserved variable, 32, 33
- reserved words, 411
- RESET, 621
- RESIDUE, 622
- REST, 44
- RESULT, 650
- RESULTANT, 119
- RETRY, 155
- RETURN, 54–56
- REVERSE, 45
- REVPRI, 101
- RHS, 41
- RIEMANNCONX command, 303, 306
- Riemannian Connections, 303
- RLFI, 626
- Rlisp, 777
- RLISP88, 768
- RLROOTNO, 633
- root finding, 631
- ROOT_VAL, 128
- ROOT_OF, 82, 83
- ROOT_VAL, 633
- ROOTACC#, 636
- ROOTMSG, 636

- ROOTPREC, [637](#)
- ROOTS, [631](#), [633](#), [634](#)
- ROOTS package, [631](#)
- ROOTS_AT_PREC, [633](#)
- ROOTSCOMPLEX, [633](#)
- ROOTSREAL, [633](#)
- ROUND, [67](#)
- ROUNDALL, [127](#)
- ROUNDBF, [126](#)
- ROUNDED, [32](#), [38](#), [71](#), [104](#), [126](#), [635](#)
- RSETQ, [334](#), [394](#), [396](#), [397](#), [424](#), [426](#)
- rsetq operator, [334](#)
- RSOLVE, [638](#)
- Rule lists, [143](#)
- SAVEAS, [97](#)
- SAVESTRUCTR, [108](#)
- Saving an expression, [107](#)
- SCALAR, [54](#)
- Scalar, [37](#)
- SCALEFACTORS operator, [211](#)
- SCALOP, [596](#)
- SCIENTIFIC_NOTATION, [30](#)
- SCOPE, [640](#)
- SDER (I) , [651](#)
- SEC, [67](#), [71](#)
- SECH, [67](#), [71](#)
- SECOND, [44](#)
- segmented assignments, [385](#)
- segmenting expressions, [343](#)
- SELECT, [80](#)
- Selector, [763](#)
- Semicolon, [47](#)
- SET, [48](#), [79](#)
- SETMOD, [127](#)
- SETS, [641](#)
- SGN
 - indeterminate sign, [295](#)
- SHARE, [762](#)
- SHARE command, [347](#), [398](#)
- SHOWRULES, [148](#)
- SHOWTIME, [61](#)
- SHUT, [151](#), [152](#)
- SHUT command, [364](#)
- SI, [67](#)
- Side effect, [41](#)
- side effects, [329](#)
- SIGN, [67](#), [132](#)
- SIGNATURE command, [306](#)
- Simplification, [38](#), [95](#)
- SIMPNONCOMDF, [74](#)
- SIMPSYS, [650](#), [652](#), [654](#)
- SIN, [67](#), [71](#)
- SINH, [67](#), [71](#)
- SixjSymbol, [661](#)
- SMACRO, [762](#)
- SolidHarmonicY, [661](#)
- SOLVE, [82](#), [83](#), [86](#), [455](#)
- SOLVE package
 - with ROOTS package, [631](#)
- SOLVESINGULAR, [86](#)
- SPACEDIM command, [289](#), [306](#)
- SPDE, [650](#)
- SPECFN, [69](#), [661](#)
- SPECFN2, [662](#)
- Spherical and Solid Harmonics, [661](#)
- spherical coordinates, [301](#), [584](#)
- SphericalHarmonicY, [661](#)
- SPLIT_FIELD function, [205](#)
- SPUR, [773](#)
- SQFRF, [636](#)
- SQRT, [67](#), [71](#)
- Standard form, [763](#)
- Standard quotient, [763](#)
- STATE, [596](#)
- Statement, [47](#)
- Stirling numbers, [661](#)
- Stirling1, [661](#)
- Stirling2, [661](#)
- String, [33](#)
- STRUCTR, [108](#)
- Structuring, [95](#)
- Struve functions, [661](#)
- STRUVEH, [67](#)
- StruveH, [661](#)
- STRUVEL, [67](#)
- StruveL, [661](#)
- Sturm Sequences, [632](#)

- SUB, [40](#), [135](#)
- SUBROUTINE, [338](#)
- subscripts
 - in GENTRAN, [339](#)
- Substitution, [135](#)
- SUCH THAT, [139](#)
- SUM, [50](#), [51](#), [665](#)
- SUM operator, [665](#)
- SUM-SQ, [665](#)
- SVEC, [585](#)
- Switch, [60](#), [61](#)
- SYM!-GENTRAN command, [393](#), [426](#)
- SYM!-GENTRANIN command, [404](#), [426](#)
- SYM!-GENTRANOUT command, [405](#), [426](#)
- SYM!-GENTRANPOP command, [408](#), [426](#)
- SYM!-GENTRANPUSH command, [407](#), [426](#)
- SYM!-GENTRANSHUT command, [406](#), [426](#)
- SYMBOLIC, [757](#)
- Symbolic mode, [757](#), [758](#), [762](#), [763](#)
- symbolic mode
 - in GENTRAN, [393](#)
- Symbolic procedure, [761](#)
- SYMMETRIC, [91](#)
- SYMMETRY, [667](#)
- SYMMETRY package, [667](#)
- system precision, [636](#)
- T, [33](#)
- TAB! *, [341](#), [402](#)
- TABLEN
 - TABLEN
 - *, [425](#)
- TABLEN! *, [380](#)
- TAN, [67](#), [71](#), [76](#)
- tangent vector, [290](#)
- TANH, [67](#), [71](#)
- TAYLOR, [672](#)
- TAYLOR operator, [672](#)
- TAYLOR package, [672](#)
- Taylor Series, [672](#)
- Taylor series
 - arithmetic, [674](#)
 - differentiation, [675](#)
 - integration, [675](#)
 - reversion, [675](#)
 - substitution, [675](#)
- TAYLORAUTOCOMBINE switch, [675](#)
- TAYLORAUTOEXPAND switch, [674](#), [675](#)
- TAYLORCOMBINE, [674](#)
- TAYLORKEEPORIGINAL, [674](#), [678](#)
- TAYLORKEEPORIGINAL switch, [674](#), [676](#)
- TAYLORORIGINAL, [678](#)
- TAYLORPRINTORDER switch, [676](#)
- TAYLORPRINTTERMS, [679](#)
- TAYLORPRINTTERMS variable, [673](#)
- TAYLORREVERT, [678](#)
- TAYLORSERIESP, [674](#)
- TAYLORTEMPLATE, [674](#), [678](#)
- TAYLORTOSTANDARD, [674](#)
- template processing, [388](#), [404](#)
- templates , [352](#)
 - file stack, [356](#)
- temporary variables
 - marking, [346](#)
 - names, [345](#)
 - type, [344](#)
 - unmarking, [348](#)
- TEMPVAR, [345](#), [426](#)
- TEMPVARNAME
 - TEMPVARNAME
 - *, [425](#)
- TEMPVARNAME ! *, [378](#)
- TEMPVARNUM
 - TEMPVARNUM
 - *, [425](#)
- TEMPVARNUM! *, [346](#), [378](#)
- TEMPVARTYPE
 - TEMPVARTYPE
 - *, [425](#)
- TEMPVARTYPE ! *, [344](#), [346](#), [378](#)
- TENSOP, [596](#)
- Terminator, [47](#)

- THIRD, [44](#)
- ThreejSymbol, [661](#)
- TIME, [60](#)
- TOTALDEG, [125](#)
- TP, [163](#)
- TPS, [680](#)
- TPS package, [490](#)
- TRA, [176](#)
- TRACE, [163](#)
- tracing
 - EXCALC, [303](#)
 - ROOTS package, [636](#)
 - SPDE package, [652](#)
 - SUM package, [666](#)
- TRFAC, [116](#)
- TRI, [690](#)
- TRIGFORM, [84](#)
- TRIGSIMP, [69](#), [691](#)
- TRINT, [176](#)
- TRROOT, [636](#)
- TRSUM, [666](#)
- true, [352](#)
- truncated power series, [680](#)
- TURTLE package, [701](#)
- TVECTOR command, [287](#), [306](#)
- type declarations, [361](#)
- U (ALFA) , [651](#)
- U (ALFA, I) , [651](#)
- UNIT, [597](#)
- unmarking temporary variables, [348](#)
- UNMARKVAR, [348](#), [426](#)
- UNTIL, [50](#)
- User packages, [175](#)
- VARDF, [295](#), [306](#)
- Variable, [32](#)
- Variable elimination, [455](#)
- variational derivative, [295](#)
- VARNAME, [107](#), [108](#)
- VAROPT, [88](#)
- VAXIMA, [325](#)
- VDF, [589](#)
- VEC command, [208](#)
- VECDIM, [775](#)
- VECOP, [596](#)
- VECTOR, [771](#)
- vector
 - addition, [587](#)
 - cross product, [587](#)
 - differentiation, [210](#)
 - division, [587](#)
 - dot product, [587](#)
 - exponentiation, [587](#)
 - inner product, [587](#)
 - integration, [210](#)
 - modulus, [587](#)
 - multiplication, [587](#)
 - subtraction, [587](#)
- vector algebra, [208](#)
- VECTORADD, [587](#)
- VECTORCROSS, [587](#)
- VECTORDIFFERENCE, [587](#)
- VECTOREXPT, [587](#)
- VECTORMINUS, [587](#)
- VECTORPLUS, [587](#)
- VECTORQUOTIENT, [587](#)
- VECTORRECIP, [587](#)
- VECTORTIMES, [587](#)
- VERBOSELOAD switch, [676](#)
- VINT, [590](#)
- VMOD, [587](#)
- VMOD operator, [209](#)
- VOLINT, [590](#)
- VOLINTEGRAL function, [212](#)
- VOLINTORDER vector, [213](#)
- VORDER, [589](#)
- VOUT, [585](#)
- VSTART, [584](#)
- VTAYLOR, [589](#)
- warnings
 - TAYLOR package, [676](#)
- wedge, [306](#)
- WEIGHT, [150](#)
- WHEN, [144](#)
- WHERE, [145](#)
- WHILE, [52](#), [54](#), [55](#), [57](#)

WHILE loop, [420](#)
Whittaker functions, [661](#)
WHITTAKERM, [67](#)
WhittakerM, [661](#)
WHITTAKERU, [67](#)
WhittakerW, [661](#)
Workspace, [96](#)
WRITE, [102](#)
WS, [27](#), [156](#)
WTLEVEL, [150](#)
WU, [715](#)

X(I), [651](#)
XCOLOR, [717](#)
XI(I), [651](#)
XIDEAL, [719](#)
XPND
 @, [291](#), [306](#)
 D, [291](#)
XPND command, [306](#)

ZEILBERG, [727](#)
ZETA, [67](#)
Zeta, [661](#)
Zeta function (Riemann's), [661](#)
ZETA(ALFA, I), [651](#)
ZTRANS, [748](#)